Linux* 版インテル® C++ コンパイラ

ユーザーズ・ガイド

© 1996 - 2004, Intel Corporation.

無断での引用、転載を禁じます。

資料番号: 253254-014J

目次

免責条項 1

インテル® C++ コンパイラへようこそ 3

インテル® C++ コンパイラへようこそ 3

本リリースの新機能 3

特徴と利点 4

製品情報 Web サイトとサポート 4

システム要件 5

IA-32 プロセッサのシステム要件 5

Itanium® プロセッサのシステム要件 5

ソフトウェアの要件 5

FLEXlm* 電子ライセンス 5

参考文献 5

本書の使い方 7

表記法 7

組込み関数名の表記 8

クラス・ライブラリ名の表記 9

コンパイラ・オプションのクイック・リファレンス 10

概要: オプション・クイック・リファレンス・ガイド 10

オプションのクイック・ガイドの表で使用される表記 10

新しいオプション 10

オプションのクイック・リファレンス・ガイド 13

コンパイラ・オプションの対応表 32
デフォルトのコンパイラ・オプション 37
アプリケーションのビルドとデバッグ 39
コンパイラの起動 39
コンパイラのデフォルトの動作 39
コンパイル・フェーズ 39
コマンドラインからのアプリケーションのビルド 40
コンパイラの起動 40
コマンドラインからのコンパイラの起動 40
コマンドラインからの makefile の実行 42
コンパイラ入力ファイル 42
コンパイル・オプション 43
概要: コンパイル 43
概要: プリプロセッサ・オプション 43
前処理オプション 44
前処理のみを行う 44
前処理ディレクティブと同じ働きをするオプション 45
事前定義済みマクロ 46
コンパイル環境 49
コンパイルの制御 53
コンパイル・フローの制御 54
コンパイル出力の制御 54

代替ツールと代替パスの指定 55
データ設定の監視 56
プリコンパイル済みヘッダファイル 57
リンク 60
リンク 60
デバッグ 61
概要: デバッグ・オプション 61
構文解析のみを行う 61
最適化とデバッグ 61
ライブラリの使用 64
概要: ライブラリの使用 64
GNU C ライブラリ 64
Dinkumware C++ ライブラリ 64
標準 C++ ライブラリ 64
デフォルト・ライブラリ 64
数値演算ライブラリ 65
インテル共有ライブラリ 66
共有ライブラリのオプション 66
ライブラリの管理 66
LD_LIBRARY_PATH の変更 67
非共有ライブラリのコンパイル 67
概要: 非共有ライブラリのコンパイル 67

グローバル・シンボルと可視属性 67
シンボル・プリエンプション 69
シンボルの可視属性の明示的な指定 70
その他の可視属性関連のコマンドライン・オプション 71
gcc* との互換性 73
gcc* との互換性 73
gcc* との相互運用性 76
相互運用性のコンパイラ・オプション 77
デフォルトのライブラリとヘッダ 78
gcc 事前定義済みマクロ 79
gcc 組込み関数 79
gcc 関数属性 80
例 80
言語の適合性 81
適合性オプション 81
C の標準規格との適合性 81
ANSI/ISO 標準準拠の C 言語について 81
コンパイラ付属のマクロ 82
C99 サポート 82
C++ の標準規格との適合性 83
コンパイラの最適化 84
最適化レベル 84

概要: 最適化レベル 84
最適化レベルの設定 84
最適化の範囲の制限 85
浮動小数点の最適化 86
浮動小数点演算の精度 86
Itanium® ベース・システムに使用する浮動小数点演算オプション 89
特定のプロセッサの最適化 90
プロセッサの最適化 90
プロセッサ固有の最適化 (IA-32 のみ) 92
特定のプロセッサに合わせて自動的に最適化する (IA-32 のみ) 93
手動 CPU ディスパッチ (IA-32 のみ) 94
プロセッサ固有のランタイム・チェック (IA-32 システム) 96
プロシージャ間の最適化 98
概要: プロシージャ間の最適化 98
プロシージャ間の最適化のオプション 99
-ip または -ipo と -Qoption 指示子の使用 99
関数のインライン展開 101
マルチファイル IPO 102
概要: マルチファイル IPO 102
実際のオブジェクト・ファイルの生成 103
マルチファイル IPO 実行ファイルの作成 104
xild を使用したマルチファイル IPO 実行ファイルの作成 105

IPO オブジェクトからのライブラリの作成 106
マルチファイル IPO の効果の分析 107
プロファイルに基づく最適化 108
概要: プロファイルに基づく最適化 108
プロファイルに基づく最適化の方法 108
基本的な PGO オプション 109
プロファイルに基づく最適化の例 111
PGO の環境変数 112
profmerge ツールによるソースファイルの再配置 112
コード・カバレッジ・ツール 113
テスト・プライオリタイゼーション・ツール 121
PGO API: プロファイル情報生成サポート 128
高水準言語の最適化 (HLO) 131
概要: 高水準言語の最適化 131
ループ変換 132
IVDEP ディレクティブによるループ・キャリー・メモリ依存の不在 132
並列プログラミング 134
概要: 並列プログラミング 134
ベクトル化 (IA-32 のみ) 134
概要: ベクトル化 134
ベクトライザのオプション 135
ループの並列化とベクトル化 136

ベクトル化プログラミングの基本となるガイドライン 136
データ依存性 138
ループの構成要素 139
ループ出口条件 140
ベクトル化ループの種類 141
ストリップ・マイニングとクリーンアップ 141
ループ本体内の文 142
言語サポートとディレクティブ 143
ベクトル化の具体例 147
ループ交換と添字: マトリックス乗算 150
自動並列化 151
概要: 自動並列化 151
自動並列化のプログラミング 152
自動並列化: 有効、オプション、および環境変数 154
自動並列化のしきい値と診断 154
OpenMP* による並列化 156
概要: OpenMP* による並列化 156
並列処理スレッドモデル 157
OpenMP、ディレクティブ形式、および診断でのコンパイル 160
OpenMP* ディレクティブと節 162
OpenMP* のサポート・ライブラリ 163
OpenMP* の環境変数 164

OpenMP* ランタイム・ライブラリ・ルーチン 165
OpenMP* の使用例 167
OpenMP* に追加されたインテル拡張機能 168
概要: インテルのワークキューイング・モデル 168
インテル拡張機能 168
ワークキューイング・構造 170
関数例 174
最適化サポート機能 176
概要: 最適化サポート機能 176
コンパイラ・ディレクティブ 176
コンパイラ・ディレクティブ 176
Itanium® ベース・アプリケーションのパイプライン化 176
ループカウントとループ分配 177
ループのアンロールのサポート 178
プリフェッチのサポート 179
ベクトル化のサポート (IA-32) 180
最適化機構レポートの作成 182
レポートを作成する最適化の指定 183
アプリケーションの時間測定 184
コンパイラの制限 186
コンパイラの制限 186
主要なファイル 187

IA-32 コンパイラの主要なファイルの概要 187
/bin ファイル 187
/include ファイル 187
/lib ファイル 188
Itanium® コンパイラの主要なファイルの概要 189
/bin ファイル 189
/include ファイル 190
/lib ファイル 190
診断およびメッセージ 192
概要: 診断およびメッセージ 192
診断メッセージ 192
言語診断 192
lint のコメントを使用した警告メッセージの出力停止 193
警告メッセージの出力停止方法とリマーク・メッセージの出力方法 193
エラーの出力個数の制限 195
リマーク・メッセージ 195
インテル® 数値演算ライブラリ 196
概要: インテル® 数値演算ライブラリ 196
IA-32 と Itanium® ベース・システムの数値演算ライブラリ 196
インテル® 数値演算ライブラリの使用 196
実関数の使用例 196
複素数関数の使用例 197

数値演算関数 200
三角関数 200
双曲線関数 204
指数関数 206
特殊関数 211
丸め関数 215
剰余関数 218
その他の関数 219
複素数関数 224
C99 マクロ 230
インテル® C++ 組込み関数リファレンス 232
はじめに 232
概要: 組込み関数 232
組込み関数を使用するメリット 233
命名と使用する構文 236
すべての IA プロセッサでサポートされる組込み関数 238
概要: すべての IA プロセッサでサポートされる組込み関数 238
整数演算に関連する組込み関数 238
浮動小数点に関連する組込み関数 238
文字列とブロックのコピーに関連する組込み関数 240
組込み関数 241
MMX® テクノロジの組込み関数 242

MMX® テクノロジのサポート 242
EMMS 命令: 必要な理由 243
EMMS を使用する際のガイドライン 244
MMX® テクノロジの一般的な組込み関数 244
MMX® テクノロジのパックド算術演算組込み関数 246
MMX® テクノロジのシフト組込み関数 249
MMX® テクノロジの論理演算組込み関数 252
MMX® テクノロジの比較組込み関数 253
MMX® テクノロジの設定組込み関数 254
Itanium® アーキテクチャ上での MMX® テクノロジ組込み関数の使用 256
ストリーミング SIMD 拡張命令 257
概要: ストリーミング SIMD 拡張命令 257
ストリーミング SIMD 拡張命令を使用する浮動小数点演算組込み関数 257
ストリーミング SIMD 拡張命令の算術演算 258
ストリーミング SIMD 拡張命令の論理演算 262
ストリーミング SIMD 拡張命令の比較操作 263
ストリーミング SIMD 拡張命令の変換操作 270
ストリーミング SIMD 拡張命令のロード操作 273
ストリーミング SIMD 拡張命令の設定操作 274
ストリーミング SIMD 拡張命令のストア操作 275
ストリーミング SIMD 拡張命令によるキャッシュ制御 276
ストリーミング SIMD 拡張命令を使用する整数演算組込み関数 277

ストリーミング SIMD 拡張命令を使用するメモリ操作と初期化操作..... 280
ストリーミング SIMD 拡張命令を使用するその他の組込み関数..... 285
Itanium® アーキテクチャ上でのストリーミング SIMD 拡張命令の使用..... 287
ストリーミング SIMD 拡張命令を使用してシャッフルを行うマクロ関数..... 289
コントロール・レジスタを読み書きするマクロ関数 290
行列の転置を行うマクロ関数..... 292
ストリーミング SIMD 拡張命令 2 292
概要: ストリーミング SIMD 拡張命令 2 の組込み関数..... 292
ストリーミング SIMD 拡張命令 2 の浮動小数点算術演算..... 293
ストリーミング SIMD 拡張命令 2 の論理演算..... 296
ストリーミング SIMD 拡張命令 2 の比較操作..... 297
ストリーミング SIMD 拡張命令 2 の変換操作..... 304
ストリーミング SIMD 拡張命令 2 の浮動小数点メモリ操作と初期化操作..... 307
ストリーミング SIMD 拡張命令 2 のロード操作..... 307
ストリーミング SIMD 拡張命令 2 の設定操作..... 309
ストリーミング SIMD 拡張命令 2 のストア操作 310
ストリーミング SIMD 拡張命令 2 のその他の操作..... 311
ストリーミング SIMD 拡張命令 2 の整数算術演算 312
ストリーミング SIMD 拡張命令 2 の整数論理演算 318
ストリーミング SIMD 拡張命令 2 の整数シフト操作 319
ストリーミング SIMD 拡張命令 2 の整数比較操作 323
ストリーミング SIMD 拡張命令 2 の変換操作..... 325

シャッフルを行うマクロ関数 326
ストリーミング SIMD 拡張命令 2 のキャッシュ操作 327
ストリーミング SIMD 拡張命令 2 のその他の操作 329
ストリーミング SIMD 拡張命令 2 の整数メモリ操作と初期化操作 333
ストリーミング SIMD 拡張命令 2 の整数ロード演算 334
ストリーミング SIMD 拡張命令 2 の整数設定操作 334
ストリーミング SIMD 拡張命令 2 の整数ストア操作 337
新しい IA-32 組込み関数 338
概要: 新しい IA-32 組込み関数 338
マクロ関数 338
浮動小数点ベクトル組込み関数 338
整数ベクトル組込み関数 340
その他の組込み関数 340
Itanium® 命令の組込み関数 341
概要: Itanium® 命令の組込み関数 341
Itanium® 命令のネイティブ組込み関数 341
ロックおよびアトミック操作に関連する組込み関数 344
ロードとストア 347
オペレーティング・システムに関連する組込み関数 347
変換組込み関数 351
getReg() および setReg() のレジスタ名 351
マルチメディア乗算 354

同期プリミティブ 361

その他の組込み関数 362

データのアライメント、メモリの割り当て組込み関数、およびインライン・アセンブリ 362

概要: データのアライメント、メモリの割り当て組込み関数、およびインライン・アセンブリ 362

アライメントのサポート 363

アライメントの合ったメモリブロックの割り当てと解放 365

インライン・アセンブリ 365

各種のプロセッサでの組込み関数の使用 367

各種のプロセッサでの組込み関数の使用 367

すべての IA プロセッサでサポートされる組込み関数 368

MMX® テクノロジの組込み関数 370

ストリーミング SIMD 拡張命令の組込み関数 372

ストリーミング SIMD 拡張命令 2 の組込み関数 377

インテル® C++ クラス・ライブラリ 384

クラス・ライブラリの概要 384

クラス・ライブラリの概要 384

ハードウェアとソフトウェアの要件 384

クラスについて 384

ライブラリの詳細 385

C++ クラスと SIMD 演算 386

機能 389

整数ベクトルクラス 391
概要: 整数ベクトルクラス 391
用語、規則、および構文 393
演算子の規則 394
代入演算子 396
論理演算子 397
加算演算子と減算演算子 398
乗算演算子 400
シフト演算子 402
比較演算子 403
条件付き選択演算子 405
デバッグ 407
アンパック演算子 410
パック演算子 416
MMX® テクノロジ・ステート消去演算子 416
ストリーミング SIMD 拡張命令用の整数組込み関数 417
Fvec と Ivec 間の変換 418
概要: 浮動小数点ベクトルクラス 419
概要: 浮動小数点ベクトルクラス 419
Fvec の表記法 420
データのアライメント 421
変換 421

コンストラクタと初期化 422
算術演算子 423
最小値演算子と最大値演算子 426
論理演算子 427
比較演算子 428
Fvec クラスの条件付き選択演算子 431
キャッシュ操作 433
デバッグ 433
ロード演算子とストア演算子 435
Fvec 演算子のアンパック演算子 436
マスク移動演算子 436
各種クラスのクイック・リファレンス 437
プログラミング例 441
キーワード 443

免責条項

【輸出規制に関する告知と注意事項】

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

【資料内容に関する注意事項】

本ドキュメントの内容を予告なしに変更することがあります。

インテルでは、この資料に掲載された内容について、市販製品に使用した場合の保証あるいは特別な目的に合うことの保証等は、いかなる場合についてもいたしかねます。また、このドキュメント内の誤りについても責任を負いかねる場合があります。

インテルでは、インテル製品の内部回路以外の使用にて責任を負いません。また、外部回路の特許についても関知いたしません。

本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。

インテルは、製品について「取引条件」で提示されている場合を除き、インテル製品の販売や使用に関して、いかなる特許または 著作権の侵害をも含み、あらゆる責任を負わないものとします。

本書に含まれている内容は、出荷時の内容を正確に著すよう記述されていますが、製品の不具合の発見とその改良に伴い、製品および本書の内容は予告なく変更される場合があります。現在報告されているソフトウェアの不具合につきましては、お問い合わせください。

いかなる形および方法によっても、インテルの文書による許可なく、この資料の一部またはすべてを複写することは禁じられてい ます。

著作権法で許可されている場合を除き、文書による事前の許可なく、複製、改変、または翻訳することを禁じます。無断転載を禁じます。

版権制限：米国政府による使用、複製、または開示は、DFARS 252-227-7013 の条項「Rights in Technical Data and Computer Software」の副項(c)(I)(ii)に規定されている制限を受けます。

非 DOD U.S. Government Departments and Agencies の権利は、FAR 52.227-19(c0(1,2)に規定されています。

Intel、インテル、Celeron、Dialogic、iCOMP、Itanium、MMX、Pentium、i386、i486、Intel386、Intel486、Intel740、IntelDX2、IntelDX4、IntelSX2、Intel SpeedStep、Intel NetBurst、Intel XScale、NetStructure、Xeon、VTune は、アメリカ合衆国およびその他の国における Intel Corporation またはその子会社の商標または登録商標です。

* その他の名称およびブランド名は、各社の商標および登録商標です。

© 1996 - 2004, Intel Corporation.

インテル® C++ コンパイラへようこそ

インテル® C++ コンパイラへようこそ

インテル® C++ コンパイラへようこそ。コンパイラを使用する前に、「システム要件」を参照してください。

ほとんどの Linux* ディストリビューションには、GNU* C ライブラリ、アセンブラ、リンカなどが含まれます。インテル C++ コンパイラには Dinkumware* C++ ライブラリが含まれています。「概要: ライブラリの使用」を参照してください。

個々のトピックの内容を表示するには、このユーザーズ・ガイドの各メイン・セクションにある個々のセクションの説明を参照してください。最新情報については、次のインテル Web サイトにアクセスしてください: http://www.intel.com/jp/developer/index.htm

本リリースの新機能

このバージョンのインテル® C++ コンパイラには、次の新機能が含まれています:

- 新しい gcc との相互運用性オプション
- 改良された gcc との互換性
- プリコンパイル済みヘッダファイルのサポート
- 新しい gcc 組込み関数
- 新しい gcc 関数属性
- HT テクノロジ インテル® Pentium® 4 プロセッサ (SSE3 対応) 用の新しい最適化のサポート
- IA-32 システム用の新しいプロセッサ固有のランタイム・チェック
- HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) 用の新しい IA-32 組込み関数
- Itanium® ベース・システム用の新しい同期プリミティブ組込み関数
- 新しいコード・カバレッジ・ツールとテスト・プライオリタイゼーション・ツール
- 新しいシンボルの可視属性オプション
- 新しい IPO のデバッグのサポート
- 更新されたインテル数値演算ライブラリ
- その他の新しいコンパイラ・オプション
- コマンドラインからコンパイラを起動するための新機能

新機能についての詳細は、リリースノートを参照してください。

特徴と利点

インテル® C++ コンパイラを使用すると、インテル® アーキテクチャ・ベースのコンピュータ上でソフトウェアの最高のパフォーマンスを引き出すことができます。インテル C++ コンパイラは、プログラム全体の最適化やプロファイルに基づく最適化などの新しいコンパイラ最適化、プリフェッチ命令、およびストリーミング SIMD 拡張命令 (SSE) とストリーミング SIMD 拡張命令 2 (SSE2) のサポートにより、高いパフォーマンスを提供します。

特徴	利点
高いパフォーマンス	組込み関数を使用しない同等のコードと比べて、パフォーマンスが大きく向上
ストリーミング SIMD 拡張命令のサポート	インテル・マイクロアーキテクチャの利点
自動ベクトライザ	コード内の SIMD 自動並列処理の利点
OpenMP* のサポート	共用メモリ並列プログラミング
浮動小数点の最適化	浮動小数点のパフォーマンスが向上
データ・プリフェッチ機能	データ送信の高速化によりパフォーマンスが向上
プロシージャ間の最適化	大規模アプリケーションのモジュールのパフォーマンスが向上
プロファイルに基づく最適化	頻繁に使用される関数のプロファイリングに基づくパフォーマンスの向上
プロセッサ・ディスパッチ	最新のインテル・アーキテクチャの機能を利用すると同時に、前世代のインテル® Pentium® プロセッサとのオブジェクト・コードの互換性を確保 (IA-32 ベースのシステムのみ)

製品情報 Web サイトとサポート

インテル® C++ コンパイラに関する最新情報は、
http://www.intel.co.jp/jp/developer/software/products/index.htm をご覧ください。

Itanium® アーキテクチャに関する詳細は、
http://www.intel.co.jp/jp/developer/design/itanium/itanium/index.htm をご覧ください。

システム要件

IA-32 プロセッサのシステム要件

- Pentium® プロセッサ、または IA-32 ベースのプロセッサを搭載したコンピュータ (Pentium 4 プロセッサを推奨)。
- 128 MB の RAM (256 MB を推奨)。
- 100 MB のディスク容量。

Itanium® プロセッサのシステム要件

- Itanium プロセッサを搭載するコンピュータ
- 256 MB の RAM。
- 100 MB のディスク容量。

ソフトウェアの要件

システム要件の一覧はリリースノートを参照してください。

FLEXlm* 電子ライセンス

インテル® C++ コンパイラは、GlobeTrotter* 社の FLEXlm という電子ライセンス技術を使用しています。この技術を使用するには、インストール・パスの `/licenses` ディレクトリに有効なライセンス・ファイルが含まれている必要があります。デフォルトのディレクトリは `/opt/intel_cc_80/licenses` です。ライセンス・ファイルの拡張子は `.lic` です。

ライセンスが必要な場合、『*Using the Intel License Manager for FLEXlm** 』(英語) を参照してください。

参考文献

インテル® C++ コンパイラの関連情報については、次の資料を参照してください:

- 『ISO/IEC 9989:1990, Programming Languages--C 』
- 『ISO/IEC 14882:1998, Programming Languages--C++ 』
- 『 *The Annotated C++ Reference Manual* 』第 3 版。Ellis Margaret、Stroustrup Bjarne 著、Addison Wesley 刊。1991 年。C++ プログラミング言語の解説

- 『*The C++ Programming Language* 』第 3 版。1997 年。Addison-Wesley Publishing Company (One Jacob Way, Reading, MA 01867) 刊。
- 『*The C Programming Language* 』第 2 版。Kernighan Brian W、Ritchie Dennis W 著。Prentice Hall 刊。1988 年。C 言語の K & R 定義の解説。
- 『*C: A Reference Manual* 』第 3 版。Harbison Samual P、Steele Guy L 著。Prentice Hall 刊。1991 年。C 言語の ANSI 標準と拡張機能の解説。
- 『*IA-32 インテル® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、上巻:基本アーキテクチャ* 』インテル社、資料番号 245470J
- 『*IA-32 インテル® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、中巻:命令セット・リファレンス* 』インテル社、資料番号 245471J
- 『*IA-32 インテル® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、下巻:システム・プログラミング・ガイド* 』インテル社、資料番号 245472J
- 『*インテル® Itanium® ベース・アセンブラ・ユーザズ・ガイド* 』
- 『*インテル® Itanium® アーキテクチャ・アセンブリ言語リファレンス・マニュアル* 』
- 『*インテル® Itanium® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、第 1 巻:アプリケーション・アーキテクチャ* 』インテル社、資料番号 245317J-001
- 『*インテル® Itanium® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、第 2 巻:システム・アーキテクチャ* 』インテル社、資料番号 245318J-001
- 『*インテル® Itanium® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、第 3 巻:命令セット・リファレンス* 』インテル社、資料番号 245319J-001
- 『*インテル® Itanium® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、第 4 巻:Itanium プロセッサ・プログラマーズ・ガイド* 』インテル社、資料番号 245319J-001
- 『*インテル® アーキテクチャ最適化リファレンス・マニュアル* 』インテル社、資料番号 245127J
- 『*インテル® プロセッサの識別と CPUID 命令* 』インテル社、資料番号 241618J。
- 『*Intel® Architecture MMX® Technology Programmer's Reference Manual* 』インテル社、資料番号 241618。
- 『*Pentium® Pro Processor Developer's Manual* 』(3 巻セット)、インテル社、資料番号 242693
- 『*PentiumII Processor Developer's Manual* 』インテル社、資料番号 243502-001
- 『*Pentium® Processor Specification Update* 』インテル社、資料番号 242480
- 『*Pentium® Processor Family Developer's Manual* 』インテル社、資料番号 241428-005

インテルのほとんどの資料は、インテルの Web サイト (http://www.intel.co.jp) から入手できます。

本書の使い方

このユーザーズ・ガイドでは、インテル® C++ コンパイラの使用方法を説明します。インテル C++ コンパイラの入門、動作内容、そして高いパーフォーマンスを実現する能力に関する情報を提供します。読者はアプリケーションのパフォーマンスを向上させるための標準的および先進的なコンパイラ最適化手法の使用法を学習できます。

標準的な最適化手法と高度な最適化手法を学べば、アプリケーションのパフォーマンスが最大まで引き出せます。また、ホスト・コンピュータのオペレーティング・システムにも精通している必要があります。

注

本書では、対象となる各アーキテクチャごとに情報や命令がどのように適用されるかを説明しています。どのアーキテクチャかを明示していない場合、説明は両方のアーキテクチャに適用されます。

表記法

本書では、次の表記法を使用しています:

`This type style`	構文要素、予約語、キーワード、ファイル名、コンピュータ出力、プログラム例の一部分のいずれかを表します。テキストは、大文字に意味がない限り小文字で表記します。
`This type style`	キーボードから入力する文字を表します。
`This type style`	識別子、式、文字列、シンボル、または値のプレースホルダを表します。プレースホルダの代わりにこれらの項目の 1 つを使用してください。
`[ items ]`	角括弧で囲まれたアイテムはオプションです。
`{ item1 | item2 |... }`	オプションのバージョンに使用されます。例えば、オプション `-x{K|W|N|B|P}` にはバージョン `-xK`、`-xW`、`-xN`、`-xB`、および `-xP` があります。
`...` （省略記号）	省略記号は、直前のアイテムを複数指定できることを表します。

組込み関数名の表記

ほとんどの組込み関数名は、次の表記法に従います:

_mm_<*intrin_op*>_<*suffix*>

<*intrin_op*>	組込み関数の基本操作を示します。例えば、加算の場合は add、減算の場合は sub になります。
<*suffix*>	命令の操作対象となるデータの型を示します。各サフィックスの最初の 1 文字または 2 文字は、データがパックドデータ (p)、拡張パックドデータ (ep)、またはスカラデータ (s) であることを示します。その他の文字は、データ型を示します: • __s 単精度浮動小数点値 • __d 倍精度浮動小数点値 • __i128 符号付き 128 ビット整数 • __i64 符号付き 64 ビット整数 • __u64 符号なし 64 ビット整数 • __i32 符号付き 32 ビット整数 • __u32 符号なし 32 ビット整数 • __i16 符号付き 16 ビット整数 • __u16 符号なし 16 ビット整数 • __i8 符号付き 8 ビット整数 • __u8 符号なし 8 ビット整数

変数名を付加した数字は、パックされたオブジェクトの要素を示します。例えば、r0 は r の最下位ワードです。一部の組込み関数は、2 つ以上の命令で実行するため、"複合組込み関数" と呼ばれます。

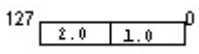
パックされた値は、右から左の順序で表現し、最下位の値がスカラ操作に使用されます。次の操作の例について考えます:

```
double a[2] = {1.0, 2.0}; __m128d t = _mm_load_pd(a);
```

上の操作の結果は、次のそれぞれの操作と同じになります:

```
__m128d t = _mm_set_pd(2.0, 1.0); __m128d t =
_mm_setr_pd(1.0, 2.0);
```

つまり、値 t を保持する xmm レジスタは、次のようになります:



"スカラ" 要素は 1.0 です。一部の組込み関数では、命令の性質上、引数として即値 (定数整数リテラル) を指定しなければなりません。

組込み関数については、「命名と使用する構文」も参照してください。

クラス・ライブラリ名の表記

クラス名は、データ型、符号の有無、ビットサイズ、要素数を表現したものです。一般的な形式で表すと次のようになります:

```
<type><signedness><bits>vec<elements>
{ F | I } { s | u } { 64 | 32 | 16 | 8 } vec { 8 | 4 | 2 | 1 }
```

各アイテムの意味は次のとおりです:

<type>	浮動小数点 (F) または整数 (I) を示します。
<signedness>	符号付き (s) または符号なし (u) を示します。Ivec クラスの場合は、このフィールドが空のままだと中間クラスを表します。符号なしの Fvec クラスはないため、Fvec クラスの場合、このフィールドは空です。
<bits>	要素あたりのビット数です。
<elements>	要素の個数です。

コンパイラ・オプションのクイック・リファレンス

概要: オプション・クイック・リファレンス・ガイド

オプションのクイック・ガイドの表で使用される表記

表記	定義
[-]	オプションに "[-]" が含まれている場合、このオプションを使用して機能を有効または無効にすることができます。例えば、-c99[-] オプションは -c99 (c99 サポートを有効にする) または -c99- (c99 サポートを無効にする) して使用することができます。
[n]	[ ] の値 n は、省略できるか、任意の値を指定できることを示します。
{ } の縦バーの値	オプションのバージョンに使用されます。例えば、オプション -x{K|W|N|B|P} にはバージョン -xK、-xW、-xN、-xB、および -xP があります。
{n}	オプションで、n に固定値の 1 つを指定する必要があることを示します。
オプションに続く*このスタイル*のワード	オプションに必要な引数を表します。複数の引数が必要な場合は、カンマで区切ります。

新しいオプション

- IA-32 アーキテクチャ固有のオプション
- Itanium® アーキテクチャ固有のオプション (Itanium ベース・システムのみ)
- IA-32、Itanium ベース・システムの両方で使用できるオプション

オプション	説明	デフォルト
-alias_args[-]	引数がエイリアス化された [エイリアス化されない] 可能性を示します。	-alias_args
-auto_ilp32 Itanium ベース・システムのみ	32 ビットアドレス空間を超えることができない (32 ビットポインタを使用する) アプリケーションを指定します。このオプションを使用するには、-ipo も指定してください。32 ビットアドレス空間を超えることができるプログラムで -auto_ilp32 オプションを使用すると、プログラム実行中に予期できない問題が発生することがあります。	オフ

-axB IA-32 のみ	インテル® Pentium® M プロセッサおよび互換性のあるインテル® プロセッサ専用のコードを生成します	オフ
-axN IA-32 のみ	インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ専用のコードを生成します	オフ
-axP IA-32 のみ	HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) 専用のコードを生成します	オフ
-complex_limited_range[-]	タイプ _Complex のデータに関する一部の算術演算で "基礎代数展開の削除" を有効にします。これは _Complex 算術を使用するプログラムでパフォーマンスを向上しますが、指数範囲の値を正しく計算しません。デフォルトは -complex_limited_range- です。	オフ
-create_pch *filename*	プリコンパイル済みヘッダ (*filename*.pchi) を作成します。	オフ
-cxxlib-gcc	gcc の C++ ランタイム・ライブラリを使用してリンクします (gcc 3.2 以降が必要です)	オフ
-cxxlib-icc	インテルの C++ ランタイム・ライブラリを使用してリンクします。	オン
-fast	プログラム全体を最高速で実行します(-O3、-ipo、-static をオンにします)。	オフ
-fminshared	主実行ファイル用にコンパイルを行います。絶対アドレス指定を使用することができ、保護されているシンボルには位置独立でないコードが生成されます。	オフ
-fno-common	コンパイラが共通変数を定義されているものとして処理し、共通データ変数の gprel アドレス指定を使用できるようにします。	オフ
-fpstkchk IA-32 のみ	FP スタックが予測された状態であることを保証するためにすべての関数呼び出しの後に補足コードを生成します。	オフ
-fvisibility= [extern|default|protected |hidden|internal]	グローバル・シンボル (共通データ、定義データ、関数) の可視属性はデフォルトで設定されます。シンボルの可視属性はソースコードで明示的に設定されます。また、シンボルの可視属性の file オプションは -fvisibility 設定よりも優先されます。	オフ
-fvisibility-extern=*file*	*file* 引数でリストされたシンボルで区切られた空間の可視属性を extern に設定します。	オフ

-fvisibility-default=*file*	*file* 引数でリストされたシンボルで区切られた空間の可視属性を default に設定します。	オフ
-fvisibility-protected=*file*	*file* 引数でリストされたシンボルで区切られた空間の可視属性を protected に設定します。	オフ
-fvisibility-hidden=*file*	*file* 引数でリストされたシンボルで区切られた空間の可視属性を hidden に設定します。	オフ
-fvisibility-internal=*file*	*file* 引数でリストされたシンボルで区切られた空間の可視属性を internal に設定します。	オフ
-fwritable-strings	文字列のリテラルが書き込み可能なデータ・セクションに配置します。	オフ
-gcc-name=*name*	コンパイラが gcc C++ ライブラリを検索できない場合に g++ の場所を指定します。-cxxlib-gcc とともに使用します。標準の gcc インストールを行わなかった場合、このオプションを使用する必要があります。	オフ
-gcc-version=*nnn*	このオプションは、コンパイラの動作を gcc バージョン *nnn* の gcc と互換にします。このバージョンのインテル® C++ コンパイラは -gcc-version=320 (デフォルト) をサポートします。	オン
-isystem*dir*	システムのインクルード・パスのはじめにディレクトリ (*dir*) を追加します。	オフ
-no-gcc	__GNUC__、__GNUC_MINOR__ 、__GNUC_PATCHLEVEL__ マクロを事前に定義しません。	オフ
-nostdinc	-X と同じです。	オフ
-pch	プリコンパイル済みヘッダを自動的に処理します。	オフ
-pch_dir *dirname*	コンパイラに *dirname* でプリコンパイル済みヘッダ用ファイルを検索/作成するように指示します。	オフ
-prefetch[-]	コンパイラによるソフトウェア・プリフェッチの挿入を有効にします [無効にします]。デフォルトは -prefetch です。	オン
-prof_format_32	デフォルトでは、インテル C++ コンパイラは 64 ビットのプロファイリング・カウンタ (.dyn および .dpi) を作成します。このオプションは、インテル C++ コンパイラ 7.0 と互換性	オフ

	のある 32 ビットのカウンタを作成します。	
-shared-libcxa	インテル libcxa C++ ライブラリを動的にリンクします。	オン
-static-libcxa	インテル libcxa C++ ライブラリを静的にリンクします。	オフ
-strict_ansi	ANSI 規格に厳密に準拠しているダイアレクトを選択します。	オフ
-T *file*	リンカにファイル (file) からリンクコマンドを読むように指示します。	オフ
-use_pch *filename*	プリコンパイル済みヘッダ (*filename*.pchi) を使用します。	オフ
-Wbrief	診断情報を簡易モードで出力します。簡易モードでは、オリジナルのソース行は表示されず、エラー・メッセージは 1 行目のみ表示されます。	オフ
-Wcheck	移植不能なコード、意図しないコード・シーケンスになる可能性のあるコード、ANSI C 標準のわずかな変更によりプログラムの動作に影響するコードを、コンパイル時にチェックします。	オフ
-Wp64 Itanium ベース・システムのみ	64 ビット・ポーティングの診断結果を出力します。	
-xB IA-32 のみ	インテル Pentium M プロセッサおよび互換性のあるインテル・プロセッサ専用のコードを生成します	オフ
-xN IA-32 のみ	インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ専用のコードを生成します	オフ
-xP IA-32 のみ	HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) 専用のコードを生成します	オフ

オプションのクイック・リファレンス・ガイド

ここでは、すべてのコンパイラ・オプションと、いくつかのリンカを制御するオプションに関するリファレンスを紹介します。

- IA-32 アーキテクチャ固有のオプション
- Itanium® アーキテクチャ固有のオプション
- IA-32、Itanium ベース・システムの両方で使用できるオプション

オプション	説明	デフォルト
-A-	すべての事前定義マクロを無効にします。	オフ
-[no]align IA-32 のみ	変数と配列のメモリ・レイアウトを分析し、構成を変更します。	オフ
-[no]restrict	restrict 指示子とともに指定すると、ポインタの一義化が有効 [無効] になります。	オフ
-A*name*[(*value*)]	*name* というシンボルを、指定した一連の *value* に関連付けます。#assert 前処理ディレクティブと同じ働きをします。	オフ
-alias_args[-]	引数がエイリアス化された [エイリアス化されない] 可能性を示します。	-alias_args
-ansi	GNU* ANSI と同じ働きをします。	オフ
-ansi_alias[-]	-ansi_alias によって、コンパイラは次の点を前提にします: • 配列が外部からアクセスされません。 • ポインタを非ポインタ型にキャストしません。また、その逆もしません。 • 2 つの異なるスカラ型のオブジェクトへの参照はエイリアス不可です。例えば、int 型のオブジェクトは float 型のオブジェクトと、または float 型のオブジェクトは double 型のオブジェクトとエイリアスの設定はできません。 プログラムが上記の条件を満たす場合、-ansi_alias フラグを設定することでプログラムの最適化が向上します。ただし、プログラムが上記の条件を 1 つでも満たさない場合、コンパイラは -ansi_alias フラグを設定することで誤ったコードを生成する可能性があります。	オフ
-auto_ilp32 Itanium ベース・システムのみ	32 ビットアドレス空間を超えることができない (32 ビットポインタを使用する) アプリケーションを指定します。このオプションを使用するには、-ipo	オフ

	も指定してください。32 ビットアドレス空間 (2^{32}) を超えることができるプログラムで `-auto_ilp32` オプションを使用すると、プログラム実行中に予期できない問題が発生することがあります。	
`-ax{K|W|N|B|P}` IA-32 のみ	指定したプロセッサ向けに専用のコード (`K`、`W`、`N`、`B`、`P`) を生成し、かつ汎用性のある IA-32 コードも生成します。 • `K` = インテル® Pentium® III プロセッサおよび互換性のあるインテル® プロセッサ • `W` = インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ • `N` = インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ • `B` = インテル Pentium M プロセッサおよび互換性のあるインテル・プロセッサ • `P` = HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) および互換性のあるインテル・プロセッサ	オフ
`-C`	前処理済みのソース出力の中にコメントを書き込みます。	オフ
`-c`	オブジェクト・ファイルが生成された後、コンパイルの処理を止めます。C/C++ のソースファイルまたは前処理済みのソースファイルのそれぞれについてコンパイラがオブジェクト・ファイルを生成します。同様に、アセンブリ・ファイルについてもアセンブラがオブジェクト・ファイルを生成します。	オフ
`-c99[-]`	C プログラムの C99 サポートを有効にします [無効にします]。	オン
`-complex_limited_range[-]`	複素数のデータタイプに関する一部の算術演算で "基本代数展開の削除" を有効にします。これは複素数算術を使用するプログラムでパ	オフ

	フォーマンスを向上しますが、指数範囲の値を正しく計算しません。デフォルトは -complex_limited_range- です。	
-create_pch *filename*	プリコンパイル済みヘッダ (*filename*.pchi) を作成します。	オフ
-cxxlib-gcc	gcc の C++ ランタイム・ライブラリを使用してリンクします (gcc 3.2 以降が必要です)	オフ
-cxxlib-icc	インテルの C++ ランタイム・ライブラリを使用してリンクします。	オン
-dM	前処理を行った後に有効なマクロ定義を出力します (-E とともに使用します)。	オフ
-D*name*[=*value*]	マクロ名 (*name*) を定義し、そのマクロ名と指定された値 (*value*) を関連付けます。#define 前処理ディレクティブと同じ働きをします。	オフ
-dryrun	ドライバ・ツール・コマンドを表示し、ツールを実行しません。	オフ
-dynamic-linker*filename*	デフォルトのリンカではなく動的リンカ (*filename*) を選択します。	オフ
-E	C/C++ のソースファイルの前処理の後、コンパイルの処理を止め、その結果を stdout に書き込みます。	オフ
-EP	#line ディレクティブを省いて、前処理の結果を stdout に出力します。	オフ
-f[no]verbose-asm	コンパイラのコメント付きアセンブリ・ファイルを生成します。	オン
-falias	プログラムでエイリアシングを前提に処理します。	オン
-fast	プログラム全体を最高速で実行します。(-O3、-ipo、-static をオンにします)。	オフ
-fcode-asm	オプションのコードのコメント付きアセンブリ・ファイルを生成します。-S とともに使用します。	オフ
-ffnalias	関数内でのエイリアシングを前提に処理します。	オン

-fminshared	主実行ファイル用にコンパイルを行います。絶対アドレス指定を使用することができ、保護されているシンボルには位置独立でないコードが生成されます。	オフ
-fno-alias	プログラムでエイリアシングしないことを前提に処理します。	オフ
-fno-common	コンパイラが共通変数を定義されているものとして処理し、共通データ変数の gprel アドレス指定を使用できるようにします。	オフ
-fno-fnalias	関数内でエイリアシングしないことを前提に処理しますが、複数の呼び出しにわたる場合はエイリアシングを前提に処理します。	オフ
-fno-rtti	RTTI サポートを無効にします。	オフ
-fnsplit[-] Itanium ベース・システムのみ	関数分割を有効にします [無効にします]。-prof_use とともに使用する場合にはデフォルトはオンです。-prof_use を使用する場合に関数分割を無効にするには、-fnsplit- も指定してください。	オフ
-fp IA-32 のみ	EBP レジスタを汎用レジスタとして使用できないようにします。	オフ
-fpic, -fPIC	IA-32 の場合、このオプションは位置独立コードを生成します。 Itanium ベース・システムの場合、このオプションは完全なシンボル・プリエンプションを許可するコードを生成します。	オフ
-fp_port IA-32 のみ	代入と型キャスト時に浮動小数点の結果を丸めます。速度に多少影響します。	オフ
-fpstkchk IA-32 のみ	FP スタックが予測された状態であることを保証するためにすべての関数呼び出しの後に補足コードを生成します。	オフ
-fr32 Itanium ベース・システムのみ	浮動小数点レジスタのうち下位 32 個だけを使用します。	オフ
-fshort-enums	enumerated 型で必要なサイズのバイトを割り当てます。	オフ
-fsource-asm	オプションのソースのコメント付きア	オフ

	センブリ・ファイルを生成します。-S とともに使用します。	
-fsyntax-only	-syntax と同じです。	オフ
-ftz[-] Itanium ベース・システムのみ	デノーマル結果をゼロにフラッシュします。-O3 とともに使用する場合にはオプションをオンにします。	オフ
-funsigned-bitfields	デフォルトの bitfield 型を unsigned に変更します。	オフ
-funsigned-char	デフォルトの char 型を unsigned に変更します。	オフ
-fvisibility-default=*file*	*file* 引数でリストされたシンボルで区切られた空間の可視属性を default に設定します。	オフ
-fvisibility-extern=*file*	*file* 引数でリストされたシンボルで区切られた空間の可視属性を extern に設定します。	オフ
-fvisibility-hidden=*file*	*file* 引数でリストされたシンボルで区切られた空間の可視属性を hidden に設定します。	オフ
-fvisibility-internal=*file*	*file* 引数でリストされたシンボルで区切られた空間の可視属性を internal に設定します。	オフ
-fvisibility-protected=*file*	*file* 引数でリストされたシンボルで区切られた空間の可視属性を protected に設定します。	オフ
-fvisibility= [extern|default|protected |hidden|internal]	グローバル・シンボル (共通データ、定義データ、関数) の可視属性はデフォルトで設定されます。シンボルの可視属性はソースコードで明示的に設定されます。また、シンボルの可視属性の file オプションは -fvisibility 設定よりも優先されます。	オフ
-fwritable-strings	文字列のリテラルが書き込み可能なデータ・セクションに配置します。	オフ
-g	オブジェクト・コードの中にシンボリック・デバッグ情報を生成します。ソースレベルでのデバッガがこの情報を使用します。-g オプションはデフォルトの最適化を -O2 から -O0 に変更します。	オフ
-gcc-name=*name*	このオプションは、コンパイラが gcc	オフ

	C++ ライブラリを検索できない場合に、-cxxlib-gcc とともに使用して、g++ の場所を指定します。標準の gcc インストールを行わなかった場合、このオプションを使用する必要があります。	
-gcc-version=*nnn*	このオプションは、コンパイラの動作を gcc バージョン *nnn* の gcc と互換にします。このバージョンのインテル® C++ コンパイラは -gcc-version=320 (デフォルト) をサポートしています。	オン
-H	インクルード・ファイルの順番を出力して、コンパイルを続行します。	オフ
-help	各種コンパイラ・オプションの一覧を出力します。	オフ
-idirafter*dir*	2 つ目のインクルード・ファイルの検索先 (-I の後) にディレクトリ (*dir*) を追加します。	オフ
-I*directory*	インクルード・ファイルの検索先に追加するディレクトリ (*directory*) を指定します。 詳細...	オフ
-i_dynamic	インテル提供のライブラリを動的にリンクします。	オフ
-inline_debug_info	コール・サイトのソース・ポジションをインライン化コードに割り当てる代わりに、インライン化コードのソース・ポジションを保存します。	オフ
-ip	ファイルを 1 個コンパイルするときにプロシージャ間の最適化を有効にします。	オフ
-IPF_fma[-] Itanium ベース・システムのみ	浮動小数点乗算と加算/減算の組み合わせを有効 [無効] にします。	オフ
-IPF_fltacc[-] Itanium ベース・システムのみ	浮動小数点の精度に影響する最適化を有効 [無効] にします。	オフ
-IPF_flt_eval_method0 Itanium ベース・システムのみ	プログラムにより指定された精度で浮動小数点オペランドが評価されます。	オフ
-IPF_fp_speculation*mode* Itanium ベース・	次の *mode* 条件で浮動小数点のス	オフ

システムのみ	ペキュレーションを有効にします: • fast - 浮動小数点演算をスペキュレートします • safe - 安全な場合のみスペキュレートします • strict - off と同じです • off - 浮動小数点演算のスペキュレーションを無効にします	
-ip_no_inlining	-ip (プロシージャ間の最適化) によるインライン展開を無効にしますが、他のプロシージャ間の最適化には影響しません。	オフ
-ip_no_pinlining IA-32 のみ	部分的なインライン展開を無効にします。-ip または -ipo が必要です。	オフ
-ipo	複数ファイルにわたるプロシージャ間の最適化を有効にします。	オフ
-ipo_c	後のリンク段階で使用できるマルチファイル・オブジェクト・ファイル (ipo_out.o) を生成します。	オフ
-ipo_obj	-ipo とともに指定すると、実際のオブジェクト・ファイルが強制的に生成されます。	オフ
-ipo_S	後のリンク段階で使用できるマルチファイル・オブジェクト・ファイル (ipo_out.s) を生成します。	オフ
-isystem*dir*	システムのインクルード・パスのはじめにディレクトリ (*dir*) を追加します。	オフ
-ivdep_parallel Itanium ベース・システムのみ	このオプションは、IVDEP ディレクティブが指定されたループにループ・キャリー・メモリ依存が確実にないことを示します。	オフ
-Kc++	ソースファイルも、認識できないファイルも、すべて C++ ソースファイルとしてコンパイルします。	オン
-Knopic, -KNOPIC Itanium ベース・システムのみ	古いオプションです。fpic を使用してください。	オン (Itanium ベース・システム) オフ (IA-32)

-KPIC, -Kpic	古いオプションです。fpic を使用してください。	オフ
-L*directory*	*directory* で指定したディレクトリを検索してライブラリを探すようリンカに命令します。	オフ
-long double IA-32 のみ	long double データ型のデフォルト・サイズを 64 ビットから 80 ビットに変更します。	オフ
-M	ソースファイルに含まれている #include 行を基にして、ソースファイルごとに makefile の依存行を生成します。	オフ
-march=*cpu* IA-32 のみ	指定された *cpu* 向けに専用のコードを生成します。指定できる *cpu* の値は次のとおりです: • pentiumpro - インテル Pentium Pro プロセッサ • pentiumii - インテル Pentium II プロセッサ。 • pentiumiii - インテル Pentium III プロセッサ。 • pentium4 - インテル Pentium 4 プロセッサ。	オフ
-mcpu=*cpu*	特定の *cpu* 向けに最適化します。IA-32 の場合、指定できる *cpu* の値は次のとおりです: • pentium - Pentium プロセッサ向けに最適化します。 • pentiumpro - Pentium Pro プロセッサ、Pentium II プロセッサ、Pentium III プロセッサ向けに最適化します。 • pentium4 - Pentium 4 プロセッサ向けに最適化します (デフォルト)。 Itanium ベース・システムの場合、指定できる *cpu* の値は次のとおりです: • itanium - Itanium プロ	オン pentium (IA-32) itanium2 (Itanium ベース・システム)

	セッサ向けに最適化します。 • `itanium2` - Itanium 2 プロセッサ向けに最適化します (デフォルト)。	
`-MD`	前処理およびコンパイルを行います。依存情報が含まれている出力ファイル (`.d` 拡張子) を生成します。	オフ
`-MF`*`file`*	makefile の依存情報を *`file`* に生成します。`-M` または `-MM` を指定する必要があります。	オフ
`-MG`	`-M` と類似していますが、見つからないヘッダファイルを、生成したファイルとして処理します。	オフ
`-MM`	`-M` と類似していますが、システム・ヘッダ・ファイルをインクルードしません。	オフ
`-MMD`	`-MD` と類似していますが、システム・ヘッダ・ファイルをインクルードしません。	オフ
`-mp`	浮動小数点演算について、できる限り ANSI C 標準と IEEE 754 標準に適合するようにします。	オフ
`-mp1`	浮動小数点の精度を上げます (速度に与える影響は `-mp` よりも低いです)。	オフ
`-mrelax` Itanium ベース・システムのみ	リンカに `-relax` を渡します。	オン
`-mno-relax` Itanium ベース・システムのみ	リンカに `-relax` を渡しません。	オフ
`-mserialize-volatile` Itanium ベース・システムのみ	volatile データ・オブジェクトの参照に対して、制限されたメモリアクセスの順序を適用します。	オフ
`-mno-serialize-volatile` Itanium ベース・システムのみ	コンパイラは、volatile データ・オブジェクトの参照に対して、ランタイム時およびコンパイル時のメモリアクセスの順序を抑制する場合があります。特に、`.rel`/`.acq` コンプリータはロードおよびストアの参照で実行されません。	オフ

-MX	インテル wb ツールで使用される情報を含む依存ファイル (.o.dep 拡張子) を生成します。	オフ
-nobss_init	ゼロに初期化される変数を DATA セクションに格納します。ゼロに初期化される変数を BSS に配置することを禁止します (DATA を使用)。	オフ
-no_cpprt	C++ ランタイム・ライブラリでリンクしません。	オフ
-nodefaultlibs	リンク時に標準ライブラリを使用しません。	
-no-gcc	__GNUC__、__GNUC_MINOR__ 、__GNUC_PATCHLEVEL__ マクロを事前に定義しません。	オフ
-nolib_inline	標準ライブラリ関数のインライン展開を禁止します。 詳細...	オフ
-nostartfiles	リンク時に標準起動ファイルを使用しません。	オフ
-nostdinc	-X と同じです。	オフ
-nostdlib	リンク時に標準ライブラリと起動ファイルを使用しません。	オフ
-O	IA-32 の -O1 と同じです。Itanium ベース・システムの -O2 と同じです。	オフ
-O0	最適化を無効にします。	オフ
-O1	最適化を有効にします。速度について最適化します。Itanium コンパイラでは、-O1 はソフトウェアのパイプライン化をオフにし、コードのサイズを減らします。	オン
-O2	IA-32 の -O1 と同じです。Itanium ベース・システムの -O と同じです。	オフ
-O3	-O2 オプションに加えて、さらに強力な高い最適化を実行しますが、コンパイル時間が長くなる場合があります。パフォーマンスに及ぼす影響はアプリケーションに依存し、パフォーマンスが向上しないアプリケーションもあります。	オフ
-Ob*n*	コンパイラによるインライン展開を制	オン

	御します。どの程度までインライン展開が行われるかは、次のように *n* の値によって異なります: • 0: インライン化を無効にします。 • 1: __inline キーワードで宣言された関数のインライン化を有効にします (デフォルト)。C++ 言語に従ったインライン化も有効にします。 • 2: 任意の関数のインライン化を有効にします。ただし、インライン化する関数はコンパイラが判断します。プロシージャ間の最適化が有効になり、-ip と同じ効果が得られます。	
-o*file*	出力ファイル (*file*) に名前を付けます。	オフ
-openmp	OpenMP* ディレクティブに基づいてマルチスレッド・コードを生成する処理をパラレライザに許可します。-openmp オプションは、最適化レベル -O2 (デフォルト) またはそれ以上で有効です。	オフ
-openmp_report{0|1|2}	OpenMP パラレライザの診断レベルを制御します。	オン -openmp_report1
-openmp_stubs	シーケンシャル・モードによる OpenMP プログラムのコンパイルを有効にします。OpenMp ディレクティブは無視され、スタブ OpenMP ライブラリがシーケンシャルにリンクされます。	オフ
-opt_report	-opt_report_file が指定されている場合を除き、最適化レポートを作成し、stderr に送ります。	オフ
-opt_report_file*filename*	最適化レポートを保持するファイル名 (*filename*) を指定します。このオプションが指定されれば、-opt_report を実行する必要はありません。	オフ
-opt_report_level*level*	出力の冗長レベル (*level*) を指定します。有効な *level* 引数は次の	オフ

	とおりです: • min • med • max *level* が指定されていない場合、min がデフォルトで使用されます。	
-opt_report_phase*name*	レポートが生成されるコンパイル・フェーズ名 (*name*) を指定します。複数のフェーズから出力を得るために同じコンパイルで複数回、オプションを使用できます。 有効な引数 (*name*) は次のとおりです: • ipo: Interprocedural Optimizer (プロシージャ間の最適化) • hlo: High Level Optimizer (高レベル最適化) • ilo: Intermediate Language Scalar Optimizer (中間言語スカラ最適化機構) • ecg: Code Generator (コード・ジェネレータ) • omp: OpenMP* • all: すべてのフェーズ	オフ
-opt_report_routine*substring*	ルーチン部分文字列 (*substring*) を指定します。名前の一部に部分文字列 (*substring*) を含むすべてのルーチンからレポートを生成します。デフォルトでは、すべてのルーチンのレポートが生成されます。	オフ
-opt_report_help	-opt_report-phase の利用可能なすべての設定を表示します。コンパイルは実行されません。	オフ
-p	-qp と同じです。	オフ
-P, -F	C/C++ のソースファイルの前処理が済んだら、コンパイルの処理を止め、その結果をファイルに書き込みます。このファイル名は、コンパイラのデフォルトのファイル命名規則に従って付けられます。	オフ

-parallel	安全に並列実行可能な並列ループを検出し、そのループに対するマルチスレッド・コードを自動的に生成します。	オフ
-par_report{0|1|2|3}	自動パラレライザの診断レベルを次のように制御します: • -par_report0: 診断情報を表示しません。 • -par_report1: 正常に並列化されたループを示します (デフォルト)。 • -par_report2: 1 に加えて正常に並列化されなかったループを示します。 • -par_report3: 2 に加えて並列化の妨げとなると判断された、または想定される依存関係についての情報を示します。	オフ
-par_threshold[n]	ループの並列化による効果が現れる確率に基づいてループの自動並列化のしきい値を設定します (n=0 から 100)。このオプションは、コンパイル時に計算量が確定できないループに使用します。 • -par_threshold0: 計算量にかかわらず並列化を行います。 • -par_threshold100: 並列実行が有効であることが確実な場合にのみ並列化されます。	オフ
-pc32 IA-32 のみ	内部 FPU 精度を 24 ビットの仮数に設定します。	オフ
-pc64 IA-32 のみ	内部 FPU 精度を 53 ビットの仮数に設定します。	オフ
-pc80 IA-32 のみ	内部 FPU 精度を 64 ビットの仮数に設定します。	オン
-pch	プリコンパイル済みヘッダを自動的に処理します。	オフ

-pch_dir *dirname*	コンパイラにプリコンパイル済みヘッダ用ファイルを *dirname* で指定したディレクトリ内で検索するか、または *dirname* で指定したディレクトリ内に作成するように指示します。	オフ
-prec_div IA-32 のみ	浮動小数点除算から乗算へ変換する最適化処理を無効にします。浮動小数点除算の精度を上げます。	オフ
-prefetch[-]	コンパイラによるソフトウェア・プリフェッチの挿入を有効にします [無効にします]。デフォルトは -prefetch です。	オン
-prof_dir *dirname*	プロファイル情報 (*.dyn、*.dpi) を格納するディレクトリ (*dirname*) を指定します。	オフ
-prof_file *filename*	プロファイル・サマリ・ファイルのファイル名 (*filename*) を指定します。	オフ
-prof_format_32	デフォルトでは、インテル C++ コンパイラは 64 ビットのプロファイリング・カウンタ (.dyn および .dpi) を作成します。このオプションは、インテル C++ コンパイラ 7.0 と互換性のある 32 ビットのカウンタを作成します。	オフ
-prof_gen[x]	プログラムをインストルメントしてインストルメント実行に備え、さらに静的プロファイル情報ファイル (.spi) も新たに生成します。x 修飾子を付けると、コード・カバレッジ・ツールで利用可能な補足情報が収集されます。	オフ
-prof_use	動的フィードバック情報を使用します。	オフ
-Qinstall *dir*	コンパイラのインストール先のルートとしてディレクトリ (*dir*) を設定します。	オフ
-Qlocation,*tool*,*path*	tool で指定したツールの場所としてパス (*path*) を設定します。	オフ
-Qoption,*tool*,*list*	コンパイルの一連の処理の中で、引数 *list* を、アセンブラやリンカなど別のツール (*tool*) に渡します。	オフ
-qp	UNIX* prof tool を使って関数のプロファイリングができるようにコンパイルとリンクを行います。	オフ

-rcd IA-32 のみ	FPU の丸め制御の変更を無効にします。浮動小数点から整数への高速変換を行います。	オフ
-S	拡張子 .s のアセンブリ・ファイルを生成し、コンパイルを停止します。	オフ
-shared	共用オブジェクトを生成します。	オフ
-shared-libcxa	インテル libcxa C++ ライブラリを動的にリンクします。	オン
-sox[-] IA-32 のみ	コンパイラのオプションとバージョン情報を実行ファイルに保存します [保存しません]。	-sox-
-static	共有ライブラリとリンクしないようにします。	オフ
-static-libcxa	インテル libcxa C++ ライブラリを静的にリンクします。	オフ
-std=c99	C プログラムの C99 サポートを有効にします。	オン
-strict_ansi	ANSI 規格に厳密に準拠しているダイアレクトを選択します。	オフ
-syntax	C/C++ のソースファイルおよび前処理済みのソースファイルの構文解析が済んだらコンパイラの処理を止めます。つまり、プログラムの構文のチェックだけを行います。コードも出力ファイルも生成しません。警告とメッセージは stderr に出力されます。	オフ
-T *file*	リンカにファイル (*file*) からリンクコマンドを読むように指示します。	オフ
-tpp1 Itanium ベース・システムのみ	Itanium プロセッサ向けに最適化します。	オフ
-tpp2 Itanium ベース・システムのみ	Itanium 2 プロセッサ向けに最適化します。生成されたコードは、Itanium プロセッサと互換性があります。	オン
-tpp5 IA-32 のみ	Pentium プロセッサ向けに最適化します。	オフ
-tpp6 IA-32 のみ	インテル Pentium Pro プロセッサ、Pentium II プロセッサ、および Pentium III プロセッサ向けに最適化します。	オフ

-tpp7 IA-32 のみ	インテル Pentium 4 プロセッサ、インテル® Xeon™ プロセッサ、インテル Pentium M プロセッサ、HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) 向けに最適化します。	オン
-U*name*	マクロ名 (*name*) を定義できないようにします。#undef 前処理ディレクティブと同じ働きをします。	オフ
-unroll0	ループのアンロールを無効にします。	オフ
-unroll 0	ループのアンロールを無効にします。	オフ
-use_asm	アセンブラからオブジェクト・ファイルを生成します。	オフ
-use msasm IA-32 のみ	GNU スタイルではなく、Microsoft* MASM スタイルのインライン化アセンブリ・フォーマットを使用します。	オフ
-use_pch *filename*	プリコンパイル済みヘッダ (*filename*.pchi) を使用します。	オフ
-u *symbol*	*symbol* が未定義のようにみせます。	オフ
-V	コンパイラのバージョン情報を表示します。	オフ
-v	ドライバ・ツール・コマンドを表示し、ツールを実行します。	
-vec report[n] IA-32 のみ	ベクトライザの診断情報を次のように制御します。 • n = 0 診断情報を表示しません • n = 1 ベクトル化ループを示します (デフォルト) • n = 2 ベクトル化および非ベクトル化ループを示します • n = 3 ベクトル化および非ベクトル化ループを示し、データの依存関係の情報を抑止します • n = 4 非ベクトル化ループを示します。 • n = 5 非ベクトル化ループを示し、データの依存関係の	オン -vec_report1

	情報を抑止します	
-w	警告メッセージをまったく表示しません。	オフ
-Wall	警告メッセージをすべて表示します。	オフ
-Wbrief	診断情報を簡易モードで出力します。簡易モードでは、オリジナルのソース行は表示されず、エラー・メッセージは 1 行目のみ表示されます。	オフ
-Wcheck	移植不能なコード、意図しないコード・シーケンスになる可能性のあるコード、ANSI C 標準のわずかな変更によりプログラムの動作に影響するコードを、コンパイル時にチェックします。	オフ
-w*n*	診断機能の動作を調整します。 • n = 0 エラーを表示する (-w と同じ) • n = 1 警告とエラーを表示する(デフォルト) • n = 2 リマーク、警告、エラーを表示する	オン -w1
-wd*L1*[,*L2*,...]	*L1* から *LN* までの診断を禁止します。	オフ
-we*L1*[,*L2*,...]	*L1* から *LN* までの診断結果の重要度をエラーに変更します。	オフ
-Werror	警告をエラーと見なします。	オフ
-wn*n*	何個エラーが出力されたらコンパイルを止めるかを指定します。個数は *n* で指定します。	オン -wn100
-wr*L1*[,*L2*,...]	*L1* から *LN* までの診断結果の重要度をリマークに変更します。	オフ
-ww*L1*[,*L2*,...]	*L1* から *LN* までの診断結果の重要度を警告に変更します。	オフ
-Wl,*o1*[,*o2*,...]	リンカに *o1*、*o2*、その他のオプションを渡します。	オフ
-Wp64 Itanium ベース・	64 ビット・ポーティングの診断結果を出力します。	オフ

システムのみ		
-x*type*	-x*type* に続くすべてのソースファイルは次のいずれかであると認識されます。 • c - C ソースファイル • c++ - C++ ソースファイル • c-header - C ヘッダファイル • cpp-output - C 前処理済みファイル • assembler - アセンブリ・ファイル • assembler-with-cpp - 前処理の必要なアセンブリ・ファイル。 • none - 認識を無効にし、ファイル拡張子へ戻します。	オフ
-X	インクルード・ファイルの検索先ディレクトリ・リストから標準ディレクトリを削除します。	オフ
-x{K|W|N|B|P} IA-32 のみ	指定したプロセッサ向けに専用のコード (K、W、N、B、P) を生成します。 • K = インテル Pentium III プロセッサおよび互換性のあるインテル・プロセッサ • W = インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ • N = インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ • B = インテル Pentium M プロセッサおよび互換性のあるインテル・プロセッサ • P = HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) および互換性のあるインテル・プロセッサ	オフ
-Xlinker *val*	リンカに直接 *val* を渡します。	オフ
-Zp{1|2|4|8|16}	構造体を 1、2、4、8、16 バイト境界上にパックします。	オン -Zp16

コンパイラ・オプションの対応表

Linux*	Windows*	説明	Linux のデフォルト
-A-	/QA-	すべての事前定義マクロを削除します。	オフ
-A*name*[(*val*)]	/QA*name*[(*val*)]	*val* という値を持つアサーション名を作成します。	オフ
-ansi	/Za	ANSI に準拠していることをコンパイラに通知するかしないかを指定します。	オン
-ax{K|W|N|B|P}	/Qax{K|W|N|B|P}	指定したプロセッサ向けに専用のコード (K、W、N、B、P) を生成し、かつ汎用性のある IA-32 コードも生成します。 • K = インテル® Pentium® III プロセッサおよび互換性のあるインテル® プロセッサ • W = インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ • N = インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ • B = インテル Pentium M プロセッサおよび互換性のあるインテル・プロセッサ • P = HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) および互換性のあるインテル・プロセッサ	オフ
-C	/C	コメントを削除しません。	オフ

-c	/c	オブジェクト・ファイル (.o) までコンパイルします。リンクを行いません。	オフ
-D*name*[=*value*]	/D*name*[=*value*]	マクロを定義します。	オフ
-E	/E	前処理を行い、その結果を標準出力に出力します。	オフ
-fp	/Oy-	すべての関数について EBP ベースのスタックフレームを使用します。	オフ
-g	/Zi	シンボリック・デバッグ情報をオブジェクト・ファイル内に生成します。-g オプションはデフォルトの最適化を -O2 から -O0 に変更します。	オフ
-H	/QH	インクルード・ファイルの順番を出力します。	オフ
-help	/help	ヘルプ・メッセージ一覧を出力します。	オフ
-I*directory*	/I*directory*	インクルード・ファイルの検索先にディレクトリを追加します。	オフ
-inline_debug_info	/Qinline_debug_info	コール・サイトのソース・ポジションをインライン化コードに割り当てる代わりに、インライン化コードのソース・ポジションを保存します。	オフ
-ip	/Qip	シングル・ファイル IPO を有効にします (複数ファイル内)。	オフ
-ip_no_inlining	/Qip_no_inlining	IP の動作を最適化します。全体のインライン化も、部分的なインライン化も禁止されます (-ip、-ipo のいずれかが必要です)。	オフ
-ipo	/Qipo	マルチファイル IPO を有効にします (複数ファイル間)。	オフ
-ipo_obj	/Qipo_obj	IP の動作を最適化します。実際のオブジェクト・ファイルを強制的に生成します (-ipo が必要です)。	オフ
-KPIC	NA	位置に依存しないコードを生成します (-Kpic と同じです)。	オフ

-Kpic	NA	位置に依存しないコードを生成します (-KPIC と同じです)。	オフ
-long_double	/Qlong_double	80 ビット長の long double 型を有効にします。	オフ
-m	NA	マップファイルを生成するようにリンカに命令します。	オフ
-M	/QM	makefile の依存情報を生成します。	オフ
-mp	/Op[-]	浮動小数点の精度を維持します (一部の最適化項目が無効になります)。	オフ
-mp1	/Qprec	浮動小数点の精度を上げます (速度に与える影響は -mp よりも低いです)。	オフ
-nobss_init	/Qnobss_init	ゼロに初期化される変数を BSS に配置することを禁止します (DATA を使用)。	オフ
-nolib_inline	/Oi[-]	組込み関数のインライン展開を禁止します。	オフ
-O	/O2		オフ
-o*file*	/Fe*file* or /Fo*file*	出力ファイル (file) に名前を付けます。	オフ
-O0	/Od	最適化を禁止します。	オフ
-O1	/O1	速度について最適化します。	オフ
-O2	/O2		オン
-P	/EP	ファイルまで前処理を行います。	オフ
-pc32	/Qpc 32	内部 FPU 精度を 24 ビットの仮数に設定します。	オフ
-pc64	/Qpc 64	内部 FPU 精度を 53 ビットの仮数に設定します。	オフ
-pc80	/Qpc 80	内部 FPU 精度を 64 ビットの仮数に設定します。	オン
-prec_div	/Qprec_div	浮動小数点除算の精度を上げます (速度に多少影響します)。	オフ
-prof_dir*directory*	/Qprof_dir*directory*	出力ファイル (*.dyn、*.dpi) のプロファイリングに使うディ	オフ

		レクトリを指定します。	
-prof_file*filename*	/Qprof_file*filename*	サマリファイルのプロファイリングに使うファイル名を指定します。	オフ
-prof_gen[x]	/Qprof_genx	プロファイリングができるようにプログラムをインストルメントします。x 修飾子を付けると、補足情報が収集されます。	オフ
-prof_use	/Qprof_use	最適化中にプロファイリング情報が使えるようにします。	オフ
-Qinstall *dir*	NA	コンパイラのインストール先のルートとして *dir* を設定します。	オフ
-Qlocation,*str*,*dir*	/Qlocation,*tool*,*path*	*str* で指定したツールの場所として *dir* を設定します。	オフ
-Qoption,*str*,*opts*	/Qoption,*tool*,*list*	str で指定したツールに opts オプションを渡します。	オフ
-qp, -p	NA	UNIX* prof tool を使って関数のプロファイリングができるようにコンパイルとリンクを行います。	オフ
-rcd	/Qrcd	浮動小数点から整数への高速変換機能を有効にします。	オフ
-restrict	/Qrestrict	ポインタの一義化ができるように restrict キーワードを有効にします。	オフ
-S	/S	拡張子 .s のアセンブリ・ファイルを生成し、コンパイルを停止します。	オフ
-sox[-]	/Qsox	コンパイラのオプションとバージョン情報を実行ファイルに保存します [保存しません]。	-sox-
-syntax	/Zs	構文チェックのみを行います。	オフ
-tpp5	/G5	Pentium プロセッサに合わせて最適化します。	オフ
-tpp6	/G6	Pentium Pro プロセッサ、Pentium II プロセッサ、Pentium III プロセッサの各プロセッサに合わせて最適化します。	オフ

-tpp7	/G7	Pentium 4 プロセッサに合わせて最適化します。	オフ
-U*name*	/U*name*	事前定義済みマクロを削除します。	オフ
-unroll0	/Qunroll0	ループのアンロールを無効にします。	オフ
-V	/QV	コンパイラのバージョン情報を表示します。	オフ
-w	/w	エラーを表示します。	オフ
-w2	/W4	リマーク、警告、エラーの各メッセージを表示します。	
-Wbrief	/WL	冗長ではない診断結果を出力します。	オフ
-w*n*	/Wn	診断機能の動作を調整します。エラーを表示するときは、n=0 にします。警告とエラーを表示するときは、n=1 にします。リマーク、警告、エラーを表示するときは、n=2 にします。	オフ
-wd*L1*[,*L2*,...]	/Qwd[tag]	L1 から LN まで、診断機能を無効にします。	オフ
-we*L1*[,*L2*,...]	/Qwe[tag]	L1 から LN までの診断結果の重要度をエラーに変更します。	オフ
-wn*n*	/Qwn[tag]	最大 *n* エラーを出力します。	オフ
-Wp64	/Wp64	64 ビット・ポーティングの診断結果を出力します。	オフ
-wr*L1*[,*L2*,...]	/Qwr[tag]	L1 から LN までの診断結果の重要度をリマークに変更します。	オフ
-ww*L1*[,*L2*,...]	/Qww[tag]	L1 から LN までの診断結果の重要度を警告に変更します。	オフ
-X	/X	インクルード・ファイルの検索先から、標準のディレクトリを外します。	オフ
-x{K|W|N|B|P}	/Qx{K|W|N|B|P}	指定したプロセッサ向けに専用のコード (K、W、N、B、P) を生成します。	オフ

		• K = インテル Pentium III プロセッサおよび互換性のあるインテル・プロセッサ • W = インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ • N = インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ • B = インテル Pentium M プロセッサおよび互換性のあるインテル・プロセッサ • P = HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) および互換性のあるインテル・プロセッサ	
-Zp{1|2|4|8|16}	/Zp[*n*]	構造体を 1、2、4、8、16 バイト境界上にパックします。	オフ

デフォルトのコンパイラ・オプション

- IA-32 アーキテクチャ固有のオプション
- Itanium® アーキテクチャ固有のオプション
- IA-32、Itanium ベース・システムの両方で使用できるオプション

オプション	説明
-c99	C プログラムの C99 サポートを有効にします。
-falias	プログラムでエイリアシングを前提に処理します。
-ffnalias	関数内でのエイリアシングを前提に処理します。
-gcc-version=320	このオプションは、コンパイラの動作を gcc バージョン *nnn* の gcc と互換にします。このバージョンのインテル® C++ コンパイラは -gcc-version=320 (デフォルト) をサポートします。
-mcpu=pentium4	Pentium® 4 プロセッサ向けに最適化します (IA-32 システムのみ)。

-mcpu=itanium2	Itanium 2 プロセッサ向けに最適化します (Itanium ベース・システムのみ)。
-O2	IA-32 の -O1 と同じです。Itanium ベース・システムの -O と同じです。
-Ob1	__inline キーワードで宣言された関数のインライン化を有効にします。C++ 言語に従ったインライン化も有効にします。
-pc80 IA-32 のみ	内部 FPU 精度を 64 ビットの仮数に設定します。
-prefetch	コンパイラによるソフトウェア・プリフェッチの挿入を有効にします。
-sox- IA-32 のみ	コンパイラのオプションとバージョン情報を実行ファイルに保存しません。
-std=c99	C プログラムの C99 サポートを有効にします。
-tpp2 Itanium ベース・システムのみ	Itanium 2 プロセッサ向けに最適化します。生成されたコードは、Itanium プロセッサと互換性があります。
-tpp7 IA-32 のみ	インテル Pentium 4 プロセッサ、インテル® Xeon™ プロセッサ、インテル Pentium M プロセッサ、HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) 向けに最適化します。
-w1	診断機能の動作を調整します。警告とエラーを表示します。
-Zp16	構造体を 16 バイト境界上にパックします。

アプリケーションのビルドとデバッグ

コンパイラの起動

コンパイラのデフォルトの動作

オプションを何も指定せずにインテル® C++ コンパイラを起動すると、次のデフォルト設定が使用されます:

- ファイル名 `a.o` の実行可能出力ファイルを生成します。
- 設定ファイルで指定されているオプションを先に起動します。「設定ファイル」を参照してください。
- 共用オブジェクトの場所は、`LD_LIBRARY_PATH` 環境変数で指定されます。
- 構造体のアライメントの最も厳密な制約条件を 8 バイトに設定します。
- エラー・メッセージと警告メッセージを表示します。
- デフォルトの `-O2` オプションを使用して標準的な最適化を実行します。「最適化レベルの設定」を参照してください。
- Unicode* (マルチバイト) 形式の文字をサポートするオペレーティング・システムでは、コンパイラはこれらの文字が含まれるファイル名を処理します。

コンパイラがコマンドライン・オプションを認識しない場合、そのオプションは無視され、警告が表示されます。システム・メッセージの詳細については、「診断メッセージ」を参照してください。

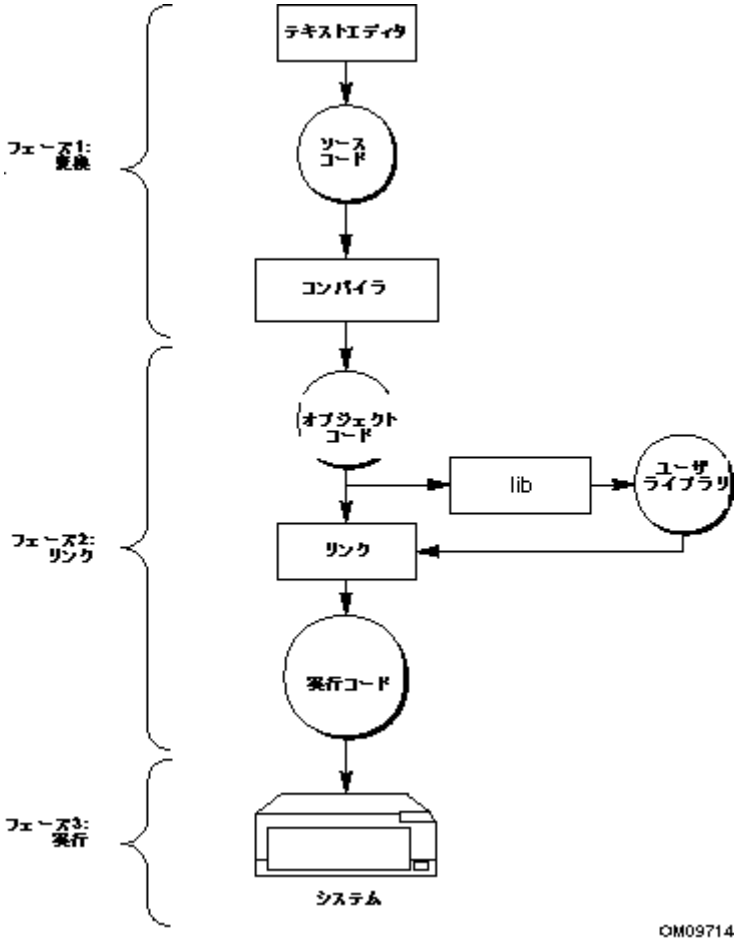
コンパイル・フェーズ

実行ファイルを生成する場合、コンパイラはコンパイル・フェーズとリンクフェーズを実行します。コンパイラを起動すると、ソースファイル名の拡張子とコマンドラインで指定したコンパイル・オプションを基にして、どのコンパイル・フェーズを実行するかをコンパイラ・ドライバが判断します。

コンパイラは、オブジェクト・ファイルと認識できないファイル名をリンカに渡します。次に、リンカは、そのファイルがオブジェクト・ファイル (`.o`) なのかライブラリ (`.a`) なのかを判断します。コンパイラ・ドライバは、どの種類の入力ファイルも正しく処理できるため、どのコンパイル・フェーズを起動するときにも使用できます。

コンパイラとシステム専用のプログラミング支援ツールの関係を下図に示します。

アプリケーションの開発サイクル



コマンドラインからのアプリケーションのビルド

コンパイラの起動

インテル® C++ コンパイラの起動方法は次のとおりです:

- 直接起動する: コマンドラインからのコンパイラの起動
- makefile を使用する: コマンドラインからの makefile の実行

コマンドラインからのコンパイラの起動

コマンド・ラインからインテル® C++ コンパイラを起動するには、次の 2 つの手順が必要です:

1. 環境変数を設定する
2. icc または icpc を使用してコンパイラを起動する

環境変数を設定する

コンパイラを実行する前に、環境変数を設定して、各種コンポーネントの場所を指定する必要があります。インテル C++ コンパイラには、環境変数の設定に使用することができるシェル・スクリプトが含まれています。デフォルトでコンパイラをインストールすると、これらのスクリプトは次の場所にあります:

- /opt/intel_cc_80/bin/iccvars.sh
- /opt/intel_cc_80/bin/iccvars.csh

環境スクリプトを実行するには、コマンドラインで次のいずれかを入力してください:

```
prompt>source /opt/intel_cc_80/bin/iccvars.sh
```

または

```
prompt>source /opt/intel_cc_80/bin/iccvars.csh
```

Linux* を起動したときにスクリプトを自動的に実行する場合は、スタートアップ・ファイルの最後にこのコマンドを追加します。

iccvars.sh 用のサンプル .bash_profile エントリ:

```
# set environment vars for Intel C++ compiler
source /opt/intel_cc_80/bin/iccvars.sh
```

icc または icpc を使用してコンパイラを起動する

icc または icpc のいずれかを使用してコマンドラインからインテル C++ コンパイラを起動することができます。呼び出しはそれぞれ、C++ ランタイム・ライブラリとヘッダファイルを含みます。C++ ランタイム・ライブラリとヘッダファイルが必要ない場合は、-no_cpprt オプションを使用してください。

コマンドライン構文

icc または icpc のいずれかを使用してインテル C++ コンパイラを起動する場合、次のコマンドを使用してください:

```
prompt>{icc|icpc} [options] file1 [file2 . . .] [linker
options]
```

引数	説明
options	1 つ以上のコマンドライン・オプションを示します。コンパイラは、ハイフン (-) が先頭にある 1 文字以上の文字をオプションとして認識します。「オプションのクイック・リファレンス・ガイド」を参照してください。
file1, file2 . . .	コンパイル・システムによって処理される 1 つ以上のファイルを示します。複数のファイルを指定することもできます。複数のファイルの区切りにはスペースを使用してください。
linker options	リンカに渡されるオプションです。

例:

```
prompt>icpc -prec_div -axW my_source1.cpp my_source2.cpp -Bstatic
```

コマンドラインからの makefile の実行

インテル® C++ コンパイラを使用してコマンドラインから make を実行するには、/usr/bin がパスに含まれていることを確認してください。C シェルを使用している場合は、.cshrc ファイルを編集して、次の行を追加できます:

```
setenv PATH /usr/bin:<インテル・コンパイラへのフルパス>
```

注

インテル・コンパイラを使用するには、makefile が設定 CC=icc をインクルードしなければなりません。インテル・コンパイラを使用するように makefile に指示するには、コマンドラインで同じ設定を使用します。makefile が gcc (GNU* C コンパイラ) 用に記述されている場合、インテル・コンパイラで認識されないコマンドライン・オプションを変更する必要があります。

オプションを変更したら、次のようにコンパイルすることができます:

```
prompt>make -f my_makefile
```

コンパイラ入力ファイル

インテル® C++ コンパイラは以下の表のファイル名拡張子を認識します。

ファイル名	ファイルの種類
filename.a	オブジェクト・ライブラリ

filename.i	icc を使用してコンパイラを起動した場合、.i ファイルは C 言語ソースファイルとして扱われます。icpc を使用してコンパイルを起動した場合、.i ファイルは C++ 言語ソースファイルとして扱われます。
filename.o	コンパイル済みのオブジェクト・モジュール
filename.s	アセンブリ・ファイル
filename.so	共用オブジェクト・ファイル
filename.S	プリプロセスが必要なアセンブリ・ファイル
filename.c	C 言語ソースファイル
filename.C filename.cc filename.CC filename.cpp filename.cxx	C++ 言語ソースファイル

コンパイル・オプション

概要: コンパイル

このセクションでは、コンパイル・プロセスとそれらによって作成される出力を指定するインテル® C++ コンパイラ・オプションについて説明します。デフォルトでは、ソースコードはコンパイラによって実行ファイルに直接変換されます。適切なオプションを指定して、コンパイラに次の出力ファイルを作成させることができます:

- -P オプションを使用した場合、前処理済みのファイル (.i) が作成されます。
- -S オプションを使用した場合、アセンブリ・ファイル (.asm) が作成されます。
- -c オプションを使用した場合、オブジェクト・ファイル (.obj) が作成されます。
- デフォルトでは、実行ファイル (.exe) が作成されます。

また、出力ファイルに名前を付けたり、リンカに渡すオプションのセットを指定できます。フェーズ限定オプションを指定すると、コンパイラは、各主要入力ファイルに対して、完了した最後のフェーズの出力を示す別の出力ファイルを作成します。

概要: プリプロセッサ・オプション

このセクションでは、プリプロセッサの動作の指定に使用できるオプションについて説明します。前処理は、マクロ置換、条件付きコンパイル、ファイルのインクルードといった処理を行います。

前処理オプション

オプション	説明
-A*name*[(*values*,...)]	*name* というシンボルを、指定した一連の *values* に関連付けます。#assert 前処理ディレクティブと同じ働きをします。
-A-	事前定義済みマクロとアサーションをすべて無効にします。
-C	前処理済みのソース出力の中にコメントを書き込みます。
-D*name*[(*value*)]	*name* という名前のマクロを定義し、そのマクロを *value* で指定した値に関連付けます。デフォルト (-D*name*) のままだと、*value* が 1 のマクロが定義されます。
-E	ソース・モジュールを展開してその結果を標準出力に出力するようにプリプロセッサに指示します。
-EP	ソース・モジュールを展開してその結果を標準出力に出力するようにプリプロセッサに指示します。#line ディレクティブが出力に含まれないこと以外は -E と同じです。
-P	ソース・モジュールを展開してその結果をカレント・ディレクトリの .i ファイルに保存するようにプリプロセッサに指示します。
-U*name*	指定したマクロ名 (*name*) に対する自動定義を行わない設定にします。

前処理のみを行う

ソースファイルをコンパイルしないで前処理のみを行うには、-E 、-P または -EP オプションを使用します。これらのオプションを使用すると、コンパイルの前処理のみが行われます。

-E オプション

-E オプションを指定すると、プリプロセッサはソース・モジュールを展開して結果を標準出力に出力します。前処理済みのソースには、コンパイラがソースファイルと行番号を特定する #line ディレクティブが含まれています。例えば、2 つのソースファイルを前処理して標準出力に出力するには、次のコマンドを実行します:

```
prompt>icpc -E prog1.cpp prog2.cpp
```

-P オプション

-P オプションを指定すると、プリプロセッサはソース・モジュールを展開して結果をカレント・ディレクトリの .i ファイルに保存します。-E オプションと異なり、-P からの出力に #line ディレクティブは含まれません。デフォルトでは、プリプロセッサはソースファイルと同じファイル名で拡張子が .i の出力ファイルを作成します。この出力ファ

イルの名前は、-o*file* オプションを使用して変更することができます。例えば、次のコマンドは、別のコンパイルの入力として使用できる、prog1.i と prog2.i という名前の 2 つのファイルを作成します:

```
prompt>icpc -P prog1.cpp prog2.cpp
```

警告

-P オプションを指定した場合、名前と拡張子が同じファイルは上書きされます。

-EP オプション

出力に #line ディレクティブを書き込まないようにプリプロセッサに指示するには、-EP オプションを使用します。-EP は -E -P と同じ働きをします。

```
prompt>icpc -EP prog1.cpp prog2.cpp
```

前処理済みのソース出力にコメントを保存する

前処理済みのソース出力にコメントを保存するには、-C オプションを使用します。しかし、前処理する命令の後のコメントは保存されません。

前処理ディレクティブと同じ働きをするオプション

前処理ディレクティブと同じ働きをするものとして、-A、-D、および -U オプションを使用することができます:

- -A は #assert 前処理ディレクティブと同じ働きをします
- -D は #define 前処理ディレクティブと同じ働きをします
- -U は #undef 前処理ディレクティブと同じ働きをします

-A オプション

アサーションを作成するには、-A オプションを使用します。**構文:** -A*name*[(*value*)]

引数	説明
name	アサーションの識別子 (名前) を指定します。
value	アサーションの値 (*value*) を指定します。*value* を指定する場合は、それを区切る括弧とともに引用符で囲む必要があります。

例えば、識別子が fruit で関連する値が orange と banana のアサーションを作成するには、次のコマンドを使用します:

```
prompt>icpc -A"fruit(orange,banana)" prog1.cpp
```

-D オプション

マクロを定義するには、-D オプションを使用します。**構文:** -D*name*[=*value*]

引数	説明
name	定義するマクロの名前です。
value	名前として入力する値です。value を入力しなかった場合、name は 1 に設定されます。英数字以外を含む値は引用符で囲んでください。

例えば、SIZE という名前で値が 100 のマクロを定義するには、次のコマンドを使用します:

```
prompt>icpc -DSIZE=100 prog1.cpp
```

-D オプションは関数の定義にも使用することができます。
例:

```
prompt>icpc -D"f(x)=x" prog1.cpp
```

-U オプション

事前定義済みマクロを削除するには、-U オプションを使用します。**構文:** -U*name*

引数	説明
name	定義を削除するマクロの名前です。

注

同じコンパイルで -D と -U オプションを使用すると、コンパイラはコマンドラインで指定された順にオプションを処理するのではなく、-U オプションの前に -D オプションを処理します。

事前定義済みマクロ

インテル® C++ コンパイラで使用可能な事前定義マクロを下の表に示します。**アーキテクチャ**の欄には、マクロをサポートするインテル® アーキテクチャが表示されます。

ISO/ANSI 標準で指定される事前定義マクロは表には含まれていません。すべての有効なマクロ定義の一覧を表示するには、`-E -dM` オプションを使用してください。例:

```
prompt>icpc -E -dM prog1.cpp
```

マクロ名	値	アーキテクチャ
__DATE__	現在の日付	両方
__ECC	1	Itanium® アーキテクチャのみ
__EDG__	1	両方
__EDG_VERSION__	302	両方
__ELF__	1	両方
__extension__	(値なし)	両方
__GNUC__	3	両方
__gnu_linux__	1	両方
__GNUC_MINOR__	2	両方
__GNUC_PATCHLEVEL__	0	両方
__gnu_linux__	1	両方
__GXX_ABI_VERSION	102	両方
__HONOR_STD	1	IA-32 のみ
__i386	1	IA-32 のみ
__i386__	1	IA-32 のみ
i386	1	IA-32 のみ
__ia64	1	Itanium アーキテクチャのみ
__ia64__	1	Itanium アーキテクチャのみ
ia64	1	Itanium アーキテクチャのみ
__ICC	800	IA-32 のみ
__INTEL_COMPILER	800	両方
_INTEGRAL_MAX_BITS	64	Itanium アーキテクチャのみ
__itanium__	1	Itanium アーキテクチャのみ
__linux	1	両方
__linux__	1	両方
linux	1	両方
__LONG_DOUBLE_SIZE__	80	IA-32 のみ

__LONG_MAX__	9223372036854775807L	Itanium アーキテクチャのみ
__lp64	1	Itanium アーキテクチャのみ
__LP64__	1	Itanium アーキテクチャのみ
_LP64	1	Itanium アーキテクチャのみ
__NO_INLINE__	1	両方
__NO_MATH_INLINES	1	両方
__NO_STRING_INLINES	1	両方
__OPTIMIZE__	1	両方
__PTRDIFF_TYPE__	int (IA-32) long (Itanium アーキテクチャ)	両方
__QMSPP_	1	IA-32 のみ
__REGISTER_PREFIX__	(値なし)	両方
__SIGNED_CHARS__	1	両方
__SIZE_TYPE__	unsigned (IA-32) unsigned long (Itanium アーキテクチャ)	両方
__STDC__	1	両方
__STDC_HOSTED__	1	両方
__TIME__	現在の時間	両方
__unix	1	両方
__unix__	1	両方
unix	1	両方
__USER_LABEL_PREFIX__	(値なし)	両方
__VERSION__		両方
__WCHAR_TYPE__	long int (IA-32) int (Itanium アーキテクチャ)	両方
__WINT_TYPE__	unsigned int	両方

マクロ定義の抑止

指定された *name* で現在有効になっているマクロ定義を抑止するには、-U*name* オプションを使用します。 -U オプションは、#undef プリプロセッサ・ディレクティブと同じ機能を果たします。

コンパイル環境

コンパイル環境のカスタマイズ

IA-32 および インテル® Itanium® アーキテクチャでは、コンパイル環境を設定する必要があります。コンパイル時の環境は、次の変数、オプション、ファイルを指定してカスタマイズすることができます:

- 環境変数 -- コンパイラと他のツールが特定のファイルを検索するパス
- 設定ファイル -- 各コンパイル時に使用するオプション
- 応答ファイル -- 各プロジェクトに使用するオプションとファイル
- インクルード・ファイル -- ソース・ヘッダ・ファイルの名前と場所

環境変数

コンパイラがライブラリやインクルード・ファイルなど特殊なファイルを検索する際のパス(検索先)を指定して、環境をカスタマイズできます。

- `LD_LIBRARY_PATH` – 共用オブジェクトの場所を指定します。
- `PATH` – システムがバイナリ実行ファイルを検索するディレクトリを指定します。
- `ICCCFG` – `icc` を使用してコンパイラを起動したときにコンパイルをカスタマイズするための設定ファイルを指定します。
- `ICPCCFG` – `icpc` を使用してコンパイラを起動したときにコンパイルをカスタマイズするための設定ファイルを指定します。
- いくつかの環境変数が一時ファイルの場所を指定するためにサポートされています。コンパイラは、`TMP`、`TMPDIR`、`TEMP` の順に変数を検索します。これらの変数が見つからない場合、一時ファイルは `/tmp` に格納されます。
- `IA32ROOT` (IA32 ベース・システム) – `bin`、`lib`、`include` および代用ヘッダのディレクトリを含むディレクトリを指します。
- `IA64ROOT` (Itanium® ベース・システム) – `bin`、`lib`、`include` および代用ヘッダのディレクトリを含むディレクトリを指します。

GNU* 環境変数

インテル® C++ コンパイラは、次の GNU 環境変数をサポートします:

- `CPATH` – C/C++ コンパイル用のインクルード・ディレクトリのパスを指定します。
- `C_INCLUDE_PATH` – C コンパイル用のインクルード・ディレクトリのパスを指定します。
- `CPLUS_INCLUDE_PATH` – C++ コンパイル用のインクルード・ディレクトリのパスを指定します。

- LIBRARY_PATH – LIBRARY_PATH の値は、PATH のようにコロンで区切られたディレクトリのリストです。
- DEPENDENCIES_OUTPUT – この変数が設定された場合、その値はコンパイラで処理されるシステム以外のヘッダファイルに基づいて Make の依存性を出力する方法を指定します。システム・ヘッダ・ファイルは依存性出力で無視されます。
- SUNPRO_DEPENDENCIES – この変数は、システム・ヘッダ・ファイルが無視されないこと以外は、DEPENDENCIES_OUTPUT と同じです。

コンパイル環境オプション

インテル C++ コンパイラには、環境変数の設定に使用することができるシェル・スクリプトが含まれています。詳細については、「コマンドラインからのコンパイラの起動」を参照してください。

設定ファイル

コマンドラインへ入力する項目が頻繁にある場合は、設定ファイルを使用してその入力作業を自動化できます。そうすれば、コマンドライン・オプションの入力時間が短くなるだけでなく、統一を図れるようになります。有効なコマンドライン・オプションであれば何でも設定ファイルに書き込めます。起動されたコンパイラは、設定ファイルに記述している各コマンドライン・オプションを上から順に処理します。

注

コンパイラを実行するたびに、設定ファイルに書かれたオプションが実行されるので注意してください。プロジェクトごとにオプションを変える必要がある場合は、「応答ファイル」を参照してください。

設定ファイルの使用方法

次に、基本的な設定ファイルの例を示します。.cfg ファイルの作成が済んだら、後はコンパイラを実行するときにその設定ファイルをコンパイラの実行ファイルと同じディレクトリに格納するだけです。シャープ (#) 文字の後ろのテキストはコメントとして認識されます。設定ファイルは icc.cfg です。

```
## Sample configuration file.
## Define preprocessor macro MY PROJECT.

-DMY_PROJECT

## Additional directories to be searched
## for INCLUDE files, before the default.

-I /project/include
```

ICCCFG を使用した場所の指定

ICCCFG 環境変数を使用して設定ファイルの場所を指定することができます:

```
ICCCFG=/cpp/config/my_options.cfg
```

icc を使用してコンパイラを起動するたび、my_options.cfg が設定ファイルとして使用されます。icpc を使用してコンパイラを起動する場合は、ICPCCFG 環境変数を使用します。

「環境変数」を参照してください。

応答ファイル

特定のコンパイル処理に使用するオプションを指定し、さらにその情報をそれぞれ別のファイルに保存するときは、応答ファイルを使用します。応答ファイルはコマンドラインのオプションとして呼び出されます。応答ファイルに含まれている各種オプションは、その応答ファイルを呼び出した場所のコマンドラインに挿入されます。

応答ファイルのサンプル

```
# response file: response1.txt
# compile with these options

-axW
-pch

# end of response1 file
```

```
# response file: response2.txt
# compile with these options

-mp1
-strict ansi

# end of response2 file
```

応答ファイルを使用すれば、コマンドラインでの入力作業が自動化されるため、コマンドライン・オプションの入力時間が短くなるだけでなく、統一が図れます。プロジェクトごとに各種オプションを保持する場合は、それぞれ別の応答ファイルを使用すると、プロジェクトが変わっても設定ファイルを編集する必要はありません。

応答ファイルの 1 行の中に任意の数のオプションやファイル名を書き込むことができます。同じコマンドラインの中で複数の応答ファイルを参照できます。

応答ファイルを使用する際は次の構文を使用します:

```
prompt>icpc @response1.txt source1.cpp @response2.txt
source2.cpp
```

注

コマンドライン上では、応答ファイル名の前にアットマーク (@) を付けることに注意してください。

インクルード・ファイル

インクルード・ディレクトリは、環境変数で指定されたデフォルトのパスと -I*directory* オプションで指定されたディレクトリから検索されます。検索先ディレクトリを複数指定するときは、-I*directory* コマンドを複数指定する必要があります。コンパイラはインクルード・ファイルのディレクトリを次の順に検索します:

- include を含むソースファイルのディレクトリ
- -I オプションで指定されたディレクトリ

インクルード・ディレクトリの削除方法

コンパイラが環境変数で指定されたデフォルトのパスを検索しないようにするには、-X オプションを使用します。-X オプションと -I オプションを併用すると、インクルード・ファイルのデフォルトのパスではなく別のパスを検索するようにコンパイラに命令できます。

例えば、デフォルトのパスの代わりにパス /alt/include を検索するようにコンパイラに命令するときは次のようにします:

```
prompt>icpc -X -I/alt/include prog.cpp
```

「インクルード・ファイルの検索」も参照してください。

インクルード・ファイルの検索

デフォルトでは、コンパイラは、CPATH、C_INCLUDE_PATH 、および CPLUS_INCLUDE_PATH 環境変数で指定されたディレクトリにある標準のインクルード・ファイルを検索します。インクルード・ファイルの格納場所は設定ファイルで指定することができます。

インクルード・ディレクトリの指定方法

インクルード・ファイルの検索先ディレクトリを追加指定するには、-I*directory* オプションを使用します。検索先ディレクトリを複数指定するときは、-I*directory* コ

マンドを複数指定する必要があります。インクルードしたファイルは、#include プリプロセッサ・ディレクティブでプログラムに取り込まれます。コンパイラはインクルード・ファイルのディレクトリを次の順に検索します:

- include を含むソースファイルのディレクトリ
- -I オプションで指定されたディレクトリ
- CPATH、C_INCLUDE_PATH、および CPLUS_INCLUDE_PATH 環境変数で指定されたディレクトリ

インクルード・ディレクトリの削除方法

コンパイラが環境変数で指定されたデフォルトのパスを検索しないようにするには、-X オプションを使用します。

-X オプションと -I オプションを併用すると、インクルード・ファイルのデフォルトのパスではなく別のパスを検索するようにコンパイラに命令できます。

例えば、デフォルトのパスの代わりにパス /alt/include を検索するようにコンパイラに命令するときは次のようにします:

```
prompt>icpc -X -I/alt/include source.cpp
```

コンパイルの制御

エラーなしで生成された出力ファイルは、その後コンパイラへの入力ファイルとして使用できます。以下の表は、出力を制御する各種オプションについて説明したものです:

オプション	入力	出力
-P	• ソースファイル	前処理済みのファイル (.i ファイル)。
-E	• ソースファイル	前処理済みのソースファイル。stdout に直接出力します。
-EP	• ソースファイル	前処理済みのソースファイル。stdout に直接出力し、行番号を省略します。
-c	• ソースファイル • 前処理済みのファイル	オブジェクト・ファイル (.o) までコンパイルします。
-S	• ソースファイル • 前処理済みのファイル	.s 拡張子のアセンブリ・ファイルを生成し、コンパイル処理を停止します。
-syntax	• ソースファイル	構文エラーの診断リストを stdout に出力します。

	• 前処理済みのファイル	構文エラーがないソースファイルについては出力されません。
(デフォルト)	• ソースファイル • 前処理済みのファイル • アセンブリ・ファイル • オブジェクト・ファイル • ライブラリ	実行ファイル (`.out` ファイル)。

コンパイル・フローの制御

オプション	説明
`-c`	オブジェクト・ファイルが生成された後、コンパイルの処理を止めます。C/C++ のソースファイルまたは前処理済みのソースファイルのそれぞれについてコンパイラがオブジェクト・ファイルを生成します。同様に、アセンブリ・ファイルについてもアセンブラがオブジェクト・ファイルを生成します。
`-Kpic, -KPIC`	位置に依存しないコードを生成します。
`-l`*name*	*name* で指定したライブラリにリンクします。
`-nobss_init`	ゼロに初期化される変数を DATA セクションに格納します。
`-P, -F`	C/C++ のソースファイルの前処理が済んだら、コンパイルの処理を止め、その結果をファイルに書き込みます。このファイル名は、コンパイラのデフォルトのファイル命名規則に従って付けられます。
`-S`	アセンブリ・ファイルのみ (拡張子 `.s`) を生成し、コンパイルを停止します。
`-sox[-]` IA-32 のみ	コンパイラのオプションとバージョン情報を実行ファイルに保存します [保存しません]。デフォルトは `-sox-` です。
`-Zp{1|2|4|8|16}`	構造体を 1、2、4、8、16 バイト境界上にパックします。

コンパイル出力の制御

オプション	説明
`-o`*name*	指定したファイル名 (*name*) でアセンブリ・ファイルを生成します。*name* が指定されていない場合、デフォルトのファイル名になります。
`-S`	アセンブリ・ファイルのみ (拡張子 `.s`) を生成し、コンパイルを停止します。

代替ツールと代替パスの指定

前処理、コンパイル、アセンブリ、およびリンクに使用される代替ツールを指定できます。また、選択したツール専用の各種オプションをコマンドラインで呼び出せます。こうした操作は、-Qlocation および -Qoption を使用して行います。次に、それぞれについて説明します。

代替コンポーネントの指定方法

ツールへの代替パスを指定するには、-Qlocation を使用します。このオプションには 2 つの引数を指定します。構文は次のとおりです:

prompt>**icpc -Qlocation**,*tool*,*path*

tool	説明
cpp	コンパイラのフロントエンド・プリプロセッサを指定します。
c	C++ コンパイラを指定します。
asm	アセンブラを指定します。
ld	リンカを指定します。

path は、ツールの所在を示す完全パスです。

他のプログラムへオプションを渡す方法

-Qoption を使用すると、*optlist* で指定したオプションが *tool* に渡されます。*optlist* は、複数のオプションをカンマで区切って並べたリストです。このコマンドの構文は次のとおりです:

prompt>**icpc -Qoption**,*tool*,*optlist*

tool	説明
cpp	コンパイラのフロントエンド・プリプロセッサを指定します。
c	C++ コンパイラを指定します。
asm	アセンブラを指定します。
ld	リンカを指定します。

-oplist は、指定したプログラムに有効な引数文字列です。1 つでも複数でも指定できます。指定する引数がコマンドライン・オプションの場合は、ハイフンを含めてください。スペースかタブ文字を含む引数を指定するときは、その引数全体を二重引用符 ("") で囲んでください。引数を複数指定するときは、それぞれをカンマで区切ってくだ

さい。例えば、コンパイラを使用してソースから実行ファイルを作成し、そのときにリンカを使用してメモリマップを作成するときは、次のようにします。

```
prompt>icpc -Qoption,link,-map,proto.map proto.cpp
```

上の例の `-Qoption, link` というオプションによって `-map` オプションがリンカに渡されます。これは、コンパイルを行うときに各種ツールに各種引数を明示的に渡す方法の 1 つです。また、`-Xlinker val` を使用してリンカに値 (*val*) を渡すこともできます。

データ設定の監視

ここでは、インテル® コンパイラから生成したコードをモニタするオプションについて説明します。

構造体タグのアライメントの指定

構造体および共用体のアライメント境界を指定する方法は 2 つあります:

- ソースファイル内にパックプラグマを配置する
- コマンドラインでアライメント・オプションを入力する

どちらでも構造体タグのアライメント境界を変更できます。

Itanium® ベース・システムでデノーマル値をゼロにフラッシュする

`-ftz` オプションは、アプリケーションが漸次アンダーフロー・モードの場合に、デノーマル結果をゼロにフラッシュします。このオプションは、デノーマル値がアプリケーション動作に影響を与えない場合に使用します。`-ftz` オプションでデノーマル値をゼロにフラッシュすると、アプリケーションのパーフォーマンスが向上する可能性があります。`-ftz` オプションのデフォルト状態はオフです。デフォルトでは、コンパイラは結果を漸次アンダーフローにします。

`-ftz` オプションは、`main()` 関数が含まれているソースに対してのみ使用する必要があります。`-ftz` オプションは、`main()` で開始されるプロセスで FTZ モードをオンにします。初期スレッドおよびそのプロセスによってその後に作成されるあらゆるスレッドは、FTZ モードで動作します。

注

`-O3` オプションは `-ftz` をオンにします。デノーマル結果をゼロにフラッシュしないようにするには、`-ftz-` を使用してください。

ゼロで初期化される変数の割り当て

デフォルトでは、明示的にゼロで初期化される変数は BSS セクションに配置されます。しかし、-nobss_init オプションを使用すると、明示的にゼロで初期化される任意の変数を、必要に応じて DATA セクションに配置できます。

プリコンパイル済みヘッダファイル

インテル® C++ コンパイラは、次のオプションを使用してコンパイル時間を大幅に短縮するプリコンパイル済みヘッダ (PCH) ファイルをサポートします:

- -pch
- -create_pch *filename*
- -use_pch *filename*
- -pch_dir *dirname*

警告

ソースでリストされたヘッダファイルの構成方法によっては、これらのオプションはコンパイル時間を増大させることもあります。PCH オプションを使用してコンパイル時間を最適化する方法については、「ソースファイルの管理」を参照してください。

-pch

-pch オプションは、適切な PCH ファイルを使用するようにコンパイラに指示します。利用可能な PCH ファイルがない場合、*sourcefile*.pchi として作成されます。このオプションは、例 1 のように、複数のソースファイルをサポートします:

例 1 のコマンドライン:
```
prompt>icpc -pch source1.cpp source2.cpp
```

.pchi ファイルが存在しない場合の例 1 の出力
```
"source1.cpp": creating precompiled header file
"source1.pchi"
"source2.cpp": creating precompiled header file
"source2.pchi"
```

.pchi ファイルが存在する場合の例 1 の出力
```
"source1.cpp": using precompiled header file "source1.pchi"
"source2.cpp": using precompiled header file "source2.pchi"
```

注

ヘッダファイルが同じ場合、-pch オプションは他のソースから作成された PCH ファイルを使用します。例えば、-pch を使用して source1.cpp をコンパイルすると、source1.pchi が作成されます。その後、-pch を使用して source2.cpp をコンパイルすると、コンパイラは同じヘッダを検出した場合 source1.pchi を使用します。

-create_pch

コンパイラが PCH ファイル (*filename*) を作成するようにするには、-create_pch *filename* オプションを使用します。このオプションを使用するときは次の点に注意してください:

- *filename* パラメータは必ず指定する必要があります。
- *filename* パラメータはフルパス名で指定できます。
- *filename* へのフルパスは存在していなければなりません。
- *filename* には拡張子 .pch が自動的に追加されます。
- このオプションは -use_pch *filename* と同時に使用することはできません。
- -create_pch *filename* オプションは単一ソースファイルのコンパイルのみをサポートします。

例 2 のコマンドライン:

```
prompt>icpc -create_pch /pch/source32.pchi source.cpp
```

例 2 の出力:

```
"source.cpp": creating precompiled header file
"/pch/source32.pchi"
```

-use_pch *filename*

このオプションは、*filename* で指定された PCH ファイルを使用するようにコンパイラに指示します。このオプションは -create_pch *filename* と同時に使用することはできません。-use_pch *filename* オプションはフルパス名をサポートします。すべてのソースファイルが同じ .pchi ファイルを使用している場合は複数のソースファイルをサポートします。

例 3 のコマンドライン:

```
prompt>icpc -use_pch /pch/source32.pchi source.cpp
```

例 3 の出力:

```
"source.cpp": using precompiled header file
/pch/source32.pchi
```

-pch_dir *dirname*

PCH ファイルへのパス (*dirname*) を指定するには、-pch_dir *dirname* オプションを使用します。このオプションは、-pch、-create_pch *filename*、および -use_pch *filename* オプションとともに使用することができます。

例 4 のコマンドライン:
prompt>**icpc -pch -pch_dir /pch source32.cpp**

例 4 の出力:
```
"source32.cpp": creating precompiled header file
/pch/source32.pchi
```

ソースファイルの管理

ソースファイルの多くが共通のヘッダファイルをインクルードしている場合、共通のヘッダファイルを最初に、続けて #pragma hdrstop ディレクティブを記述します。このプラグマはコンパイラに PCH ファイルの生成を停止するように指示します。例えば、source1.cpp、source2.cpp、および source3.cpp がすべて common.h をインクルードしている場合、common.h の後に #pragma hdrstop を記述してコンパイル時間を最適化します。

```
#include "common.h"
#pragma hdrstop
#include "noncommon.h"
```

-pch オプションを使用してコンパイルした場合:

```
prompt>icpc -pch source1.cpp source2.cpp source3.cpp
```

コンパイラは 3 つのソースファイルについて 1 つの PCH ファイルを生成します:

```
"source1.cpp": creating precompiled header file
"source1.pchi"

"source2.cpp": using precompiled header file "source1.pchi"

"source3.cpp": using precompiled header file "source1.pchi"
```

#pragma hdrstop を使用しないと、common.h に続いて異なるヘッダが記述されている場合、ソースファイルごとに異なる PCH ファイルが作成され、コンパイル時間は長くなります。#pragma hdrstop は、これらの PCH オプションを使用しないコンパイルには影響しません。

リンク

リンク

ここでは、ツールやライブラリとのリンクを制御、カスタマイズするオプション、および ld リンカへの出力を定義するオプションについて説明します。リンカに関する詳細は、ld man ページを参照してください。

オプション	説明
-L*directory*	*directory* で指定したディレクトリを検索してライブラリを探すようリンカに命令します。
-Qoption,*tool*,*list*	一連のコンパイル処理の中で、アセンブラやリンカなど別のプログラムに引数リストを渡します。
-shared	このリンカ・オプションは、実行ファイルの代わりに、動的共用オブジェクト (DSO) をビルドするようコンパイラに命令します。
-shared-libcxa	-shared-libcxa は、-static-libcxa と逆の効果が得られます。このオプションが使用されると、インテルが提供する libcxa C++ ライブラリが動的にリンクされ、-static オプションが使用された場合の静的リンクを無効にします。注: デフォルトでは、すべての C++ 標準およびサポート・ライブラリは動的にリンクされます。
-i_dynamic	インテルが提供するライブラリをすべて動的にリンクします。
-static	ライブラリをすべて静的にリンクします。 -static を使用していない場合: • /lib/ld-linux.so.2 がリンクされます。 • その他のすべてのライブラリは動的にリンクされます。 -static を使用している場合: • /lib/ld-linux.so.2 はリンクされません。 • その他のすべてのライブラリは静的にリンクされます。
-static-libcxa	デフォルトでは、インテルが提供する libcxa C++ ライブラリは動的にリンクされます。標準ライブラリをデフォルト設定のままでリンクしながら、libcxa を静的にリンクするには、コマンドラインで -static-libcxa を使用します。
-Bstatic	このオプションは、リンカのコマンドライン中で指定されます。このオプションは、コマンドラインから渡されたライブラリの動作を制御するために使用されます。
-Bdynamic	このオプションは、リンカのコマンドライン中で指定されます。このオプションは、コマンドラインから渡されたライブラリの動作を制御するために使用されます。

リンクを行わない設定にする

リンクを行わない設定にするには、-c オプションを使用します。例えば、次のコマンドは file.o と file2.o という 2 つのオブジェクト・ファイルを生成します。

```
prompt>icpc -c file1.cpp file2.cpp
```

注

上記のコマンドを実行しても、これらのファイルはリンクされません。

デバッグ

概要: デバッグ・オプション

ここでは、コンパイルのデバッグやコンパイル・エラーの表示およびチェックを行うツールとして使用できる基本的なコマンドライン・オプションについて説明します。このセクションで説明するオプションは次のとおりです:

- 構文解析のみを行う
- 最適化とデバッグ

構文解析のみを行う

C++言語のエラーに関する構文解析が終了した後ソースファイルの処理を停止するには、-syntax オプションを使用します。これは、ソースファイルが構文的または意味的に正しいかどうかをすばやく確認するための方法です。コンパイラから出力ファイルは生成されません。次の例では、コンパイラは prog.cpp という名前のファイルをチェックして、診断情報を標準エラー出力に表示します:

```
prompt>icpc -syntax prog.cpp
```

最適化とデバッグ

ここでは、コンパイルのデバッグやコンパイル・エラーの表示およびチェックに使用できるコマンドライン・オプションについて説明します。最適化のときにデバッグ情報を得ることができるオプションは次のとおりです:

オプション	説明
-O0	最適化を無効にします。-fp オプションを有効にします。
-g	ソースレベルでのデバッガで使用するため、オブジェクト・コードの中にシンボリック・デバッグ情報と行番号を生成します。コマンドラインで -g とともに -O1、-O2、または -O3 が明示的に指定されていなければ、-O2 をオフにして -O0 をデフォルトにします。
-fp IA-32 のみ	EBP レジスタを汎用レジスタとして使用できないようにします。
オプション	**-fp に対する影響**
-O1、-O2、または -O3	-fp を無効にします。
-O0	-fp を有効にします。

最適化とデバッグの組み合わせ

-O0 オプションはすべての最適化をオフにするため、最適化が行われる前にプログラムをデバッグすることができます。デバッグ情報を得るには、-g オプションを使用します。コマンドラインで、オブジェクト・ファイルにシンボリック・デバッグ情報を生成する -g オプションとともに、最適化オプション -O1、-O2、または -O3 が指定されると、コンパイラはシンボリック・デバッグ対応のコードを生成します。

-g オプションとともに -O1、-O2、または -O3 オプションを指定すると、返されるデバッグ情報の一部が最適化の副作用として不正確になることがあります。

最適化とデバッグ・オプションは明示的に選択するようにしてください:

- 最適化の影響を受けないようにプログラムをデバッグする必要がある場合は、すべての最適化をオフにする -O0 オプションを使用してください。
- 最適化を有効にしてプログラムをデバッグする必要がある場合は、コマンドラインで -g オプションとともに -O1、-O2、または -O3 オプションを指定することができます。

注

-O1、-O2、または -O3 オプションを指定しないで -g オプションを指定すると、プログラムの速度は遅くなります。この場合、-g オプションはプログラムの速度を遅くする -O0 オプションをオンにするためです。-O2 と -g オプションが指定された場合、コードは -g が指定されていない場合とほぼ同じ速度で実行されます。

次の表は、-g オプションと最適化オプションを組み合わせて使用した場合の効果を説明したものです。

オプション	結果
-g	デバッグ情報が生成され、-O0 (最適化無効) が有効になります。IA-32 を対象にコンパイルする場合は、fp が有効になります。
-g -O1	デバッグ情報が生成され、-O1 による最適化が有効になります。
-g -O2	デバッグ情報が生成され、-O2 による最適化が有効になります。
-g -O3 -fp	デバッグ情報が生成され、-O3 による最適化が有効になります。IA-32 を対象にコンパイルする場合は、fp が有効になります。

デバッグとアセンブル

アセンブリ・ファイルはデバッグ情報なしで生成されますが、オブジェクト・ファイルを生成すると、そのオブジェクト・ファイルにはデバッグ情報が含まれます。このため、オブジェクト・ファイルをリンクしてから GDB デバッガを使用すると、完全なシンボリック表現が得られます。

ライブラリの使用

概要: ライブラリの使用

インテル® C++ コンパイラは、GNU* C ライブラリ、Dinkumware* C++ ライブラリ、および標準 C++ ライブラリを使用します。各ライブラリに関する資料は、次の URL を参照してください:

GNU C ライブラリ

http://www.gnu.org/software/libc/manual/html_node/index.html (英語)

Dinkumware C++ ライブラリ

http://www.dinkumware.com/htm_cpl/lib_cpp.html (英語)

標準 C++ ライブラリ

http://gcc.gnu.org/onlinedocs/libstdc++/documentation.html (英語)

デフォルト・ライブラリ

次のライブラリがインテル® C++ コンパイラとともに提供されます:

ライブラリ	説明
`libguide.a` `libguide.so`	OpenMP* 用
`libguide_stats.a` `libguide_stats.so`	パフォーマンス統計とプロファイル情報を含むパラレライザ・ツール用の OpenMP スタティック・ライブラリ
`libompstub.a`	OpenMP の未使用時に OpenMP サブルーチンの参照を解決するライブラリ
`libsvml.a`	SVML (Short Vector Mathematical Library)
`libirc.a`	PGO および CPU ディスパッチ用にインテルが提供するライブラリ
`libircmt.a`	`libirc.a` のマルチスレッド・バージョン
`libimf.a` `libimf.so`	インテル® 数値演算ライブラリ

`libcprts.a`
`libcprts.so`
`libcprts.so.3`
	Dinkumware* C++ ライブラリ

libunwind.a libunwind.so libunwind.so.3	Unwinder ライブラリ
libcxa.a libcxa.so libcxa.so.3	C++ 機能のインテル・ランタイム・サポート
libcxaguard.a libcxaguard.so libcxaguard.so.3	-cxxlib-gcc オプションとともに相互運用性のサポートに使用されます。 「gcc との相互運用性」を参照してください。

-cxxlib-gcc オプションを起動すると、次の置換が行われます:

- libcprts は、gcc* ディストリビューション (3.2 以降) の libstdc++ と置換されます
- libcxa と libunwind は、gcc* ディストリビューション (3.2 以降) の libgcc と置換されます

別のライブラリや補助的なライブラリにプログラムをリンクするときは、コマンドラインの最後に目的のライブラリを指定してください。例えば、prog.cpp をコンパイルして mylib.a とリンクするには、次のコマンドを使用します:

```
prompt>icpc prog.cpp mylib.a
```

ld リンカの場合、mylib.a ライブラリは、コマンドライン上で libimf.a ライブラリよりも前に指定してください。

警告

Linux* のシステム・ライブラリとコンパイラ・ライブラリは、-align オプション付きではビルドされません。このため、-align オプション付きでコンパイルし、コンパイラが配布したライブラリまたはシステム・ライブラリへ呼び出しを行い、インターフェイスに long long、double、または long double があると、アライメントの違いにより、間違った結果になります。 -align 付きでビルドされたコードは、(-align なしでは動作しない場合) -align 付きでビルドされない限り、インターフェイスでこれらの型を使用するライブラリへの呼び出しはできません。

数値演算ライブラリ

インテルの数値演算ライブラリ、libimf.a は、標準 C ランタイム・ライブラリにある数値演算関数の最適化されたバージョンを含みます。libimf.a にある関数は、インテル® Pentium® III プロセッサおよびインテル Pentium 4 プロセッサでのプログラム実行速度が最適化されています。Itanium® コンパイラも、Itanium ベースのシステ

ムでのパフォーマンスを最適化するように設計された libimf.a を含みます。インテル数値演算ライブラリは、デフォルトでリンクされます。

「ライブラリの管理」および「インテル® 数値演算ライブラリ」を参照してください。

インテル共有ライブラリ

デフォルトでは、インテル® C++ コンパイラはインテルが提供する C++ ライブラリを動的にリンクします。GNU* と Linux* システム・ライブラリも動的にリンクされます。

共有ライブラリのオプション

オプション	説明
-i_dynamic	インテルが提供する C++ ライブラリを動的にリンクするには、-i_dynamic オプションを使用します (デフォルト)。このオプションは、アプリケーション・バイナリのサイズを小さくしますが、すべてのダイナミック版ライブラリが、アプリケーションが実行されるシステム上にインストールされている必要があります。
-shared	-shared オプションは、実行ファイルの代わりに DSO (Dynamic Shared Object) を作成するように、コンパイラに指示します。詳細は、ld man ページを参照してください。

「リンク」も参照してください。

ライブラリの管理

環境変数 LD_LIBRARY_PATH には、セミコロンで区切られたディレクトリ・リストが含まれています。リンカは、ライブラリ (.a) ファイルを探すときに、このディレクトリの中を検索します。他のライブラリをリンカに検索させたいときは、その名前を、LD_LIBRARY_PATH、コマンドライン、または応答ファイル (次の注を参照) のいずれかに追加できます。どの場合も、追加したライブラリ名のほうが先にリンカに渡され、その後、当該ドライバがいつも指定するインテルのライブラリがリンカに渡されます。

注

応答ファイルはコマンドラインで指定された場所で処理されます。ライブラリが応答ファイルで指定され、オブジェクト・ファイルからの参照が応答ファイルの後で見つかった場合、これらのライブラリでは解決されません。

LD_LIBRARY_PATH の変更

ディレクトリ (例えば、/libs) を LD_LIBRARY_PATH に追加する場合、次のいずれかを行います:

- コマンドライン: prompt>**export LD_LIBRARY_PATH=/libs:$LD_LIBRARY_PATH**
- スタートアップ・ファイル **export LD_LIBRARY_PATH=/libs:$LD_LIBRARY_PATH**

file.cpp をコンパイルして mylib.a ライブラリをリンクするには、次のコマンドを入力します:

```
prompt>icpc file.cpp mylib.a
```

ファイル名は、次の順番でリンカに渡されます:

1. オブジェクト・ファイル
2. コマンドライン、応答ファイル、または設定ファイルのいずれかで指定されたオブジェクトまたはライブラリ
3. インテル® 数値演算ライブラリ libimf.a

非共有ライブラリのコンパイル

概要: 非有用ライブラリのコンパイル

このセクションでは、次の情報について説明します:

グローバル・シンボルと可視属性

シンボル・プリエンプション

シンボルの可視属性の明示的な指定

その他の可視属性関連のコマンドライン・オプション

グローバル・シンボルと可視属性

グローバル・シンボルは、宣言されたコンパイル単位 (単一ソースファイルとそのインクルード・ファイル) の外部で見えるシンボルです。C/C++ では、これは static キーワードのないファイルレベルでの宣言を意味します。次に例を示します:

```
int x = 5;          // global data definition
extern int y;       // global data reference
int five()          // global function definition
{ return 5; }
extern int four(); // global function reference
```

完成したプログラムは、メイン・プログラムとメイン・プログラムによって参照されるデータおよび関数の定義を含む複数の共有可能なオブジェクト (.so) ファイルから構成されます。同様に、共有可能なオブジェクトは他の共有可能なオブジェクトで定義されているデータや関数を参照します。同時に実行している複数のプロセスがその仮想メモリにマップされた共有可能なオブジェクトを含む場合、物理メモリにはそのオブジェクトの読み取り専用部分の 1 つのコピーのみが存在するため、これらのオブジェクトは共有可能なオブジェクトと呼ばれます。メイン・プログラム・ファイルとそのファイルが参照する共有可能なオブジェクトは、プログラムのコンポーネントと呼ばれます。

コンパイル単位中の各グローバル・シンボル定義や参照には、それらが定義されたコンポーネントの外部からどのように参照されるかを制御する可視属性があります。可視属性の値は 5 つあります:

- EXTERNAL – コンパイラはシンボルを別のコンポーネントで定義されたものとして扱います。定義の場合、これは別のコンポーネントにある同じ名前の定義によってシンボルが無効にされる (プリエンプトされる) とコンパイラが仮定することを意味します。「シンボル・プリエンプション」を参照してください。関数シンボルの可視属性が external の場合、コンパイラはそのシンボルが間接的に呼び出され、間接呼び出しスタブをインライン展開できることを知っています。
- DEFAULT – 他のコンポーネントはシンボルを参照することができます。シンボル定義は別のコンポーネントにある同じ名前の定義によって無効に (プリエンプト) されます。
- PROTECTED – 他のコンポーネントはシンボルを参照することができますが、別のコンポーネントにある同じ名前の定義によってプリエンプトすることはできません。
- HIDDEN – 他のコンポーネントはシンボルを直接参照することはできません。しかし、そのアドレスは間接的に (例えば、別のコンポーネントの関数への呼び出し引数として、または別のコンポーネントの関数でデータ項目参照にそのアドレスを格納して) 他のコンポーネントに渡すことができます。
- INTERNAL – シンボルは、シンボルが定義されたコンポーネントの外部から直接的にも間接的にも参照することはできません。

スタティックなローカルシンボル (C/C++ では、ファイルスコープでの宣言またはキーワード static による宣言) の通常の可視属性は HIDDEN で、他のコンポーネント (または同じコンポーネント内の他のコンパイル単位) から直接参照することはできませんが、間接的に参照することはできます。

注

可視属性は定義にも参照にも適用されます。シンボル参照の可視属性は、対応する定義がその可視属性になるというアサーションです。

シンボル・プリエンプション

共有可能なオブジェクトの関数やデータ項目を使用するときに、それらの代わりに独自に定義したものを使用することができます。例えば、標準 C ランタイム・ライブラリの共有可能なオブジェクト `libc.so` を使用するときに、独自のヒープ管理ルーチン `malloc()` および `free()` の定義を使用することができます。この場合、`lib.so` 内の `malloc()` および `free()` への呼び出しが `libc.so` にある定義ではなく独自のルーチンの定義を呼び出すことが重要です。独自の定義は、共有可能なオブジェクト内の定義を無効に (プリエンプト) します。

共有可能なオブジェクトのこの特徴は、シンボル・プリエンプションと呼ばれます。ランタイム・ローダがコンポーネントをロードするとき、コンポーネント内の default 可視属性のシンボルはすべて、既にロードされているコンポーネント内の同じ名前のシンボルによるプリエンプションに従います。メイン・プログラム・イメージは常に最初にロードされるので、メイン・プログラムが定義するシンボルはプリエンプトされません。

default 可視属性のシンボルは実行時までメモリアドレスにバインドされないため、シンボル・プリエンプションが発生する可能性により多くのコンパイラの最適化は禁止されます。例えば、default 可視属性のルーチンへの呼び出しは、コンパイル単位が共有可能なオブジェクトにリンクされた場合にプリエンプトされるため、インライン展開することができません。プリエンプト可能なデータシンボルは、名前が異なるコンポーネント中のシンボルにバインドされるため、GP-相対アドレス指定を使用してアクセスすることはできません。GP-相対アドレスはコンパイル時にはわかりません。

シンボル・プリエンプションは、コンパイラの最適化と全く逆の効果であるため、ほとんど使用されていない機能です。この理由のため、コンパイラはすべてのグローバル・シンボル定義をデフォルトでプリエンプト可能ではない (つまり、protected 可視属性) として扱います 。他のコンパイル単位で定義されているシンボルへのグローバル参照は、デフォルトでプリエンプト可能である (つまり、default 可視属性) と仮定されます。すべてのグローバル定義や参照をプリエンプト可能にする場合は、このデフォルトを無視する `-fpic` オプションを指定してください。

シンボルの可視属性の明示的な指定

データまたは関数宣言で visibility 属性を使用して個々のシンボルの可視属性を明示的に設定することができます。
例:

```
int i   attribute   ((visibility("default")));
void __attribute__ ((visibility("hidden"))) x () {...}
extern void y() __attribute__ ((visibilty("protected");
```

visibility 宣言属性には、次の 5 つのキーワードを使用することができます:

- external
- default
- protected
- hidden
- internal

visibility 宣言属性の値は、-fvisibility、-fpic、または -fno-common 属性のデフォルト設定よりも優先されます。

複数のシンボルで同じ visibility 属性を指定する場合、次の 5 つのコマンドライン・オプションのうち 1 つを使用して可視属性を設定することができます:

- -fvisibility-external=*file*
- -fvisibility-default=*file*
- -fvisibility-protected=*file*
- -fvisibility-hidden=*file*
- -fvisibility-internal=*file*

file は、可視属性を設定するシンボル名のリストを含むファイルのパス名です。 ファイル中のシンボル名は、余白 (ブランク、TAB 文字、または改行) で区切ります。例えば、コマンドライン・オプションが次のような場合:

```
-fvisibility-protected=prot.txt
```

そして、ファイル prot.txt が次のような場合:

```
a
     b c d
  e
```

シンボル a、b、c、d、および e の可視属性が protected に設定されます。これは、各シンボルで次のように宣言した場合と同じです:

```
__attribute__ ((visibility=("protected")))
```

可視属性を明示的に設定するこれらの 2 つの方法 (宣言で `__attribute((visibilty()))` を使用する方法とファイルでシンボル名を指定する方法) は互いに排他的であり、両方を同時に行うことはできません。

次のコマンドライン・オプションの 1 つを使用してシンボルのデフォルトの可視属性を設定することができます:

- `-fvisibility=external`
- `-fvisibility=default`
- `-fvisibility=protected`
- `-fvisibility=hidden`
- `-fvisibility=internal`

このオプションは、可視属性リストファイルで指定がなく、宣言に `__attribute__((visibilty()))` がないシンボルの可視属性を設定します。例えば、コマンドライン・オプションが次のような場合:

```
-fvisibility=protected -fvisibility-default=prot.txt
```

そして、ファイル `prot.txt` の内容が前の例と同じ場合、a、b、c、d、および e を除くグローバル・シンボルの可視属性が protected に設定されます。しかし、これらの 5 つのシンボルの可視属性は default に設定されプリエンプト可能になります。

その他の可視属性関連のコマンドライン・オプション

-fminshared

`-fminshared` オプションは、コンパイル単位がメイン・プログラム・コンポーネントの一部になり、共有可能なオブジェクトの一部としてリンクされないことを示します。メイン・プログラムで定義されたシンボルはプリエンプトできないため、このオプションにより、コンパイラは default 可視属性で宣言されたシンボルを protected 可視属性であるかのように扱います (つまり、`-fminshared` は `-fvisibility=protected` を暗示します)。また、コンパイラはメイン・プログラム用に位置に依存しないコードを生成する必要はありません。グローバル・オフセット・テーブル (GOT) のサイズを減らしてメモリのトラフィックを減らす、絶対アドレス指定を使用することができます。

-fpic

`-fpic` オプションは、完全なシンボル・プリエンプションを指定します。グローバル・シンボル定義は、明示的に他の方法で指定されなければ、グローバル・シンボル参照と同様に default 可視属性 (つまり、プリエンプト可能) になります。

-fno-common

初期化子がなく、extern または static キーワードのない次に示す通常の C/C++ ファイルスコープ宣言では、共通シンボルとして表現されます。

```
int i;
```

他のコンパイル単位にその名前用のグローバル定義がない場合を除いて、これらのシンボルは外部参照として扱われ、リンカはこれらのシンボルにメモリを割り当てます。-fno-common オプションは、コンパイラが共通シンボルになるシンボルをグローバル定義として扱い、コンパイル時にシンボルにメモリを割り当てるようにします。これは、コンパイラがシンボルにアクセスするときにより効率的な GP 相対アドレス指定を使用することを可能にします。

gcc* との互換性

gcc* との互換性

インテル® C++ コンパイラで作成された C 言語オブジェクト・ファイルは、GNU* gcc コンパイラと glibc (GNU C 言語ライブラリ) とバイナリ互換です。C 言語オブジェクト・ファイルは、インテル・コンパイラまたは gcc コンパイラのいずれかでリンクすることができます。しかし、リンカにインテル・ライブラリを正確に渡すには、インテル・コンパイラを使用してください。詳細は、「リンク」および「デフォルト・ライブラリ」を参照してください。

GNU C には、ISO 標準 C にない、さまざまな非標準の機能が含まれています。C 言語に対するこれらの拡張機能のいくつかは、このバージョンのインテル C++ コンパイラでサポートされています。詳細は、http://www.gnu.org/home.ja.html を参照してください。

gcc 言語拡張	インテル・コンパイラのサポート	説明と例
Statements and Declarations in Expressions	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/Statement-Exprs.html#Statement%20Exprs
Locally Declared Labels	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Local-Labels.html#Local%20Labels
Labels as Values	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Labels-as-Values.html#Labels%20as%20Values
Nested Functions	非サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Nested-Functions.html#Nested%20Functions
Constructing Function Calls	非サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Constructing-Calls.html#Constructing%20Calls
Naming an Expression's Type	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Naming-Types.html#Naming%20Types
Referring to a Type with typeof	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Typeof.html#Typeof
Generalized Lvalues	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Lvalues.html#Lvalues
Conditionals with Omitted Operands	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Conditionals.html#Conditionals
Double-Word Integers	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Long-Long.html#Long%20Long
Complex Numbers	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/

		Complex.html#Complex
Hex Floats	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Hex-Floats.html#Hex%20Floats
Arrays of Length Zero	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Zero-Length.html#Zero%20Length
Arrays of Variable Length	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Variable-Length.html#Variable%20Length
Macros with a Variable Number of Arguments.	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Variadic-Macros.html#Variadic%20Macros
Slightly Looser Rules for Escaped Newlines	非サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Escaped-Newlines.html#Escaped%20Newlines
String Literals with Embedded Newlines	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Multi-line-Strings.html#Multi-line%20Strings
Non-Lvalue Arrays May Have Subscripts	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Subscripting.html#Subscripting
Arithmetic on void-Pointers	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Pointer-Arith.html#Pointer%20Arith
Arithmetic on Function-Pointers	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Pointer-Arith.html#Pointer%20Arith
Non-Constant Initializers	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Initializers.html#Initializers
Compound Literals	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Compound-Literals.html#Compound%20Literals
Designated Initializers	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Designated-Inits.html#Designated%20Inits
Cast to a Union Type	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Cast-to-Union.html#Cast%20to%20Union
Case Ranges	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Case-Ranges.html#Case%20Ranges
Mixed Declarations and Code	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Mixed-Declarations.html#Mixed%20Declarations
Declaring Attributes of Functions	ほぼサポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Function-Attributes.html#Function%20Attributes
Attribute Syntax	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Attribute-Syntax.html#Attribute%20Syntax
Prototypes and Old-Style Function Definitions	非サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Function-Prototypes.html#Function%20Prototypes
C++ Style Comments	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/C---Comments.html#C++%20Comments
Dollar Signs in Identifier Names	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Dollar-Signs.html#Dollar%20Signs
The Character ESC in	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/

Constants		Character-Escapes.html#Character%20Escapes
Specifying Attributes of Variables	ほぼサポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Variable-Attributes.html#Variable%20Attributes
Specifying Attributes of Types	ほぼサポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Type-Attributes.html#Type%20Attributes
Inquiring on Alignment of Types or Variables	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Alignment.html#Alignment
An Inline Function is As Fast As a Macro	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Inline.html#Inline
Assembler Instructions with C Expression Operands	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Extended-Asm.html#Extended%20Asm
Controlling Names Used in Assembler Code	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Asm-Labels.html#Asm%20Labels
Variables in Specified Registers	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Explicit-Reg-Vars.html#Explicit%20Reg%20Vars
Alternate Keywords	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Alternate-Keywords.html#Alternate%20Keywords
Incomplete enum Types	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Incomplete-Enums.html#Incomplete%20Enums
Function Names as Strings	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Function-Names.html#Function%20Names
Getting the Return or Frame Address of a Function	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Return-Address.html#Return%20Address
Using Vector Instructions Through Built-in Functions	いくつかサポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Vector-Extensions.html#Vector%20Extensions
Other built-in functions provided by GCC	ほぼサポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Other-Builtins.html#Other%20Builtins
Built-in Functions Specific to Particular Target Machines	非サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Target-Builtins.html#Target%20Builtins
Pragmas Accepted by GCC	非サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Pragmas.html#Pragmas
Unnamed struct/union fields within structs/unions	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Unnamed-Fields.html#Unnamed%20Fields
Minimum and Maximum operators in C++	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Min-and-Max.html#Min%20and%20Max
When is a Volatile Object Accessed?	非サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/Volatiles.html#Volatiles
Restricting Pointer	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/

Aliasing		Restricted-Pointers.html#Restricted%20Pointers
Vague Linkage	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/Vague-Linkage.html#Vague%20Linkage
Declarations and Definitions in One Header	非サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/C---Interface.html#C++%20Interface
Where's the Template?	extern テンプレートをサポート	http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/Template-Instantiation.html#Template%20Instantiation
Extracting the function pointer from a bound pointer to member function	非サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/Bound-member-functions.html#Bound%20member%20functions
C++-Specific Variable, Function, and Type Attributes	サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/C---Attributes.html#C++%20Attributes
Java Exceptions	非サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/Java-Exceptions.html#Java%20Exceptions
Deprecated Features	非サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/Deprecated-Features.html#Deprecated%20Features
Backwards Compatibility	非サポート	http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/Backwards-Compatibility.html#Backwards%20Compatibility

注

インテル C++ コンパイラは、http://www.gnu.org/software/binutils/manual/gas-2.9.1/html_chapter/as_16.html の gcc ドキュメンテーションで定義されているように、アセンブラ・コードが AT&T* System V/386 構文を使用する場合、gcc* スタイルのインライン・アセンブリをサポートします。

gcc* との相互運用性

あるコンパイラで生成されたオブジェクト・ファイルやライブラリが別のコンパイラで生成されたオブジェクト・ファイルやライブラリとリンク可能で、生成される実行ファイルが正しく動作する場合、C++ コンパイラは相互運用可能です。インテル® C++ コンパイラ 8.0 は、GNU gcc* コンパイラとの相互運用性と互換性を大幅に改良しました。このセクションでは、新しい相互運用性オプションについて説明します。

互換性の詳細については、「gcc との互換性」を参照してください。

相互運用性のコンパイラ・オプション

インテル C++ コンパイラのオプションで gcc との相互運用性に影響するオプションは次のとおりです:

- `-cxxlib-gcc`
- `-gcc-name`
- `-gcc-version`

-cxxlib-gcc オプション

`-cxxlib-gcc` オプションは、gcc コンパイラに含まれている C++ ライブラリとヘッダファイルを使用してアプリケーションをビルドします。次のファイルをインクルードします:

- `libstdc++` 標準 C++ ヘッダファイル
- `libstdc++` 標準 C++ ライブラリ
- `libgcc` C++ 言語サポート

`-cxxlib-gcc` オプションを使用してアプリケーションをコンパイルおよびリンクして生成された C++ オブジェクト・ファイル、ライブラリ、実行ファイルは、gcc 3.2 で生成された C++ オブジェクト・ファイル、ライブラリ、実行ファイルと相互運用可能です。これは、インテル・コンパイラ 8.0 で生成された C++ コードが gcc 3.2 でビルドされたサードパーティの C++ ライブラリとともに使用できることを意味します。

`-cxxlib-gcc` オプションは gcc 3.2 を含む Linux ディストリビューションでのみ使用することができます。これは C++ ABI 規格で必要です。

デフォルトでは、インテル C++ コンパイラは製品に含まれているライブラリとヘッダファイルを使用します。g++ でコンパイルされた (gnu C++ ヘッダに対してコンパイルされた) コードをリンクすると、ヘッダの違いにより互換性がなくなるため、実行時にエラーが発生する可能性があります。

インテル C++ ライブラリに対して 1 つの共有ライブラリをビルドし、gnu C++ ライブラリに対して 2 つ目の共有ライブラリをビルドし、1 つのアプリケーションで両方のライブラリを使用すると、2 つの C++ ランタイム・ライブラリを使用することになります。アプリケーションは両方のライブラリからシンボルを使用するため、次の問題が発生します:

- 一部初期化されたライブラリ
- アクセスされていないバッファへのデータ入力による `I/O` 操作の消失
- その他の予測できない結果

インテル C++ コンパイラは 1 つのアプリケーションで複数のランタイム・ライブラリをサポートしません。

警告

複数のランタイム・ライブラリを使用してアプリケーションのコンパイルが正常に行われた場合でも、特に新しいコードが共有ライブラリに対してリンクされるとき、アプリケーションは非常に不安定になります。

アプリケーションが g++ で生成されたソースファイルとインテル C++ コンパイラで生成されたソースファイルを含む場合、`-cxxlib-gcc` オプションを使用するようにしてください。このオプションは、1 セットのランタイム・ライブラリでビルドするため、インテル・コンパイラに g++ ライブラリとヘッダファイルを使用するように指示します。この結果、プログラムは正常に動作するようになります。

-gcc-name オプション

`-gcc-name=`*name* は `-cxxlib-gcc` とともに使用するオプションで、コンパイラが gcc C++ ライブラリを見つけられない場合に g++* の場所を指定します。標準の gcc インストールを行わなかった場合、このオプションを使用する必要があります。

-gcc-version

`-gcc-version=`*nnn* オプションは、コンパイラの動作を gcc バージョン *nnn* と互換にします。このバージョンのインテル C++ コンパイラは `-gcc-version=320` (デフォルト) をサポートします。

デフォルトのライブラリとヘッダ

`-cxxlib-icc` オプションは、インテル・コンパイラに含まれている C++ ライブラリとヘッダファイルを使用してアプリケーションをビルドします。次のファイルをインクルードします:

- `libcprts` 標準 C++ ヘッダファイル
- `libcprts` 標準 C++ ライブラリ
- `libcxa` と `libunwind` C++ 言語サポート

`-cxxlib-icc` オプションはデフォルトでオンで、すべてのサポートされている Linux ディストリビューションで使用することができます。リリースノートを参照してください。

対応するライブラリとヘッダの一覧

インテル・ライブラリ/ヘッダ	gcc ライブラリ/ヘッダ
libcprts	libstdc++
libcxa/libunwind	libgcc

gcc 事前定義済みマクロ

インテル C++ コンパイラ 8.0 には、gcc でもサポートされている次の事前定義済みマクロが含まれています:

- __GNUC__
- __GNUC_MINOR__
- __GNUC_PATCHLEVEL__

これらのマクロ定義が必要ない場合は、-no-gcc オプションを指定してください。gcc との相互運用性 (-cxxlib-gcc) が必要な場合、-no-gcc コンパイラ・オプションを使用しないでください。

「GNU 環境変数」も参照してください。

gcc 組込み関数

このバージョンのインテル® C++ コンパイラは、次の gcc 組込み関数をサポートします:

```
__builtin_abs
__builtin_labs
__builtin_cos
__builtin_cosf
__builtin_fabs
__builtin_fabsf
__builtin_memcmp
__builtin_memcpy
__builtin_sin
__builtin_sinf
__builtin_sqrt
__builtin_sqrtf
__builtin_strcmp
__builtin_strlen
__builtin_strncmp
__builtin_abort
__builtin_prefetch
```

```
__builtin_constant_p
__builtin_printf
__builtin_fprintf
__builtin_fscanf
__builtin_scanf
__builtin_fputs
__builtin_memset
__builtin_strcat
__builtin_strcpy
__builtin_strncpy
__builtin_exit
__builtin_strchr
__builtin_strspn
__builtin_strcspn
__builtin_strstr
__builtin_strpbrk
__builtin_strrchr
__builtin_strncat
__builtin_alloca
__builtin_ffs
__builtin_index
__builtin_rindex
__builtin_bcmp
__builtin_bzero
__builtin_sinl
__builtin_cosl
__builtin_sqrtl
__builtin_fabsl
__builtin_frame_address (IA-32 のみ)
__builtin_return_address (IA-32 のみ)
```

gcc 関数属性

このバージョンのインテル® C++ コンパイラは、次の gcc 関数属性をサポートします:

- `noinline` – 関数がインライン展開されることを防ぎます
- `always_inline` – 最適化が指定されていない場合でも関数をインライン展開します
- `used` – 関数が参照されていない場合でも関数のコードを出力します

例

```
int round_sqrt(int) __attribute__ ((always_inline));
```

この例では、最適化が指定されていない場合でも関数 `round_sqrt()` はインライン展開されます。

言語の適合性

適合性オプション

オプション	説明
-ansi	GNU* ANSI と同じ働きをします
-strict_ansi	ANSI 規格に厳密に準拠しているダイアレクトを選択します
-ansi_alias[-]	-ansi_alias によって、コンパイラは次の点を前提にします: • 配列が外部からアクセスされません。 • ポインタを非ポインタ型にキャストしません。また、その逆もしません。 • 2 つの異なるスカラ型のオブジェクトへの参照はエイリアス不可です。例えば、int のオブジェクトは float 型のオブジェクトと、または float 型のオブジェクトは double 型のオブジェクトとエイリアスの設定はできません。 プログラムが上記の条件を満たす場合、-ansi_alias フラグを設定することでプログラムの最適化が向上します。ただし、プログラムが上記の条件を 1 つでも満たさない場合、コンパイラは -ansi_alias フラグを設定することで誤ったコードを生成する可能性があります。

C の標準規格との適合性

インテル® C++ コンパイラは、次のどちらの C コードにも対応できるように設定できます:

- GNU* ANSI と同じ ANSI 準拠のコード (-ansi オプション)
- ANSI 規格に厳密に準拠しているダイアレクト (-strict_ansi オプション)

このコンパイラは、デフォルトでは、ANSI/ISO 標準に制限されずに拡張を受け入れるように設定されます。

ANSI/ISO 標準準拠の C 言語について

インテル C++ コンパイラは、C 言語のコンパイルについて定めた ANSI/ISO 標準 (ISO/IEC 9899:1990) に準拠しています。ANSI/ISO 標準に準拠するためには、翻訳処理に要求する最低要件を満たしている必要があります。インテル C++ コンパイ

ラは、翻訳処理に対して ANSI/ISO の要求している最低要件をすべて満たしています。

コンパイラ付属のマクロ

C 言語の ANSI/ISO 標準に準拠するためには、コンパイラに所定の事前定義済みマクロが付属していなければなりません。次の表は、同標準に従ってインテル C++ コンパイラに組み込まれたマクロの一覧です。

コンパイラは、標準から要求される事前定義マクロのほかにも事前定義マクロをいくつか用意しています。

マクロ	説明
__cplusplus	名前 __cplusplus は、C++ 変換単位をコンパイルするときに定義されます。
__DATE__	コンパイルの日付。Mmm dd yyyy 形式の文字列リテラルです。
__FILE__	コンパイルされるファイルの名前を表す文字列リテラルです。
__LINE__	現在の行番号。10 進数の定数で表現します。
__STDC__	名前 __STDC__ は、C 変換単位をコンパイルするときに定義されます。
__TIME__	コンパイルの時間。hh:mm:ss 形式の文字列リテラルです。

C99 サポート

-c99 オプションを使用することによって、このバージョンのインテル C++ コンパイラは、次の C99 機能をサポートします:

- 制限付きポインタ (restrict キーワード。-restrict で利用可能)。下記の注を参照してください。
- 可変長配列
- 柔軟な配列メンバ
- 複素数のサポート (_Complex キーワード)
- 16 進浮動小数点定数
- コンパウンド・リテラル
- 指定初期化子
- 混在する宣言とコード
- 可変長引数を持つマクロ
- インライン関数 (inline キーワードを使用)
- ブール型 (_Bool キーワード)

注

`-restrict` オプションは、ANSI 標準規格で定義された `restrict` キーワードを認識します。`restrict` キーワードでポインタを限定することによって、ポインタ経由でアクセスされるオブジェクトが、与えられたスコープ内では、当該ポインタのみによってアクセスされることを宣言できます。この制限が正しい場合のみこの `restrict` キーワードを使用することが重要です。この場合、プログラムの正当性には影響を与えませんが、より良い最適化が可能になる場合があります。

次の C99 機能はサポートされていません:

- `#pragma STDC FP_CONTRACT`
- `#pragma STDC FENV_ACCESS`
- `#pragma STDC CX_LIMITED_RANGE`
- `long double` (128 ビット表記)

C++ の標準規格との適合性

インテル® C++ コンパイラは、C++ 言語の ANSI/ISO 標準 (ISO/IEC 14882:1998) に準拠していますが、テンプレート用の `export` キーワードは実装していません。

コンパイラの最適化

最適化レベル

概要: 最適化レベル

このセクションでは、コマンドライン・オプション、-O0、-O1、-O2、および -O3 について説明します。-O0 オプションは、最適化を無効にします。他の 3 つのオプションを指定した場合、コンパイラによる最適化が有効になります。これらの最適化の 1 つを指定するには、オプションのより詳細な説明で示されているように、アプリケーションの性質と構造を考慮する必要があります。一般に、-O1、-O2、および -O3 オプションは、次のように最適化を行います:

- -O1 -- コードのサイズと局所性
- -O2 -- コードの速度 (デフォルト)
- -O3 -- -O2 に加えて、さらに強力な最適化を有効にします

これらのオプションは、この後で説明されているように多少の違いがありますが、IA-32 アーキテクチャと Itanium® アーキテクチャでほぼ同様に動作します。

最適化レベルの設定

次の表は、-O0、-O1、-O2、-O3、および -fast オプションの効果を説明したものです。表は最初に IA-32 アーキテクチャと Itanium® アーキテクチャの両方で共通の特性を説明し、次に -On オプションの動作の詳細を (ある場合) アーキテクチャ別に説明します。

オプション	効果
-O0	最適化を無効にします。
-O1	好ましいコードサイズとコードの局所性に最適化します。ループのアンロールを無効にします。任意の分岐および実行時間がループ内のコードに支配されない、非常に大きなコードサイズのアプリケーションでパフォーマンスが向上します。ほとんどの場合は、-O1 よりも -O2 を推奨します。 **IA-32 システム:** コードサイズを減らす組込み関数のインライン化を無効にします。 **Itanium ベース・システム:** ソフトウェアのパイプライン化とグローバル・コード・スケジューリングを無効にします。
-O2, -O	デフォルトでは、このオプションがオンになっています。速度について最適化します。一般的に推奨される最適化レベルです。 **Itanium ベース・システム:** ソフトウェアのパイプライン化を有効にします。

-O3	-O2 オプションに、ループやメモリアクセス変換などの最適化項目を加えて最適化を実行します。-O3 の最適化はコードによっては -O2 の最適化よりも遅くなります。このオプションは、浮動小数点演算を多く使用するループや大きなデータセットを処理するループを含むアプリケーションに推奨します。 **IA-32 システム:** -ax{K|W|N|B|P} や -x{K|W|N|B|P} オプションと組み合わせると、コンパイラは -O2 よりも詳細にデータの依存性を分析します。そのためコンパイル時間が長くなる場合があります。
-fast	-fast オプションは、実行時のパフォーマンスを向上させる次のオプションを含めることで、プログラム全体にわたって実行速度を上げます: • -O3 (最高速で高レベルの最適化) • -ipo (複数ファイルにわたるプロシージャ間の最適化を有効にします) • -static (共有ライブラリとリンクしないようにします) -fast で設定されるオプションの 1 つを無効にするには、コマンドラインで -fast オプションの後に無効にするオプションを指定します。-fast で設定されるオプションは、バージョンによって変更になる場合があります。 特定のプロセッサ用に -fast の最適化を行うには、-x オプションのうちの 1 つを使用してください。 例: prompt>icpc -fast -xW source_file.cpp

最適化の範囲の制限

最適化の範囲を絞ったり、最適化を禁止したりする場合は、以下の各オプションを使います。

オプション	説明
-O0	最適化を無効にします。-fp オプションを有効にします。
-mp	浮動小数点演算の精度にいくらかの正と負の影響をもたらす最適化を制限し、宣言された精度のレベルを保ち浮動小数点演算が ANSI および IEEE* 標準に準拠するようにします。
-g	コマンドラインで -g とともに -O1、-O2、または -O3 が明示的に指定されていなければ、デフォルトの -O2 オプションをオフにして -O0 をデフォルトにし ます。
-nolib_inline	組込み関数のインライン展開を無効にします。

注

#pragma optimize を使用して、特定の関数のすべての最適化をオフにすることができます。次の例では、関数 foo() ですべての最適化がオフになります:

```
#pragma optimize("", off)
foo(){
...
}
```

#pragma optimize の有効な 2 番目の引数は "on" または "off" です。引数 "on" を使用すると、foo() 関数は、残りのプログラムと同じように最適化してコンパイルされます。コンパイラは、最初の引数の値を無視します。

浮動小数点の最適化

浮動小数点演算の精度

IA-32 および Itanium® ベース・システムで利用可能なオプション

-mp オプション

-mp オプションを指定すると、精度は宣言された水準が保たれます。また、浮動小数点演算の処理は、ANSI および IEEE 標準にほぼ準拠する結果となります。このオプションは、ほとんどのプログラムのパフォーマンスに不利に働きます。目的のアプリケーションにこのオプションが必要かどうかよくわからない場合は、このオプションを指定した場合と指定しない場合で実際にプログラムをコンパイルし、実行して、パフォーマンスと精度に対する効果を評価してみてください。-mp オプションを指定すると、プログラムのコンパイルに次の影響が生じます:

- 浮動小数点型として定義されたユーザ変数は、レジスタに割り当てられません。
- IA-32 システムで、式がスピル (レジスタからメモリへ移動) する場合、64 ビット (倍精度) ではなく、80 ビット (拡張精度) としてスピルされます。
- 浮動小数点演算の比較は、NaN (非数) の動作以外は IEEE 754 の仕様に準拠します。
- 演算は、コード内で指定したとおりに実行されます。例えば、除算が「逆数の乗算」に変更されることはありません。
- コンパイラは関連付けを変更せずに、指定した順序で浮動小数点演算を実行します。
- 浮動小数点値に対しては、定数の畳込みによる最適化は実行しません。定数の畳込みとは、1 を掛けたり、1 で割ったり、0 の足し引きをしたりといった計算を省くことです。定数の畳込みを実行しないので、例えば、0.0 の足し算についても記述したとおりに実行します。また、コンパイル時の浮動小数点演算も実行しません。これは、浮動小数点例外についてもその状態を変えないようにするためです。

- 浮動小数点演算は ANSI C に準拠します。float 型および double 型への代入を行うと、その精度は、80 ビット (拡張型) から 32 ビット (float) か 64 ビット (double) に丸められます。-mp を指定しなければ、精度が丸められずに変数が再使用される場合もあります。
- 関数のインライン展開を無効にする -nolib_inline オプションを設定します。

-mp1 オプション

浮動小数点の精度を高くするには、-mp1 オプションを使用します。-mp1 は、-mp よりも禁止される最適化処理が少なく、パフォーマンスに与える影響も小さくなります。

IA-32 のみのオプション

警告

デフォルトの精度制御方式または丸めモードを変更すると (例えば、-pc32 フラグの使用やユーザの介入によって)、いくつかの算術関数で返された結果に影響する場合があります。

-long_double オプション

long double 型のサイズを 80 ビットに変更するには、-long_double を使用します。コンパイラのデフォルトの long double 型は、サイズが double 型と同じ 64 ビットになっています。このオプションを使用すると、このオプションを使用しないでコンパイルした他のファイルと互換性のない部分やライブラリ・ルーチンへの呼び出し命令に整合しない部分がいくつか発生します。したがって、このオプションを指定してコンパイルするときは、long double 型の変数はファイル単体の中だけのローカル変数にすることをお勧めします。

-prec_div オプション

-xK や -xW などを指定して最適化する場合、インテル® C++ コンパイラは浮動小数点の除算を、分母の逆数による乗算に変更します。計算速度を上げるため、例えば A/B を A×(1/B) として計算します。ただし、B が 2^{126} より大きい場合は 1/B の値が 0 になります。1/B の値を維持しなければならない場合は、-prec_div を使用して、浮動小数点の除算を乗算に変換する最適化処理を禁止してください。-prec_div を指定すると、精度は上がりますが、若干パフォーマンスが落ちます。

-pc*n* オプション

浮動小数点の仮数の精度を制御するには、-pc*n* オプションを使用します。浮動小数点アルゴリズムの中には、浮動小数点値の仮数部または小数部の精度に影響を受

けやすいものがあります。例えば、除算や平方根の計算のように反復処理が多いものは、-pc*n* オプションを使用して精度を下げると計算が速くなる場合があります。*n* は次のいずれかの値に設定すると、仮数はそれぞれ示したビット数に丸められます。

- -pc32: 24 ビット（単精度）-- 上記の注意を参照してください。
- -pc64: 53 ビット（単精度）
- -pc80: 64 ビット（単精度）-- デフォルト

n のデフォルト値は倍精度を示す 80 です。このオプションを指定すると、全面的な最適化を実行できます。このオプションで影響されるのは浮動小数点値の小数部だけなので、-Op オプションとは異なり、パフォーマンスに悪影響は与えません。指数の部分は影響を受けません。-pc*n* オプションを指定すると、main() 関数がコンパイルされるときの浮動小数点の精度の制御方式が変わります。-pc*n* を使用するプログラムはエントリポイントとして main() を使用しなければならず、main() を含むファイルは -pc*n*. を指定してコンパイルしなければなりません。

-rcd オプション

浮動小数点から整数への変換を必要とするコードのパフォーマンスを改善するには、-rcd オプションを使用します。これは、丸めモードの変更を制御して最適化を行います。デフォルトでは、浮動小数点の丸めモードは「最近値への丸め」となっています。つまり、各値は浮動小数点の計算中に丸められます。ただし、C 言語の場合は、整数への変換時に浮動小数点値を切り捨てる必要があります。これを行うには、コンパイラは、浮動小数点から整数へ変換する前に丸めモードを「切り捨て」方式に変更し、変換が終わったらまた元に戻すという処理をしなければなりません。-rcd オプションを指定すると、浮動小数点から整数への変換をはじめとして浮動小数点の計算すべてについて、丸めモードは「切り捨て」方式に変更できなくなります。このオプションを指定するとパフォーマンスは改善されるかもしれませんが、浮動小数点から整数への変換処理については C 言語のセマンティクスに適合しなくなります。

-fp_port オプション

-fp_port オプションは、代入と型キャストの際に浮動小数点の結果を丸めます。速度に多少影響します。

-fpstkchk オプション

関数呼び出しが浮動小数点値を返す場合、戻り値は FP スタックの一番上に配置されます。戻り値が未使用の場合、コンパイラは正しい状態で FP スタックを維持するためにスタックから値をポップします。しかし、アプリケーションが関数のプロトタイプを省略するか、関数のプロトタイプを正しく行わない場合、戻り値はスタックに残されます。この結果、FP スタックが最終的にオーバーフローすることがあります。

一般に、FP スタックがオーバーフローすると、NaN 値が FP 計算で用いられるため、プログラムの結果は異なります。また、オーバーフロー・ポイントが実際の問題の箇所とは異なっていることもあります。`-fpchkstk` オプションは、正しくない呼び出しが発生した直後に問題があるコードを知らせるので、これらの問題をより簡単に見つけることができます。

Itanium® ベース・システムに使用する浮動小数点演算オプション

次のオプションは、Itanium® ベース・システムにおける浮動小数点計算用にコンパイラの最適化を制御します:

- `-ftz[-]`
- `-IPF_fma[-]`
- `-IPF_fp_speculation`*mode*
- `-IPF_flt_eval_method0`
- `-IPF_fltacc[-]` (デフォルト: `-IPF_fltacc-`)

デノーマル結果をゼロにフラッシュする

デノーマル結果をゼロにフラッシュするには、`-ftz-` オプションを使用します。

浮動小数点積和/積差演算の縮約

`-IPF_fma[-]` は、浮動小数点積和/積差演算の 1 つの演算への縮約を有効 [無効] にします。`-mp` が指定されない限り、コンパイラは可能な限りこれらの演算を縮約します。`-mp` オプションは、縮約を無効にします。デフォルトのコンパイラ動作を無効にするには、`-IPF_fma` および `-IPF_fma-` を使用します。例えば、`-mp` と `-IPF_fma` をともに使用すると、コンパイラは縮約演算を有効にします (Itanium ベース・システムのみ):

```
prompt>icpc -mp -IPF_fma prog.cpp
```

FP スペキュレーション

`-IPF_fp_speculation`*mode* は、次のいずれかのモード (*mode*) で、浮動小数点演算のスペキュレーションを設定します:

- `fast`: 浮動小数点演算のスペキュレーションを有効にします。
- `safe`: 安全な場合のみ浮動小数点演算のスペキュレーションを有効にします。
- `strict`: 浮動小数点演算のスペキュレーションを無効にします。
- `off`: 浮動小数点演算のスペキュレーションを無効にします。

FP 演算の評価

-IPF_flt_eval_method0 は、プログラムで宣言された変数型により指定された精度で、浮動小数点式の演算子に関する表現を評価するようにコンパイラに指示します。

FP 結果の精度制御

-IPF_fltacc[-] は、浮動小数点の精度に影響を与える最適化を有効 [無効] にします。デフォルト (-IPF_fltacc-) では、コンパイラは浮動小数点の精度を低下する最適化を適用する場合があります。浮動小数点の精度を高めるために -IPF_fltacc または -mp を使用できますが、いくつかの最適化を無効にしなくてはなりません。

特定のプロセッサの最適化

プロセッサの最適化

IA-32 にのみ該当するプロセッサの最適化

-tpp{5|6|7} オプションは、アプリケーションのパフォーマンスを特定のインテル・プロセッサ向けに最適化します。最適化を行ったアプリケーションは、下記の表で記述されているプロセッサ上でも実行することができます。インテル® C++ コンパイラには、gcc* 互換バージョンの -tpp オプションが含まれています。これらのオプションは、**gcc* バージョン**欄に示します。

オプション	gcc* バージョン	最適化の対象
-tpp5	-mcpu=pentium	インテル® Pentium® プロセッサ
-tpp6	-mcpu=pentiumpro	インテル Pentium Pro プロセッサ、インテル Pentium II プロセッサ、およびインテル Pentium III プロセッサ
-tpp7	-mcpu=pentium4	インテル Pentium 4 プロセッサ、インテル Pentium M プロセッサ、HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応)

注

-tpp7 オプションはデフォルトでオンです。

例

次の呼び出しはすべて、Pentium 4 プロセッサ向けにアプリケーションを最適化します。これらの結果は Pentium、Pentium Pro、Pentium II および Pentium III プロセッサ上でも実行することができます。

```
prompt>icpc prog.cpp

prompt>icpc -tpp7 prog.cpp

prompt>icpc -mcpu=pentium4 prog.cpp
```

Itanium® ベース・システムにのみ該当するプロセッサの最適化

`-tpp{1|2}` オプションは、アプリケーションのパフォーマンスを特定のインテル Itanium プロセッサ向けに最適化します。最適化を行ったアプリケーションは、下記の表で記述されているプロセッサ上でも実行することができます。インテル C++ コンパイラには、gcc* 互換バージョンの `-tpp` オプションが含まれています。これらのオプションは、**gcc* バージョン**欄に示します。

オプション	gcc* バージョン	最適化の対象
`-tpp1`	`-mcpu=itanium`	Itanium プロセッサ
`-tpp2`	`-mcpu=itanium2`	Itanium 2 プロセッサ

注

`-tpp2` オプションはデフォルトでオンです。

例

次の呼び出しはすべて、Itanium 2 プロセッサ向けにアプリケーションを最適化します。これらの結果は Itanium プロセッサ上でも実行することができます。

```
prompt>icpc prog.cpp

prompt>icpc -tpp2 prog.cpp

prompt>icpc -mcpu=itanium2 prog.cpp
```

プロセッサ固有の最適化 (IA-32 のみ)

-x{K|W|N|B|P} オプションは、プログラムを特定のインテル® プロセッサ向けに最適化します。生成されるコードには、他のプロセッサでサポートされない機能の無条件の使用が含まれることがあります。

オプション	最適化の対象
-xK	インテル® Pentium® III プロセッサおよび互換性のあるインテル・プロセッサ。
-xW	インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ。
-xN	インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ。関数 main() がこのオプションを使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラー・メッセージを出力します。このオプションは、インテル・プロセッサ固有の最適化に加えて新しい最適化も有効にします。
-xB	インテル Pentium M プロセッサおよび互換性のあるインテル・プロセッサ。関数 main() がこのオプションを使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラー・メッセージを出力します。このオプションは、インテル・プロセッサ固有の最適化に加えて新しい最適化も有効にします。
-xP	HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応)。関数 main() がこのオプションを使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラー・メッセージを出力します。このオプションは、インテル・プロセッサ固有の最適化に加えて新しい最適化も有効にします。

インテル以外から提供されている x86 プロセッサ上でプログラムを実行する場合は、-x{K|W|N|B|P} オプションを指定しないでください。

例

次の呼び出しは、インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ向けに prog.cpp をコンパイルします。生成されるバイナリは、Pentium プロセッサ、Pentium Pro プロセッサ、Pentium II プロセッサ、Pentium Pentium III プロセッサ、MMX® テクノロジ Pentium プロセッサ、およびインテル以外から提供されている x86 プロセッサ上では正常に実行されない可能性があります。

```
prompt>icpc -xW prog.cpp
```

警告

-x{K|W|N|B|P} を使用してコンパイルされたプログラムを互換性のないプロセッサ上で実行すると、不正な命令例外や他の予期しない動作が発生することがあります。-xN、-xB、または -xP を使用してコンパイルされたプログラムを、サポートされていないプロセッサ (上の表を参照) 上で実行すると、次のランタイム・エラーが表示されます:

```
Fatal Error : This program was not built to run on the
processor in your system.
```

特定のプロセッサに合わせて自動的に最適化する (IA-32 のみ)

-ax{K|W|N|B|P} オプションは、指定されたインテル® プロセッサ固有の機能を利用する関数バージョンを生成できるかどうかを確認するようにコンパイラに命令します。固有の関数バージョンを生成できることが判明した場合、コンパイラはそれがパフォーマンスの向上につながるかどうかをチェックします。パフォーマンスの向上につながると判明した場合、コンパイラはプロセッサ固有の関数バージョンと汎用バージョンの両方を生成します。汎用バージョンはすべての IA-32 プロセッサ上で実行できます。

プログラムの実行時に、使用されているインテル・プロセッサに応じて、この 2 つのバージョンのどちらを実行するかが選択されます。この方法により、プログラムは従来の IA-32 プロセッサとの互換性を保ちながら、最新のインテル・プロセッサ上でパフォーマンスを大幅に向上することができます。

ただし、-ax{K|W|N|B|P} を使用した場合には、次のような欠点があります:

- プロセッサ固有のコード・バージョンと汎用のコード・バージョンの両方が含まれるため、コンパイルされたバイナリのサイズが大きくなります。
- 使用するコードを決定するランタイム・チェックによって、パフォーマンスに影響があります。

注

このオプションでコンパイルするアプリケーションは、すべての IA-32 プロセッサ上で実行できます。-x と -ax オプションの両方を指定した場合、-x オプションで指定されたプロセッサの種類と互換性のあるプロセッサでのみ実行できる汎用コードが生成されます。

オプション	最適化の対象となるプロセッサ
-axK	インテル® Pentium® III プロセッサおよび互換性のあるインテル・プロセッサ。
-axW	インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ。
-axN	インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ。このオプションは、インテル・プロセッサ固有の最適化に加えて新しい最適化も有効にします。
-axB	インテル Pentium M プロセッサおよび互換性のあるインテル・プロセッサ。このオプションは、インテル・プロセッサ固有の最適化に加えて新しい最適化も有効にします。
-axP	HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) および互換性のあるインテ

	ル・プロセッサ。このオプションは、インテル・プロセッサ固有の最適化に加えて新しい最適化も有効にします。

例

下記のコンパイルは、次のような 1 つの実行ファイルを生成します:

- すべての IA-32 プロセッサで使用できる汎用バージョン
- パフォーマンスが向上する場合、Pentium III プロセッサ用に最適化されたバージョン
- パフォーマンスが向上する場合、Pentium 4 プロセッサ用に最適化されたバージョン

```
prompt>icpc -axKW prog.cpp
```

手動 CPU ディスパッチ (IA-32 のみ)

`__declspec(cpu_specific)` および `__declspec(cpu_dispatch)` をコードに使用して、アプリケーションが実行されるプロセッサに特有の命令を生成し、また他のプロセッサ上でも正常に実行できるようにします。

注

手動 CPU ディスパッチは、インテル® Itanium® プロセッサの認識には使用できません。これらの拡張属性の構文は以下のとおりです:

- `cpu_specific(`*`cpuid`*`)`
- `cpu_dispatch(`*`cpuid-list`*`)`

`cpuid` および *`cpuid-list`* の値は、下の表のとおりです:

プロセッサ	***`cpuid`* の値**
インテル以外から提供されている x86 プロセッサ	`generic`
インテル® Pentium® プロセッサ	`pentium`
インテル MMX® テクノロジ Pentium プロセッサ	`pentium_mmx`
インテル Pentium Pro プロセッサ	`pentium_pro`
インテル Pentium II プロセッサ	`pentium_ii`
インテル Pentium III プロセッサ	`pentium_iii`
インテル Pentium III (xmm レジスタを除く)	`pentium_iii_no_xmm_regs`

インテル Pentium 4 プロセッサ	pentium_4
インテル Pentium M プロセッサ	pentium_m
HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応)	future_cpu_10

***cpuid-list* の値**
cpuid
cpuid-list, cpuid

属性は大文字と小文字の区別はありません。__declspec(cpu_dispatch) を宣言した関数本体は空でなければなりません。そして、これはスタブ (empty-bodied 関数) と呼ばれます。

次のガイドラインに従って、自動プロセッサ・ディスパッチ・サポートを使用してください:

1. **cpu_dispatch のスタブは、cpu_specific で定義された cpuid を持っていなければなりません。**
   関数 f の cpu_dispatch スタブが cpuid p を持つ場合、cpuid p を持つ f の cpu_specific 定義をプログラムのどこかに記述する必要があります。そうでない場合、未解決外部エラーとなります。cpu_specific 関数が static で宣言されない限り、cpu_specific 関数の定義は、対応する cpu_dispatch スタブと同じ交換単位に記述する必要はありません。インライン属性は、すべての cpu_specific および cpu_dispatch 関数で、無効です。
2. **cpu_specific 関数のスタブを持つ必要があります。**
   関数 f を __declspec(cpu_specific(p)) として定義した場合、cpu_dispatch スタブもまたプログラム内で f に記述する必要があります。そして、p は、そのスタブの cpuid-list になければなりません。そうでない場合、cpu_specific 定義は呼び出されず、またエラー条件も生成しません。
3. **コマンドライン設定の上書き**
   cpu_dispatch スタブをコンパイルすると、その本体は、プログラムが実行されているプロセッサを決定するコードに置き換えられます。そして、cpuid-list に定義された利用可能な "ベストの" cpu_specific 関数をディスパッチします。cpu_specific 関数は、コマンドライン・オプションの設定にかかわらず、指定されたプロセッサに応じて最適化します。

プロセッサ・ディスパッチ例

これらの関数の使用方法の例を次に示します:

```
#include <mmintrin.h>
/* Pentium processor function does not use intrinsics to add two
arrays. */

  declspec(cpu specific(pentium))
void array_sum(int *r, int *a, int *b,size_t l)
{
   for (; length > 0; l--)
   *result++ = *a++ + *b++;
}

/* Implementation for a Pentium processor with MMX technology uses
an MMX instruction intrinsic to add four elements simultaneously. */

  declspec(cpu specific(pentium MMX))
void array sum(int *r,int const *a, int *b, size t l)
{
     m64 *mmx result = (  m64 *)result;
     m64 const *mmx a = (  m64 const *)a;
     m64 const *mmx b = (  m64 const *)b;

   for (; length > 3; length -= 4)
   *mmx result++ =  mm add pi16(*mmx a++, *mmx b++);

   /* The following code, which takes care of excess elements, is not
   needed if the array sizes passed are known to be multiples of
four. */

   result = (unsigned short *)mmx r;
   a = (unsigned short const *)mmx a;
   b = (unsigned short const *)mmx_b;

   for (; length > 0; l--)
   *result++ = *a++ + *b++;
}

__declspec(cpu_dispatch(pentium, pentium_MMX))
void array sum (int *r,int const *a, int *b, size t l) )

{

/* Empty function body informs the compiler to generate the
CPU-dispatch function listed in the cpu_dispatch clause. */

}
```

プロセッサ固有のランタイム・チェック (IA-32 システム)

インテル® C++ コンパイラの最適化はランタイム時に効果を発揮します。IA-32 システムでは、コンパイラはプログラム中に記述されているランタイム・チェックを実行するコード・セグメントを挿入して、プロセッサ固有の最適化を強化します。

-xN、-xB、または -xP によるサポートされたプロセッサのチェック

実行エラーを防ぐために、コンパイラはプログラム中に正しいプロセッサが使用されていることをチェックするコードを挿入します。 -xN、-xB、または -xP オプションを使用

してコンパイルされたプログラムは、インテル® Pentium® 4 プロセッサ、インテル Pentium M プロセッサ、HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応)、またはこれらと互換性のあるインテル・プロセッサ上で実行されているかどうか、ランタイム時にチェックされます。プログラムがこれらのいずれかのプロセッサ上で実行されていない場合、プログラムはエラーを出力して終了します。

例

プログラム `prog.cpp` を HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) 用に最適化するには、次のコマンドを使用します:

```
prompt>icpc -xP prog.cpp
```

生成されたプログラムがインテル Pentium III プロセッサやインテル Pentium 4 プロセッサなどの、HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) をサポートしていないプロセッサ上で実行されると、プログラムは終了します。

プログラムを複数の IA-32 プロセッサ上で実行する場合は、プロセッサ固有の最適化を行う `-x{ }` オプションではなく、プロセッサ固有のコードと汎用コードを生成する `-ax{ }` オプションを使用してください。

FTZ と DAZ フラグの設定

これまで、IA-32 プロセッサでは FTZ (ゼロ・フラッシュ) と DAZ (denormals are zeros) フラグはデフォルトでオフになっていました。しかし、これらのフラグをオンにすると、IEEE 準拠ではなくなりますが、最新の IA-32 プロセッサ上で実行される漸次アンダーフロー・モードでデノーマルな浮動小数点値が使用され、プログラムのパフォーマンスが大幅に向上します。このため、インテル Pentium III プロセッサ、インテル Pentium 4 プロセッサ、インテル Pentium M プロセッサ、HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応)、および互換性のある IA-32 プロセッサの場合、コンパイラはデフォルトでこれらのフラグをオンにするように変更されました。コンパイラは、プログラムがこれらのインテル・プロセッサのいずれかで実行されていることを確認するため、プロセッサのランタイム・チェックを実行するコードをプログラム中に挿入します。

例

- インテル Pentium III プロセッサ上でプログラムを実行すると、FTZ は有効になりますが、DAZ は有効になりません。
- インテル Pentium M プロセッサまたは HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) 上でプログラムを実行すると、FTZ と DAZ の両方が有効になります。

これらのフラグは、サポートが確認されたインテル・プロセッサによってのみオンにされます。

インテル以外のプロセッサの場合、次のマクロを使用してフラグを手動で設定することができます:

FTZ を有効にする: `_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON)`

DAZ を有効にする:
`_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON)`

これらのマクロのプロトタイプは、`xmmintrin.h` (FTZ) および `pmmintrin.h` (DAZ) にあります。

プロシージャ間の最適化

概要: プロシージャ間の最適化

`-ip` と `-ipo` を使用してプロシージャ間の最適化 (IPO) を有効にすれば、コンパイラにコードを分析させて、次の表に示す最適化をすればどの部分に効果が出るかを調べられます。

IA-32 および Itanium® ベース・アプリケーション

最適化項目	影響を受ける部分
関数のインライン展開	呼び出し、ジャンプ、分岐、ループ
プロシージャ間での定数伝播	引数、グローバル変数、戻り値
モジュール・レベルでの静的変数の監視	現状以上の最適化、ループ不変コード
不要コードの排除	コードのサイズ
関数特性の伝播	呼び出し命令の削除と呼び出し命令の移動。さらに、例外が発生してもリターンされない関数、アライメントが必要なスタック、引数のアライメントに関する部分も影響を受けます。
マルチファイルの最適化	`-ip` と同じ部分に影響を与えますが、複数のファイル全体にわたって最適化を行います。

IA-32 アプリケーションのみ

最適化項目	影響を受ける部分
レジスタ内での引数の受け渡し	呼び出し、レジスタの使用

関数のインライン展開は、プロシージャ間の最適化機構によって実行する主な最適化機能のうちの 1 つです。コンパイラは、頻繁に実行する関数呼び出しが存在していると判断した場合、その呼び出し命令を、当該関数自体のコードに置き換えることがあります (呼び出しのインライン展開)。

`-ip` オプションを指定すると、現在のソースファイル内で定義している関数内での呼び出し命令について関数のインライン展開を実行します。ただし、`-ipo` オプションを使用してマルチファイル IPO を指定すると、別々のファイル内で定義している関数内での呼び出し命令について関数のインライン展開が実行されます。このため、`-ipo` を指定する場合、アプリケーション全体または複数の関連するソースファイルを一緒にコンパイルすることは重要です。

IPO による最適化は、デフォルトでは実行されません。

プロシージャ間の最適化のオプション

オプション	説明
`-ip`	単一ファイルのコンパイルでプロシージャ間の最適化を有効にします。
`-ip_no_inlining`	`-ip` (プロシージャ間の最適化) によるインライン展開を無効にしますが、他のプロシージャ間の最適化には影響しません。
`-ipo`	複数ファイルにわたるプロシージャ間の最適化を有効にします。
`-ipo_c`	後のリンク段階で使用できるマルチファイル・オブジェクト・ファイルを生成します。
`-ipo_obj`	`-ipo` とともに指定すると、実際のオブジェクト・ファイルが強制的に生成されます。
`-ipo_S`	後のリンク段階で使用できるマルチファイル・アセンブリ・ファイル (`ipo_out.asm`) を生成します。
`-inline_debug_info`	コール・サイトのソース・ポジションをインライン化コードに割り当てる代わりに、インライン化コードのソース・ポジションを保存します。
`-nolib_inline`	標準ライブラリ関数のインライン展開を禁止します。

-ip または -ipo と -Qoption 指示子の使用

特定のインライン展開およびループ最適化を選択するには、適切なキーワードを用いて `-Qoption` を使用します。このオプションを入力する場合は、次の例に示すように、`-ip` または `-ipo` を指定しなければなりません:

```
prompt>icpc -ip -Qoption,tool,opts
```

tool は C++ (c)で、*opts* は -Qoption 指示子です (下記参照)。

-Qoption 指示子

-Qoption なしで -ip または -ipo を指定すると、コンパイラは次のことを行います:

- 関数のインライン展開
- 定数の引数伝播
- レジスタ内の引数の受け渡し
- 関数レベルでの静的変数の監視

次の -Qoption 指示子を使用して、プロシージャ間の最適化を設定できます。これを有効にするには、例に示すように、-Qoption オプションとともに -ip または -ipo を入力しなければなりません:

prompt>**icpc -ip -Qoption,***c,ip_specifier*

ip_specifier は次の表で説明される指定子のいずれかです:

指示子	説明
-ip_args_in_regs=0	レジスタ内での引数の受け渡しを無効にするオプションです。デフォルトでは、ローカルに呼び出された外部関数はレジスタ内で引数の受け渡しができます。通常、スタティック関数のみはレジスタ内での引数の受け渡しができますが、それには条件が 2 つあり、1 つはその関数のアドレスが取得されないこと、もう 1 つは個数の定まらない引数をその関数が使用しないことです。
-ip_ninl_max_stats=n	インライン展開する関数 1 個について、中間言語の文を最大何個まで許容させるかを設定するオプションです。n は正の整数です。通常は、中間言語の文の個数は、ソース言語の文の実際の個数を超えます。n のデフォルト値は 230 です。コンパイラは、ユーザのインライン関数用に、制限をより大きくします。
-ip_ninl_min_stats=n	インライン展開する関数 1 個について、中間言語の文を最小何個まで許容させるかを設定するオプションです。n は正の整数です。ip_ninl_min_stats のデフォルト値は次のとおりです: • IA-32 コンパイラ: ip_ninl_min_stats = 7 • Itanium® コンパイラ: ip_ninl_min_stats = 15
-ip_ninl_max_total_stats=n	インライン化のために、中間言語文で、関数のサイズに展

	開できる最大数を設定します。n は正の整数でデフォルト値は 2000 です。

次のコマンドは source.cpp でプロシージャおよびプロシージャ間の最適化を有効にし、各関数に対して中間言語の文の数の最大増加数を 5 に設定します。

```
prompt>icpc -ip -Qoption,c,-ip_ninl_max_stats=5 source.cpp
```

関数のインライン展開

ユーザ関数のインライン展開の制御

下の表に示すオプションを使用して、関数のインライン展開を制御することができます:

-ip_no_inlining	このオプションは、-ip オプションも指定されている場合にのみ使用できます。この場合、-ip_no_inlining オプションは、-ip によるプロシージャ間の最適化の結果から生じるインライン展開を無効にしますが、他のプロシージャ間の最適化には影響しません。
-ip_no_pinlining	部分的なインライン化を無効にします。-ip または -ipo を指定しても使用できます。

関数のインライン展開の条件

以上の諸条件を満たすと、コンパイラはどのルーチンをインライン展開すればプログラムのパフォーマンスに最も寄与するかを調べて選び出します。プロファイルに基づく最適化 (-prof_use) を使用するかどうかによって、コンパイラが使用するヒューリスティックは異なります。-ip または -ipo でプロファイルに基づく最適化を使用する場合、コンパイラは次のヒューリスティックを使用します:

- デフォルト・ヒューリスティックでは、プログラムに対して収集されたプロファイル情報を基に、最も頻繁に実行される呼び出しサイトに重点がおかれます。
- デフォルトでは、コンパイラは 230 以上の中間文では、関数のインライン化は行いません。この値は、オプション -Qoption,c,-ip_ninl_max_stats=*new_value* を使用して変更できます。注: inline または __inline のようにユーザに宣言される関数には、より高い制限があります。
- 通常のデフォルト・ヒューリスティックは、直接再帰 (direct recursion) を検出すると停止します。
- デフォルト・ヒューリスティックを使用すると、インライン化の最低条件を満たしている非常に小さな関数は必ずインライン化します。

- Itanium® ベース・アプリケーションのデフォルト値: `ip_ninl_min_stats=15`。
- IA-32 アプリケーションのデフォルト値: `ip_ninl_min_stats = 7`。この上限は、オプション `-Qoption,c,-ip_ninl_max_stats=`*`new_value`* を使用して変更できます。

`-ip` または `-ipo` でプロファイルに基づく最適化を使用しない場合、コンパイラは次のヒューリスティックを使用します:

- インライン展開が最終的なプログラムのサイズを増加しない場合、関数をインライン展開します。
- 関数が `inline` または `__inline` キーワード付きで宣言された場合、関数をインライン展開します。

マルチファイル IPO

概要: マルチファイル IPO

マルチファイル IPO とは、最適化の余地がないかどうかについての情報を、マルチファイル・プログラムの個々のプログラム・モジュールから得るものです。コンパイラはこの情報を使用して、複数のモジュール全体にわたって最適化を行います。

プログラムを構築する工程はコンパイルとリンケージに分かれています。マルチファイル IPO の処理内容は、実行するのがコンパイルかリンケージかによって違ってきます。

コンパイル・フェーズ

各ソースファイルがコンパイルされるたびに、マルチファイル IPO はソースコードの中間表現 (IR) を擬似オブジェクト・ファイルに格納します。このオブジェクト・ファイルには、最適化に使うサマリ情報が含まれています。

特に指定しない限り、コンパイラは、マルチファイル IPO のコンパイル・フェーズの最中に、擬似オブジェクト・ファイルをいくつか生成します。実際のオブジェクト・ファイルの代わりに擬似オブジェクト・ファイルを生成すると、マルチファイル IPO のコンパイル・フェーズに要する時間が短くなります。各擬似オブジェクト・ファイルは、対応するソースファイルの IR を含みますが、実コードもデータも含みません。この擬似オブジェクトは、`-ipo` オプションを使用するか `xild` ツールを使用してコンパイラにリンクする必要があります。

注

-ipo か xild を使用して擬似オブジェクト・ファイルをリンクしないとリンケージ・エラーが発生します。場合によっては、擬似オブジェクト・ファイルを使用できない場合があります。詳細については、「実際のオブジェクト・ファイルの生成」を参照してください。

リンケージ・フェーズ

-ipo を指定すると、コンパイラはリンカの直前に起動されます。IR を含むオブジェクト・ファイルすべてを対象にしてマルチファイル IPO が実行されます。

注

スタティック・ライブラリ (.a ファイル) についてはマルチファイル IPO は利用できません。詳細については、「実際のオブジェクト・ファイルの生成」を参照してください。

-ipo を指定すると、ドライバとコンパイラで自動的にプログラム全体を検出できるようになります。プログラム全体を検出すれば、プロシージャ間の定数伝播、スタックフレームのアライメント、データ・レイアウト、共通ブロックのパディングといった最適化をもっと効率よく実行しますが、削除される不要な関数の数も増えます。このオプションは安全です。

実際のオブジェクト・ファイルの生成

状況によっては、-ipo を使用して実際のオブジェクト・ファイルを生成する必要があります。IPO を行う場合、擬似オブジェクト・ファイルではなく実際のオブジェクト・ファイルを強制的に生成するために、ipo_obj オプションと -ipo オプションを組み合わせて使用します。

次の場合は、-ipo_obj を使用する必要があります:

- -ipo を指定したコンパイル・フェーズで生成したオブジェクトを、xild ツールか xild -lib ツールでスタティック・ライブラリに格納する場合。スタティック・ライブラリについてはマルチファイル IPO を利用できないため、スタティック・ライブラリはすべてリンカに渡されます。擬似オブジェクト・ファイルを含むスタティック・ライブラリにリンクするとリンケージ・エラーが発生します。-ipo_obj を指定すると、スタティック・ライブラリの中で使用できるオブジェクト・ファイルが生成されます。
- 一方、xiar または xild -lib を使用して作成したスタティック・ライブラリは、普通のライブラリとして機能します。

- -ipo を指定したコンパイル・フェーズで生成したオブジェクトを、-ipo オプションと xild を使用しないでリンクする場合。
- -ipo を指定してコンパイルを行っている最中に、-S を用いてソースファイルごとにアセンブリ・リストを生成する場合。-ipo_obj ではなく -S と一緒に -ipo を使用すると、警告メッセージが出て、コンパイルしたソースファイルごとに空のアセンブリ・ファイルが生成されます。

バージョン番号による IL ファイルの管理

IPO コンパイルは、コンパイル・フェーズとリンクフェーズの 2 つに分けて行われます。コンパイル・フェーズでは、コンパイラは中間言語 (IL) バージョンのコードを含むファイルを生成します。リンクフェーズでは、コンパイラは IL を読み込んでコンパイルを完了し、実際のオブジェクト・ファイルまたは実行ファイルを生成します。

一般に、コンパイラのバージョンが異なると、異なる定義に基づいて IL が生成されるため、IL は互換がない場合があります。インテル® C++ コンパイラは各コンパイラの IL 定義に独自のバージョン番号を割り当てます。コンパイラがバージョン番号が異なるファイルの IL を読み込もうとすると、コンパイルは続行されますが、IL は破棄され、コンパイルでは使用されません。その際、コンパイラは非互換の IL に関する警告を表示します。

オブジェクトとライブラリ中の IL: さらなる最適化

インテル・コンパイラが生成した IL はオブジェクト・ファイルの特別なセクションに格納されます。その後、オブジェクト・ファイルに格納された IL はライブラリに配置されます。このライブラリが、その IL を生成したものと同じコンパイラで起動されて IPO コンパイルに使用された場合、コンパイラはライブラリから IL を抽出してプログラムの最適化に使用することができます。

マルチファイル IPO 実行ファイルの作成

ここでは、IA-32 と Itanium® ベースのシステムを対象としたコンパイルでマルチファイル IPO 実行ファイルを作成する方法を説明します。

-ipo を使用してソースモジュールのコンパイルとリンクを別々に行う場合:

1. 次のように -ipo を使用してコンパイルします:
   prompt>**icpc -ipo -c a.cpp b.cpp c.cpp**
2. .o ファイルの生成後にコンパイルを中止するときは -c オプションを使用してください。各オブジェクト・ファイルは、対応するソースファイルの IR を含みます。上記の結果を使用して、プロシージャ間の最適化を行います:
   prompt>**icpc -ipo a.o b.o c.o**

マルチファイル IPO は IR を含むモジュールにだけ適用され、IR を含まないオブジェクト・ファイルはリンク段階に渡されます。効率を上げたいときは、次のように手順 1 と 2 を組み合わせてください:

```
prompt>icpc -ipo a.cpp b.cpp c.cpp
```

プロファイル情報を含むマルチファイル IPO を使用すると、さらに最適化を図ることができます。「プロファイルに基づく最適化: 具体例」を参照してください。

xild を使用したマルチファイル IPO 実行ファイルの作成

インテル・リンカ `xild` は次のように動作します:

- `IR` を含むオブジェクトが見つかった場合、インテル® コンパイラを起動してマルチファイル IPO を実行します
- GNU リンカ `ld` を起動してアプリケーションをリンクします

`xild` のコマンドライン構文は次のとおりです:

```
prompt>xild [<options>] <LINK_commandline>
```

各アイテムの意味は次のとおりです:

- `[<options>]` (オプション) には、あらゆる `gcc` リンカ・オプション、または `xild` でのみサポートされるオプションを含めることができます。
- `<LINK_commandline>` は、`ld` への有効な引数のセットを含むリンカ・コマンドラインです。

マルチファイル IPO 実行ファイルを `ipo_file` に格納するには、オプション `-o`*`filename`* を使用します。
例:

```
prompt>xild oipo_file a.o b.o c.o
```

`xild` は、IR を含むオブジェクトの IPO を実行するためにインテル・コンパイラを呼び出し、リンクされるオブジェクトの新しい一覧を作成します。そして、`xild` は `ld` を呼び出して新しいリストで指定されたオブジェクト・ファイルにリンクし、`-o`*`filename`* オプションで指定された `ipo_file` 実行ファイルを生成します。

注

-ipo オプションを使用した場合、コマンドラインにオブジェクト・ファイルとリンカ引数を複数入力すると、その並び順が変わってしまうことがあります。したがって、コマンドラインに引数を複数入力する場合は、その順番を崩してはならないプログラムに対して -ipo を使用すると、プログラムが誤動作することがあります。

使用規則

次のような場合、インテル・リンカ xild を使用して、アプリケーションをリンクする必要があります:

- ソースファイルがマルチファイル IPO を有効にしてコンパイルされた場合。マルチファイル IPO が -ipo コマンドライン・オプションを指定して有効にされた場合。
- ld を起動してアプリケーションをリンクする場合。

xild のオプション

xild がサポートしている追加オプションは、マルチファイル IPO の結果を検証するために使用することができます。次の表で、これらのオプションを説明します。

オプション	説明
-ipo_o[file.s]	マルチファイル IPO コンパイルのアセンブリ・ファイルを生成します。リストファイルの名前またはファイルを配置するディレクトリ (バックスラッシュ付き) を指定します。デフォルトのリストファイル名は、ipo_out.s です。
-ipo_o[file.o]	マルチファイル IPO コンパイルのオブジェクト・ファイルを生成します。オブジェクト・ファイルの名前またはファイルを配置するディレクトリ (バックスラッシュ付き) を指定します。デフォルトのオブジェクト・ファイル名は、ipo_out.o です。
-ipo_fcode-asm	アセンブリ・ファイルにコードバイトを追加します。
-ipo_fsource-asm	アセンブリ・ファイルに高水準言語のソースコードを追加します。
-ipo_fverbose-asm, -ipo_fnoverbose-asm	バージョンと xild のアセンブリ・ファイルで使用されるオプションを含むコメントの挿入を有効または無効にします。

IPO オブジェクトからのライブラリの作成

通常、ライブラリは ar などのライブラリ・マネージャを使用して作成されます。ライブラリ・マネージャは、オブジェクト・リストを読み取り、そのオブジェクトを、次のリンク段階で使用するライブラリに挿入します。

```
prompt>xiar cru user.a a.o b.o
```

a.o と b.o を含むライブラリ user.a が作成されます。

ただし、-ipo -c を使用してオブジェクトが作成された場合、オブジェクトは有効なオブジェクトではなく、そのオブジェクト・ファイルの中間表現 (IR) だけを含みます。
例:

```
prompt>icpc -ipo -c a.cpp b.cpp
```

リンク時のコンパイルに使用される IR だけを含んだ a.o と b.o が作成されます。ライブラリ・マネージャでは、このオブジェクトをライブラリに挿入できません。

この場合はライブラリ・ドライバ xild -ar を使用しなければなりません。このドライバは、オブジェクト・ファイルに保存している IR 上にコンパイラを呼び出し、ライブラリに挿入できる有効なオブジェクトを生成します。

```
prompt>xild -lib cru user.a a.o b.o
```

「xild を使用したマルチファイル IPO 実行ファイルの作成」を参照してください。

マルチファイル IPO の効果の分析

-ipo_c と -ipo_S オプションは、マルチファイル IPO の効果を分析する場合や、完成したプログラムになる前の複数のモジュール間でマルチファイル IPO の実験をする場合に便利です。

複数のファイル全体にわたって最適化を行い、オブジェクト・ファイルを生成するには、-ipo_c オプションを使用します。このオプションを使用すると、-ipo の解説で述べた最適化処理を行いますが、最後のリンク段階に進む前にその処理は停止し、最適化されたオブジェクト・ファイルはそのまま残ります。このファイルのデフォルト名は ipo_out.o です。

複数のファイル全体にわたって最適化を行い、アセンブリ・ファイルを生成するには、-ipo_S オプションを使用します。このオプションを使用すると、-ipo の解説で述べた最適化処理を行いますが、最後のリンク段階に進む前にその処理は停止し、最適化されたアセンブリ・ファイルはそのまま残ります。このファイルのデフォルト名は ipo_out.s です。

「関数のインライン展開」も参照してください。

プロファイルに基づく最適化

概要: プロファイルに基づく最適化

プロファイルに基づく最適化 (PGO) を行うと、アプリケーションのどの領域が最も頻繁に実行するかがコンパイラに伝えられます。コンパイラはこの領域を知ると、以前のコンパイルのフィードバックを使用して、より慎重にアプリケーションの最適化を行います。例えば、PGO を使用するとコンパイラは関数のインライン化について的確な判断が下せる場合が多くなり、その結果、プロシージャ間の最適化の効率が向上します。

インストルメント済みプログラム

プロファイルに基づく最適化は、ソースコードとコンパイラの特殊コードからインストルメント済みプログラムを作成します。インストルメント済みコードを実行するたびにインストルメント済みプログラムは動的情報ファイルを生成します。2 回目にコンパイルすると、動的情報ファイルはサマリファイルにマージされます。このファイルのプロファイル情報を使用して、コンパイラは、プログラムで最もよく使用されるパスの実行の最適化を試みます。

サイズや速度に厳密に使用されるような他の最適化とは異なり、IPO と PGO の結果はさまざまです。これは、各プログラムは異なるプロファイルと異なる最適化の機会を持つためです。このガイドラインは、IPO と PGO を使用する利点があるがどうかを決定するのに役立ちます。

プロファイルに基づく最適化の方法

PGO は、コンパイルの時点での予測が困難な、繰返し実行される分岐を含むコードの最適化として最も有効に働きます。例えば、エラーチェック処理が多く含まれているのに、実際にエラー判定を下してみるとほとんどの場合エラー条件に合致しないコードがこれに相当します。これにより分岐予測をほとんど誤ることのない、いわゆる "コールド (cold)" なエラー処理コードを配置できます。"ホット" コードと "コールド" コードが交互に配置される状況をなくせば、命令キャッシュの動作が改善されます。例えば、PGO を使用するとコンパイラは関数のインライン化について的確な判断が下せる場合が多くなり、その結果、プロシージャ間の最適化の効率が向上します。

PGO フェーズ

PGO を行うには次の 3 つのフェーズが必要です:

- フェーズ 1: -prof_gen[x] によるインストルメンテーション・コンパイルとリンク
- フェーズ 2: 実行ファイルの実行によるインストルメント済み実行
- フェーズ 3: -prof_use によるフィードバック・コンパイル

PGO の適否を決める手掛かりの 1 つは、コードのどの部分に処理が集中しているかを知ることです。プログラムに与えられるデータセットがいつもほとんど変わらず、何度実行しても同じような動作にしかならない場合は、PGO によってプログラム実行が最適化されます。一方、プログラムに与えられるデータセットが毎回異なり、そのたびに種々のアルゴリズムが呼び出されることもあります。このような場合、プログラムは実行のたびに違った動作になるときがあります。

実行するたびに毎回動作が大きく異なるようなコードに対しては、PGO はそれほど有効ではありません。プロファイルを最新状態に保つ作業をするだけの価値が、そのプロファイル情報から得られるかどうかを常に考える必要があります。

x 修飾子を付けて -prof_gen[x] を使用すると、インテル® C++ コンパイラのコード・カバレッジ・ツールのような、コード・カバレッジ・ツールを使用して追加のソースの位置を収集することができます。これらのツールがないと、-prof_genx はよりよい最適化を行うことができず、並列コンパイル時間が遅くなります。

基本的な PGO オプション

オプション	説明
-prof_gen[x]	インストルメント済み実行の準備として、インストルメント済みコードをオブジェクト・ファイル内に生成するようにコンパイラに命令するオプションです。
-prof_use	プロファイルによって最適化された実行ファイルを生成し、利用可能な動的情報 (.dyn) ファイルをいくつかマージして pgopti.dpi ファイルを 1 個作成するようにコンパイラに命令するオプションです。

コードの動作が実行ごとに大きく異なる場合は、プロファイル情報によって得られるメリットが、最新のプロファイルを維持する作業に見合うものであるかどうか検討する必要があります。基本となるプロファイルに基づく最適化には、上記のオプションが PGO フェーズで使用されます。

インストルメント済みコードの作成

-prof_gen[x] オプションは、各基本ブロックの実行カウントを取得するために、プロファイル用プログラムをインストルメントします。PGO のフェーズ 1 を使用して、インストルメント済みのプログラムを実行する準備として、インストルメント済みコードをオ

ブジェクト・ファイルに生成するようにコンパイラに指示します。並列化は -prof_genx コンパイルで自動的にサポートされます。

プロファイルによって最適化された実行ファイルの生成

-prof_use オプションは、PGO のフェーズ 3 で使用され、プロファイルによって最適化された実行ファイルを生成し、使用できる動的情報 (.dyn) ファイルを pgopti.dpi ファイルにマージするようコンパイラに指示します。

注

動的情報ファイルは、インストルメント済み実行ファイルを実行するフェーズ 2 で生成されます。

インストルメント済みプログラムを何回か実行する場合は、-prof_use を指定すると、実行するたびに動的情報ファイルがマージされ、その前の pgopti.dpi ファイルは上書きされます。

関数分割の無効 (Itanium® プロセッサ用コンパイラのみ)

-fnsplit- は関数分割を無効にします。フェーズ 3 で -prof_use により関数分割は有効になります。これは、ルーチンを異なるセクションに分割することによって、コードの局所性を向上させるためです。異なるセクションとは、コールドまたは、あまり実行されないコードを含むセクションと、残りのコード (ホットコード) を含むセクションです。

次のような理由により、-fnsplit- を使用して、関数分割を無効にできます。

- 最も重要なことは、デバッグの機能を向上させることです。デバッグのシンボルテーブルでは、分割ルーチン、すなわちホット・コード・セクションとコールド・コード・セクションを持つルーチンを表示するのは困難です。
- -fnsplit- オプションはルーチン内の分割を無効にしますが、関数のグループ化を有効にします。これは、コールド・コード・セクションまたはホット・コード・セクションのいずれかにルーチン全体を配置する最適化機能です。関数のグループ化は、デバッグ機能を低下させません。
- 別の理由としては、プロファイル・データが実際のプログラム動作をしなかった場合、つまり、ルーチンが実際は稀ではなく頻繁に使用される場合です。

プロファイルに基づく最適化の例

PGO の基本的なフェーズは次の 3 つです:

- インストルメンテーション・コンパイルとリンク
- インストルメント済み実行
- フィードバック・コンパイル

インストルメンテーション・コンパイルとリンク

-prof_gen を使用して、インストルメント済み情報付きの実行ファイルを生成します。また、多くのプログラムに適した -prof_dir オプションを使用してください。特に、複数のディレクトリに渡るソース・ファイルを持つアプリケーションには使用してください。-prof_dir は、プロファイル情報が同じ場所で生成されることを保証します。
例:

```
prompt>icpc -prof_gen -prof_dir /profdata -c a1.cpp a2.cpp
a3.cpp
prompt>icpc a1.o a2.o a3.o
```

2 番目のコマンドの代わりにリンカを直接使用して、インストルメント済みプログラムを生成できます。

インストルメント済み実行

代わりの何らかのデータセットを使用してインストルメント済みプログラムを実行して、動的情報ファイルを作成します。

```
prompt>./a.out
```

作成される動的情報ファイルは、a.out を実行するたびに毎回別の名前と .dyn というサフィックスが付きます。また、このインストルメント済みファイルは、特定のデータセットでこのプログラムを実行したときの振舞いを予測するのに役立ちます。このプログラムは、入力データを変えて何度でも実行できます。

フィードバック・コンパイル

-prof_use を指定してソースファイルのコンパイルとリンクを行い、動的情報を使用して、そのプロファイルに従ってプログラムを最適化します。

```
prompt>icpc -prof_use -ipo a1.cpp a2.cpp a3.cpp
```

最適化のほかに、コンパイラは pgopti.dpi ファイルを生成します。一般に、第 1 フェーズではデフォルトの最適化 (-O2) を行い、第 3 フェーズでは、さらに高度な最

適化 (-ipo) を指定します。上の例では第 1 フェーズで -O2 を使用し、第 3 フェーズで -O2 -ipo を使用しました。

注

-prof_gen[x] オプションを使用すると、-ipo オプションは無視されます。x 修飾子を付けると、補足情報が収集されます。

PGO の環境変数

下の表に示した環境変数は、動的情報ファイルを格納するディレクトリを指定するときや、pgopti.dpi を上書きするかどうかを指定するときに使用します。環境変数の設定方法については、ご使用のオペレーティング・システムのマニュアルを参照してください。

プロファイルに基づく最適化に使う環境変数

変数	説明
PROF_DIR	動的情報ファイルの作成先ディレクトリを指定する変数です。この変数は、プロファイル処理の 3 つのフェーズすべてに適用されます。
PROF_NO_CLOBBER	フィードバック・コンパイル・フェーズでの処理を少し変更する変数です。デフォルトでは、フィードバック・コンパイル・フェーズでは、すべての動的情報ファイルから得たデータがマージされます。そして .dyn ファイルが既存の pgopti.dpi ファイルよりも新しい場合は、pgopti.dpi ファイルを新たに作成します。この変数を設定すると、コンパイラは既存の pgopti.dpi ファイルを上書きせずに、警告を発行します。他の動的情報ファイルを使用する場合は、その既存の pgopti.dpi ファイルを削除してください。

profmerge ツールによるソースファイルの再配置

コンパイラは、プロファイル・サマリ情報を調べるためにソースファイルへのフルパスを使用します。このため、デフォルトでは、次の場合に問題が発生します:

- アプリケーションのソースを移動した後でプロファイル・サマリ・ファイル (.dpi) を使用する場合
- 異なるディレクトリにある同一のアプリケーション・ソースをビルドしている別のユーザとプロファイル・サマリ・ファイルを共有する場合

ソースの再配置

アプリケーション・ソースの移動、およびプロファイル・サマリ・ファイルの共有を可能にするには、-src_old および -src_new オプションを指定して profmerge ツールを使用します。
例:

```
prompt>profmerge -prof_dir <p1> -src_old <p2> -src_new <p3>
```

各アイテムの意味は次のとおりです:

- <p1> は、動的情報ファイル（.dpi）へのフルパスです。
- <p2> は、ソースファイルへの古いフルパスです。
- <p3> は、ソースファイルへの新しいフルパスです。

上記のコマンドは pgopti.dpi ファイルを読みます。profmerge ツールは、pgopti.dpi ファイル中の <p2> プリフックスではじまるソースパスを <p3> に置換します。pgopti.dpi ファイルは、新しいソースパス情報で更新されます。

pgopti.dpi ファイルに対して profmerge ツールを複数回実行することができます (例えば、ソースファイルが複数のディレクトリにある場合)。
例:

```
prompt>profmerge -prof_dir -src_old /src/prog_1 -src_new
/src/prog_2

prompt>profmerge -prof_dir -src_old /proj_1 -src_new
/proj_2
```

-src_old と -src_new に指定する値では大文字と小文字は区別されません。同様に、右上がりスラッシュ (/) とバックスラッシュ (\) 文字は同一の文字として扱われます。

profmerge のソース再配置機能は pgopti.dpi ファイルを修正するため、ソースの再配置を行う前にファイルをバックアップしておくとよいでしょう。

コード・カバレッジ・ツール

インテル® C++ コンパイラのコード・カバレッジ・ツールは、IA-32 アーキテクチャと Itanium® アーキテクチャの両方で使用できるツールで、さまざまな方法により、開発効率を改善し、問題を少なくして、アプリケーションのパフォーマンスを向上させます。コード・カバレッジ・ツールの主な特徴は次のとおりです:

- 色分けしたアプリケーションのコード・カバレッジ情報のビジュアル・プレゼンテーション
- アプリケーションの各基本ブロックの動的実行カウントの表示
- アプリケーションを 2 回実行した場合の差分カバレッジまたはプロファイルの比較

コマンドライン構文

このツールの構文は次のとおりです。

```
codecov [-codecov_option]
```

`-codecov_option` はツール・オプションです。オプションを使用しない場合、ツールはプログラム全体についてトップレベルのコード・カバレッジを出力します。

ツールのオプション

次の表は、ツールが使用するオプションの一覧です。

オプション	説明	デフォルト
`-help`	コード・カバレッジ・ツールのすべてのオプションを出力します。	
`-spi` *file*	静的プロファイル情報ファイル (`.spi`) のパス名を設定します。	`pgopti.spi`
`-dpi` *file*	動的プロファイル情報ファイル (`.dpi`) のパス名を設定します。	`pgopti.dpi`
`-prj`	プロジェクト名を設定します。	
`-counts`	動的実行カウントを生成します。	
`-nopartial`	一部カバーされたコードを完全にカバーされたコードとして扱います。	
`-comp`	処理するファイルのリストを含むファイル名を設定します。	
`-ref`	ref_dpi_file に関する差分カバレッジを検索します。	
`-demang`	関数名とその引数の両方を復号します。	
`-mname`	Web ページの所有者の名前を設定します。	
`-maddr`	Web ページの所有者のメールアドレスを設定します。	
`-bcolor`	カバーされなかったブロックの html カラー名またはコードを設定します。	`#ffff99`
`-fcolor`	カバーされなかった関数の html カラー名またはコードを設定します。	`#ffcccc`
`-pcolor`	一部カバーされたコードの html カラー名またはコードを設定します。	`#fafad2`

-ccolor	カバーされたコードの html カラー名またはコードを設定します。	#ffffff
-ucolor	不明なコードの html カラー名またはコードを設定します。	#ffffff

アプリケーションのコード・カバレッジのビジュアル・プレゼンテーション

アプリケーションをテストするときにインストルメント済みバイナリを実行して集められたプロファイル情報を基に、インテル・コンパイラはコード・カバレッジ・ツールを使用して HTML ファイルを作成します。これらの HTML ファイルはテストによって実行された (実行されなかった) ソースコードの部分を示します。パフォーマンス・ワークロードのプロファイルが適用されると、コード・カバレッジ情報は実行されたワークロードがアプリケーションの重要なコードをどの程度カバーするかを示します。プロファイルに基づく最適化の利点を完全に受けるためには、パフォーマンス・クリティカルなモジュールの高いカバレッジが必要です。

コード・カバレッジ・ツールは、2 つのレベルのカバレッジを作成することができます:

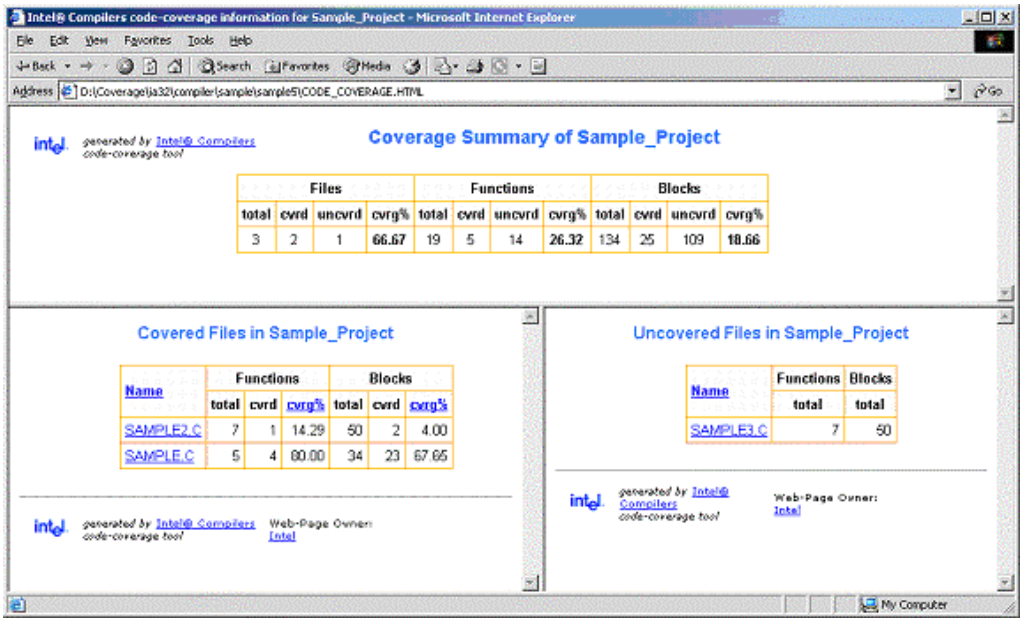
- トップレベル -- 選択されたモジュールのグループ
- 個々のモジュール・ソース・ビュー

トップレベルのカバレッジ

トップレベルのカバレッジは選択されたモジュールのコード・カバレッジ全体をレポートします。次のオプションがあります:

- 処理するモジュールを選択することができます。
- 選択されたモジュールについて、ツールはモジュールとそのカバレッジ情報のリストを生成します。情報には、モジュールにある関数とブロックの総数、カバーされた部分が含まれます。
- レポートされた表でカラムのタイトルをクリックすると、リストが次の項目に基づいて昇順または降順でソートされます:
  - 基本ブロック・カバレッジ
  - 関数カバレッジ
  - 関数名

次の例は、プロジェクトのトップレベル・カバレッジの要約を示しています。モジュール名 (例えば、SAMPLE.C) をクリックすると、ブラウザはそのモジュールのカバレッジ・ソース・ビューを表示します。



フレームのブラウジング

コード・カバレッジ・ツールは、カバーされなかったコードが容易に識別できるようにフレームを作成します。上のフレームにはカバーされなかった関数のリストが、下のフレームにはカバーされた関数のリストがそれぞれ表示されます。カバーされなかった関数については、各関数の基本ブロックの総数も表示されます。カバーされた関数については、ブロックの総数とカバーされたブロックの数、その比率 (つまり、カバレッジ率) が表示されます。

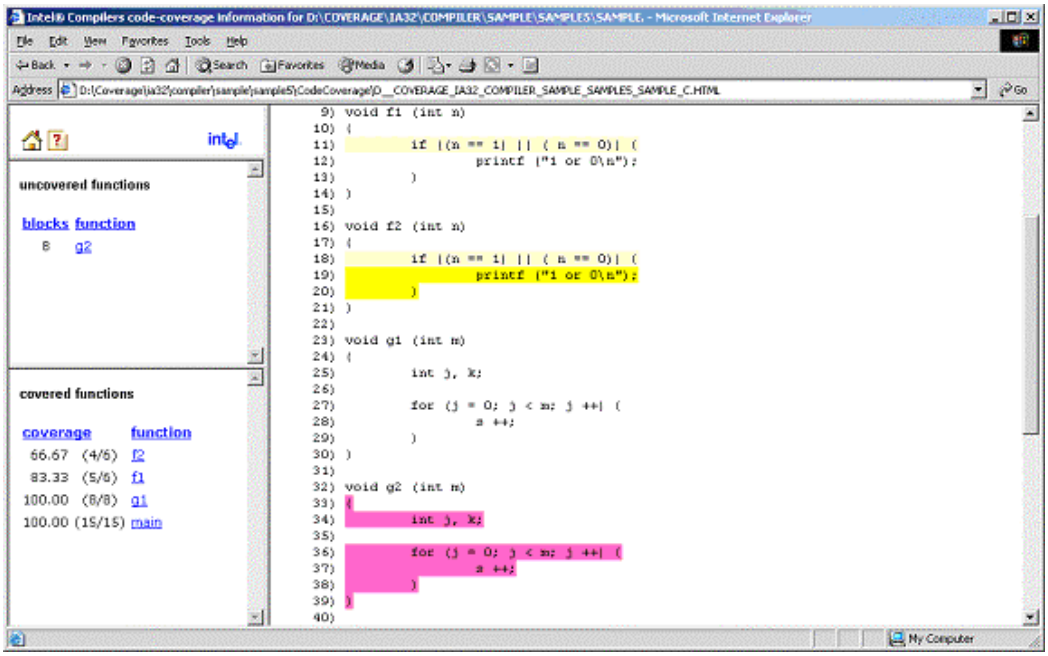
例えば、66.67(4/6) は、対応する関数の 6 つのブロックのうち 4 つのブロックがカバーされたことを示します。関数のブロック・カバレッジ率は 66.67% になります。これらのリストは、カバレッジ率、ブロック数、関数名でソートすることができます。 関数名はソースビューで関数が開始する場所とリンクされています。つまり、クリック 1 つで、ユーザはリストで最も少なくカバーされた関数を見ることができます。もう一回クリックすると、ブラウザは関数のソースを表示します。その後、ユーザはソースビューでスクロールして関数のソースを参照することができます。

個々のモジュール・ソース・ビュー

個々のモジュール・ソース・ビューで、ツールはカバーされなかった関数のリストとカバーされた関数のリストを表示します。リストは、ソースコードを容易に参照できるように、2 つの別々のフレームに表示されます。リストは、次の項目でソートすることができます:

- カバーされなかった関数内のブロックの数
- カバーされた関数のブロック・カバレッジ
- 関数名

次の図は、SAMPLE.C のカバレッジ・ソース・ビューの例です。



コード・カバレッジ用の配色の設定

ツールは、次のようにカバレッジ・カテゴリを色分けします:

- カバーされたコード
- カバーされなかった基本ブロック
- カバーされなかった関数
- 一部カバーされたコード
- 不明

カバレッジ情報を示すためにツールが使用するデフォルト色を、次の表に示します。

色の種類	意味
カバーされたコード	テストが実行されたコードの部分を示します。デフォルト色は -ccolor オプションで無効にすることができます。
カバーされなかった基本ブロック	テスト中に実行された関数内にある、テストが実行されなかった基本ブロックを示します。デフォルト色は -bcolor オプションで無効にすることができます。
カバーされなかった関数	テスト中に呼び出されなかった関数を示します。デフォルト色は -fcolor オプションで無効にすることができます。
一部カバーされたコード	この場所でコード用に複数の基本ブロックが生成されたことを示します。カバーされたブロックとカバーされなかったブロックがあります。デフォルト色は -pcolor オプションで無効にすることができます。

不明	このソース行に対してソースが生成されなかったことを示します。ほとんどの場合、この場所のソースはコメント、ヘッダファイルのインクルード、または変数宣言です。デフォルト色は -ucolor オプションで無効にすることができます。

デフォルト色は、上記の表の各カバレッジ・カテゴリで説明されているオプションを使用して、任意の HTML カラーにカスタマイズすることができます。

コード・カバレッジ・プレゼンテーションでは、コード・カバレッジ・ツールは次のヒューリスティックを使用します。ソース文字は、プロファイル情報によって基本ブロックの先頭として示されたソースの場所に達するまでスキャンされます。その基本ブロックのプロファイル情報でカバレッジのカテゴリが変わったことが示された場合、ツールはコードのその部分のカバレッジ条件に対応するように色を変更し、HTML ファイルに適切なタグを挿入します。

注

コードのコンテキスト中の色を解釈する必要があります。例えば、実行されなかった基本ブロックに続くコメント行はカバーされなかったブロックと同じ色になります。C/C++ アプリケーションの閉じ括弧も同様です。

モジュール・サブセットのカバレッジ解析

インテル・コンパイラのコード・カバレッジ・ツールの機能の 1 つに、アプリケーションのモジュール・サブセットの効率的なカバレッジ解析があります。この解析を行うには、-comp オプションを選択してツールを実行します。

アプリケーション全体、またはサブセットのプロファイル情報を生成し、カバーされたモジュールを異なるコンポーネントに分割してから、コード・カバレッジ・ツールを使用して各コンポーネントのカバレッジ情報を取得します。アプリケーション・モジュールのサブセットのみが -prof_genx オプション付きでコンパイルされた場合、カバレッジ情報はこのコンパイル・オプションに関係するモジュール用にだけ生成されるため、他のモジュール用のプロファイルを生成するオーバーヘッドを回避することができます。

処理するモジュールを指定するには、-comp オプションを使用します。このオプションは引数としてファイル名を指定します。指定するファイルは、解析するモジュール名またはディレクトリ名を含むテキストファイルでなければなりません:

```
codecov -prj Project_Name -comp component1
```

注

コンポーネント・ファイルの各行には 1 つのモジュール名のみを記述するようにしてください。

コンポーネント・ファイルに記述されているアプリケーションのすべてのモジュールについてカバレッジ解析が行われます。例えば、上記の例で、ファイル `component1` のある行に `mod1.cpp` と記述されていた場合、同じ名前のモジュールがあるアプリケーションのすべてのモジュールが選択されます。ユーザは、特定のパス情報を記述することで特定のモジュールを指定することができます。例えば、ある行に `/cmp1/mod1.cpp` と記述されていた場合、ディレクトリ `cmp1` にある `mod1.cpp` という名前のモジュールのみが選択されます。コンポーネント・ファイルが指定されていない場合、`-prof_genx` オプション付きでコンパイルされたすべてのファイルがカバレッジ解析用に選択されます。

動的カウンタ

この機能は、アプリケーションの各基本ブロックの動的実行カウントを表示して、カバレッジとパフォーマンス・チューニングの両方に有用な情報を提供します。

コード・カバレッジ・ツールは、動的実行カウントに関する情報を生成するように設定することができます。この設定を行うには、`-counts` オプションを使用します。カウント情報は、対応する基本ブロックが開始するソースの場所で、**^** 記号の後のコードで正確に表示されます。そのソースの場所でコードに複数の基本ブロックが生成される場合 (例えば、マクロ)、そのようなブロックの総数と実行されたブロックの総数が実行カウントの前に表示されます。

特定の状況では、1 つのソース用に生成されたすべてのブロックを 1 つのエンティティとして考慮する必要があります。この場合、少なくとも 1 つのブロックがカバーされたのであれば、1 つのソースの場所用に生成されたすべてのブロックがカバーされたと仮定する必要があります。仮定するには、`-nopartial` オプションを使用します。このオプションが指定されれば、カバレッジの決定は無効になり、関連する統計がそれに従って調節されます。コードの 11 行目および 12 行目は、12 行目の `printf` 分がカバーされたことを示しています。しかし、11 行目の条件の 1 つのみが常に真であったとします。`-nopartial` オプションを使用すると、ツールは (11 行目のコードのように) 一部がカバーされたコードを完全にカバーされたコードとして扱います。

差分カバレッジ

コード・カバレッジ・ツールを使用すると、アプリケーションの 2 つの実行 (リファレンス実行と新規実行) を比較して、新規実行ではカバーされるがリファレンス実行ではカバーされないコードを識別することができます。この機能は、アプリケーションがカスタ

マによって実行される場合に、アプリケーションのテストでカバーされないアプリケーションのコードの部分を検索するために使用することができます。また、アプリケーションのテストスペースに新しく追加されたテストのインクリメンタル・カバレッジの影響を検索するためにも使用されます。

差分カバレッジ用のリファレンス実行の動的プロファイル情報は、次のコマンドのように、-ref オプションで指定されます。

```
codecov -prj Project_Name -dpi customer.dpi -ref
appTests.dpi
```

差分カバレッジのカバレッジ統計は、新規実行で実行され、リファレンス実行で実行されなかったコードの比率をパーセントで表示します。この場合、コード・カバレッジ・ツールはカバーされなかったコードを含むモジュールのみを表示します。

ソースビューの配色も同様に解釈すべきです。コードが両方の実行で同じカバレッジ・プロパティ (カバーされたまたはカバーされなかった) の場合、コードはカバーされたコードとして扱われます。コードが新規実行でカバーされ、リファレンス実行でカバーされなかった場合、コードはカバーされなかったコードとして扱われます。逆に、コードがリファレンス実行でカバーされ、新規実行でカバーされなかった場合、差分カバレッジのソースビューはコードをカバーされたコードとして表示します。

差分カバレッジ用の実行

インテル・コンパイラのコード・カバレッジ・ツールを差分カバレッジ用に実行するには、次のファイルが必要です:

- アプリケーションのソース
- インテル・コンパイラでインストルメント済みバイナリ用に -prof_genx オプション付きでアプリケーションをコンパイルしたときに生成される .spi ファイル。
- インテル・コンパイラの profmerge ツールで各アプリケーション・テストの動的プロファイル情報ファイル (.dyn) をマージして生成される .dpi ファイル、またはインテル・コンパイラで -prof_use オプション付きでアプリケーションをコンパイルしたときに暗黙的に生成される .dpi ファイル。

一旦必要なファイルが利用可能になると、コード・カバレッジ・ツールを次のコマンドラインから起動できます:

```
codecov -prj Project_Name -spi pgopti.spi -dpi pgopti.dpi
```

-spi および -dpi オプションは、対応するファイルへのパスを指定します。

コード・カバレッジ・ツールには、各 HTML ページの最後に -mname および -maddr オプションを使用して指定されたアドレスに電子メールを送信するリンクを生成する、次の追加オプションもあります。

```
codecov -prj Project_Name -mname John_Smith -maddr
js@company.com
```

テスト・プライオリタイゼーション・ツール

インテル® コンパイラのテスト・プライオリタイゼーション・ツールを使用すると、以前のアプリケーション実行プロファイルを基にアプリケーション・テストの選択と重要度付けを行う、プロファイルに基づく最適化を行うことができます。ツールは、テストがボトルネックとなる大規模なアプリケーションのテストと開発にかかる時間を大幅に節約します。ツールは、IA-32 アーキテクチャと Itanium® アーキテクチャの両方で使用することができます。

ツールは、アプリケーションのコードのサブセットに対して最も適切なテストを選択し、重要度付けを行います。アプリケーションの特定のモジュールが変更された場合、テスト・プライオリタイゼーション・ツールはその変更によって最も影響を受けるテストを知らせます。ツールは、以前に実行されたアプリケーションのプロファイル・データを分析して、アプリケーションのコンポーネントとテストの依存性を確認し、この情報を基にテストのプロセスをガイドします。

特徴と利点

ツールは、アプリケーションのコード・カバレッジを基に、効果的なテスト階層を示します。ツールを使用する利点は、次のように要約することができます:

- アプリケーションの任意のサブセットで全体をカバーするために必要なテストの数を最小限にする。ツールはテストの全セットと全く同じコード・カバレッジを達成するアプリケーション・テストの最小サブセットを定義します。
- テストのターンアラウンド時間を短くする。長い時間を費やして多くの失敗を見つける代わりに、ユーザは、セットの変更によって問題が発生する少数のテストを素早く見つけることができます。
- テストの実行時のデータを基に、最短時間で特定のレベルのコード・カバレッジを達成するテストを選択して重要度付けを行います。

コマンドライン構文

このツールの構文は次のとおりです:

```
tselect -dpi_list file
```

-dpi_list は必須のツール・オプションで、重要度付けが必要なテストの .dpi ファイルのリストを含む DPI リストファイル (file) へのパスを設定します。

ツールのオプション

次の表は、ツールが使用するオプションの一覧です。

オプション	説明
-help	テスト・プライオリタイゼーション・ツールのすべてのオプションを出力します。
-spi *file*	静的プロファイル情報ファイル (.spi) のパス名を設定します。デフォルトは pgopti.spi です。
-dpi_list *file*	動的プロファイル情報ファイル (.dpi) の名前を含むファイルのパス名を設定します。ファイルの各行には 1 つの .dpi ファイル名のみを記述し、オプションでファイル名の後にその実行時間を記述します。名前はテストを一意に識別しなければなりません。
-prof_dpi *file*	出力レポートファイルのパス名を設定します。
-comp	処理するファイルのリストを含むファイル名を設定します。
-cutoff *value*	累積ブロック・カバレッジがあらかじめ計算された合計カバレッジの *value*% に達した場合に終了します。*value* は 0.1 以上でなければなりません (例えば、99.00)。100 まで設定できます。
-nototal	あらかじめ合計カバレッジを計算しません。
-mintime	テストの実行時間を最小限にします。各テストの実行時間は、dpi_list ファイルのテスト名が含まれる行に、テスト名に続いて dd:hh:mm:ss 形式で記述されていなければなりません。
-verbose	プログラム進行に関してより多くのロギング情報を生成します。

使用要件

アプリケーションのテストでテスト・プライオリタイゼーション・ツールを実行するには、次のファイルが必要です:

- インテル・コンパイラでインストルメント済みバイナリ用に -prof_genx オプション付きでアプリケーションをコンパイルしたときに生成される .spi ファイル。
- インテル・コンパイラの profmerge ツールで各アプリケーション・テストの動的プロファイル情報ファイル (.dyn) をマージして生成される .dpi ファイル。ユーザは、個々のテスト用に生成されたすべての .dyn ファイルに profmerge ツールを適用して、テストを一意に識別できるように .dpi ファイルに名前を付ける必要があります。profmerge ツールは指定されたディレクトリに存在するすべての .dyn ファイルをマージします。

注

以前に実行した、または他のテストで使用した、無関係な .dyn ファイルがディレクトリ中に含まれていないことを確認することは非常に重要です。この確認を怠ると、プロファイル情報は不適切なプロファイル・データに基づくことになります。その結果、不適切なカバレッジ情報が生成され、最適化されたコードのパフォーマンスに悪影響を与えることがあります。

注

ツールを正しく実行するには、次の操作を行ってください:

- 各テストが一意に識別できるように、各テスト .dpi ファイルに名前を付けます。
- DPI リストファイル (すべての .dpi テストファイルの名前を記述したテキストファイル) を作成します。このファイルの名前がテスト・プライオリタイゼーション・ツールの実行コマンドの入力になります。DPI リストファイルの各行には、1 つの .dpi ファイル名のみを記述するようにしてください。オプションで、ファイル名に続けてテストの実行時間を dd:hh:mm:ss 形式で記述することができます。

例: Test1.dpi 00:00:60:35 は、Test1 が (0 日 0 時間) 60 分 35 秒実行されることを示します。実行時間の記述はオプションです。しかし、実行時間が記述されていない場合、ツールは実行時間を最小限にするテストの重要度付けを行いません。テストの数を最小限にする重要度付けのみを行います。

使用モデル

次の図は、テスト・プライオリタイゼーション・ツールの使用モデルを示しています。

IA-32 システム用の簡単な例 (`myApp.c`) について順に説明します。

1. 環境変数の設定

```
PROF_DIR=/myApp/prof_dir
```

2. コマンドの発行

```
prompt>icpc -prof_genx myApp.c
```

このコマンドは、プログラムをコンパイルして、インストルメント済みバイナリと静的プロファイル情報ファイル `pgopti.spi` を生成します。

3. コマンドの発行

```
rm PROF_DIR /*.dyn
```

関係しない `.dyn` ファイルを削除します。

4. コマンドの発行

```
myApp < data1
```

このコマンドは、インストルメント済みアプリケーションを実行して、PROF_DIR で指定されたディレクトリに拡張子 .dyn で 1 つ以上の新規動的プロファイル情報ファイルを生成します。

5. コマンドの発行

```
profmerge -prof_dpi Test1.dpi
```

このステップで、profmerge ツールは、すべての .dyn ファイルを、Test1 におけるアプリケーションの合計のプロファイル情報を表わす 1 つのファイル (Test1.dpi) にマージします。

6. コマンドの発行

```
rm PROF_DIR /*.dyn
```

関係しない .dyn ファイルを削除します。

7. コマンドの発行

```
myApp < data2
```

このコマンドは、インストルメント済みアプリケーションを実行して、PROF_DIR で指定されたディレクトリに拡張子 .dyn で 1 つ以上の新規動的プロファイル情報ファイルを生成します。

8. コマンドの発行

```
profmerge -prof_dpi Test2.dpi
```

このステップで、profmerge ツールは、すべての .dyn ファイルを、Test2 におけるアプリケーションの合計のプロファイル情報を表わす 1 つのファイル (Test2.dpi) にマージします。

9. コマンドの発行

```
rm PROF_DIR /*.dyn
```

関係しない .dyn ファイルを削除します。

10. コマンドの発行

```
myApp < data3
```

このコマンドは、インストルメント済みアプリケーションを実行して、PROF_DIR で指定されたディレクトリに拡張子 .dyn で 1 つ以上の新規動的プロファイル情報ファイルを生成します。

11. コマンドの発行

```
profmerge -prof_dpi Test3.dpi
```

このステップで、profmerge ツールは、すべての .dyn ファイルを Test3 におけるアプリケーションの合計のプロファイル情報を表わす 1 つのファイル (Test3.dpi) にマージします。

12. tests_list という名前で 3 行のファイルを作成します。1 行目に Test1.dpi、2 行目に Test2.dpi、3 行目に Test3.dpi と記述します。

これらの項目が利用可能になると、次の例で記述されているように PROF_DIR ディレクトリでコマンドラインからテスト・プライオリタイゼーション・ツールを起動できます。すべての例で、同じセットのデータを参照しています。

例 1 テストの数を最小限にする

```
tselect -dpi_list tests_list -spi pgopti.spi
```

-spi オプションは .spi ファイルへのパスを指定します。

テスト・プライオリタイゼーション・ツールを実行した場合の出力例を次に示します:

```
Total number of tests = 3
Total block coverage ~ 52.17
Total function coverage ~ 50.00
```

Num	%RatCvrg	%BlkCvrg	%FncCvrg	Test Name @ Options
1	87.50	45.65	37.50	Test3.dpi
2	100.00	52.17	50.00	Test2.dpi

この例で、テスト・プライオリタイゼーション・ツールは次の情報を出力しています:

- 3 つのテストすべてを実行して、52.17% のブロック・カバレッジと 50.00% の関数カバレッジを達成した。
- Test3 はアプリケーションの基本ブロックの 45.65% をカバーしている。これは 3 つのテストすべてで達成できる合計ブロック・カバレッジの 87.50% である。
- Test2 を追加することで、52.17% の累積ブロック・カバレッジまたは Test1、Test2、Test3 の合計ブロック・カバレッジの 100% を達成した。
- Test1 を除去しても合計ブロック・カバレッジに悪影響はない。

例 2 実行時間を最小限にする

tests_list ファイルで、各テストの実行時間が次のように記述されているとします:

```
Test1.dpi 00:00:60:35
Test2.dpi 00:00:10:15
Test3.dpi 00:00:30:45
```

次のコマンドは、実行時間を最小限にする -mintime オプション付きでテスト・プライオリタイゼーション・ツールを実行します:

```
tselect -dpi_list tests_list -spi pgopti.spi -mintime
```

出力例を次に示します:

```
Total number of tests = 3
Total block coverage ~ 52.17
Total function coverage ~ 50.00
Total execution time = 1:41:35
```

num	elapsedTime	%RatCvrg	%BlkCvrg	%FncCvrg	Test Name @ Options
1	10:15	75.00	39.13	25.00	Test2.dpi
2	41:00	100.00	52.17	50.00	Test3.dpi

この例では、すべてのテストを続けて実行すると 1 時間 45 分 35 秒必要ですが、選択したテストを実行すると 41 分で同じ合計ブロック・カバレッジを達成できることが示されています。

注

優先度付けが実行時間を最小限にすることを基に行われる場合のテストの順序 (最初に Test2、次に Test3) は、優先度付けがテストの数を最小限にすることを基に行われる場合と異なることがあります。例 1 では、最初に Test3、次に Test2 になります。例 2 では、Test2 は実行時間あたりのカバレッジが最も高くなるテストです。このため、最初のテストとして選択されます。

他のオプションの使用

-cutoff オプションは、テスト・プライオリタイゼーション・ツールが指定されたレベルの基本ブロック・カバレッジに達した場合、ツールを終了します。

```
tselect -dpi_list tests_list -spi pgopti.spi -cutoff 85.00
```

ツールが上記の例で 85.00 の cutoff 値で実行された場合、45.65% のブロック・カバレッジを達成する `Test3` のみが選択されます。これは 3 つのテストすべてで達する合計ブロック・カバレッジの 87.50% に相当します。

テスト・プライオリタイゼーション・ツールは、すべてのテストを実行して得られる合計カバレッジを決定するために、すべてのプロファイル情報を最初にマージします。`-nototal` オプションは、このステップをスキップします。この場合、全体的なカバレッジは不明なまま、確実なカバレッジ情報のみがレポートされます。

PGO API: プロファイル情報生成サポート

PGO API サポートの概要

プロファイル情報生成サポート (Profile Information Generation Support) で、プロファイルに基づいた最適化のインストルメント済みの実行フェーズ中にプロファイル情報の生成を制御できます。通常、プロファイル情報は、インストルメント済みアプリケーションが、標準的な `exit()` 関数を呼び出してアプリケーションを終了したときにインストルメント済みアプリケーションによって生成されます。このセクションで説明される関数は、プロファイル情報が次の状況で確実に生成されなければならない場合があります。

- 非標準の終了ルーチンでインストルメント済みアプリケーションが終了する場合
- インストルメント済みアプリケーションが非終了 (`exit()` が呼び出されない) アプリケーションの場合
- プロファイル情報の生成時を制御する必要がある場合

このセクションでは、プロファイル情報生成サポート (Profile Information Generation Support、Profile IGS) を構成する関数および環境変数について説明します。関数は、関数が使用されるソースファイルの先頭に `#include <pgouser.h>` を挿入することにより使用することができます。

`-prof_gen` または `-prof_genx` のいずれかを使用してコンパイルする場合、コンパイラは _PGO_INSTRUMENT の `define` を設定します。

プロファイル情報のダンプ

```
void _PGOPTI_Prof_Dump(void);
```

説明

この関数は、インストルメントされたアプリケーションにより収集されたプロファイル情報をダンプします。プロファイル情報は .dyn ファイルに記録されます。

推奨する使用方法

アプリケーションを終了する関数本体にこの関数への呼び出しを挿入します。通常、_PGOPTI_Prof_Dump の呼び出しは、一度だけでなければなりません。この関数を _PGOPTI_Prof_Reset() とともに使用して、(入力データの複数のセットから) 複数の .dyn ファイル を作成することができます。

例

```
// Selectively collect profile information for the portion
// of the application involved in processing input data.

input_data = get_input_data();

while(input data)
{
    PGOPTI Prof Reset();
   process data(input data);
   _PGOPTI_Prof_Dump();
   input data = get input data();
}
```

動的プロファイル・カウンタのリセット

```
void _PGOPTI_Prof_Reset(void);
```

説明

この関数は、動的プロファイル・カウンタをリセットします。

推奨する使用方法

この関数を使用し、インストルメント済みのアプリケーションのセクションでプロファイル情報を収集する前にプロファイル・カウンタをクリアにします。
PGOPTI_Prof_Dump() の例を参照してください。

プロファイル情報のダンプとリセット

```
void _PGOPTI_Prof_Dump_And_Reset(void);
```

説明

この関数は、2 回以上呼び出される場合があります。各呼び出しで、プロファイル情報を新しい .dyn ファイルにダンプします。そして、動的プロファル・カウンタはリセットされ、その後、インストルメント済みのアプリケーションの実行が続行します。

推奨する使用方法

この関数の定期的な呼び出しにより、非終了アプリケーションは、1 つまたは複数のプロファイル情報ファイルを作成することができます。これらのファイルは、プロファイルに基づく最適化のフィードバック・フェーズ中に結合されます。この関数の直接的な使用により、プロファイル情報の生成時にアプリケーションを正確に制御することができます。

インターバル・プロファイル・ダンプ

```
void _PGOPTI_Set_Interval_Prof_Dump(int interval);
```

説明

この関数は、インターバル・プロファイル・ダンプをアクティブにし、ダンプ発生時のおおよその頻度を設定します。interval パラメータは、プロファイルのダンプが発生する時間の間隔 (ミリ秒単位) で指定します。例えば、interval が 5000 に設定された場合、プロファイル・ダンプとリセットは約 5 秒ごとに行われます。ダンプとリセットの時間設定は、アプリケーションのインストルメントされた関数へのエントリに対して行われるため、インターバルは概算となります。

注

- interval を 0 または負の数に設定すると、インターバル・プロファイルダンプは無効になります。
- interval が非常に小さな値で設定されると、インストルメント済みのアプリケーションがプロファイル情報のダンプにほとんどの時間を費やしてしまう場合があります。interval にはできるだけ大きな値を設定し、アプリケーションが本来の作業を行え、十分なプロファイル情報が収集されるようにしてください。

推奨する使用方法

この関数は、非終了 アプリケーションの開始時に呼び出され、インターバル・プロファイル・ダンプを開始します。環境変数の PROF_DUMP_INTERVAL をアプリケーションが開始する前に必要な interval の値に設定して、インターバル・プロファイル・ダ

ンプを開始することもできます。インターバル・プロファイル・ダンプの目的は、プロファイルするのに非終了アプリケーションのソースコードの変更を最小限にすることです。

環境変数

PROF_DUMP_INTERVAL

この環境変数は、インストルメント済みのアプリケーションでインターバル・プロファイル・ダンプに使用されることがあります。詳細は、_PGOPTI_Set_Interval_Prof_Dump の推奨する使用方法を参照してください。

高水準言語の最適化 (HLO)

概要: 高水準言語の最適化

高レベル最適化 (HLO) とは、Fortran や C++ といった高水準プログラミング言語で開発されたアプリケーションに含まれているループや配列など、ソースコード上の特性を利用した最適化手法です。これには、ループ交換、ループ融合、ループ展開、ループ分配、アンロール・ジャム、ブロッキング、データ・プリフェッチ、スカラ置換、データ・レイアウトの最適化といった手法があります。高レベルな最適化をオンにするオプションは -O3 です。

IA-32 および Itanium® ベース・アプリケーション	
-O3	-O2 オプションに加えて、さらに強力な最適化 (例えば、ループ変換やプリフェッチ) を有効にします。-O3 は、最も高速になるように最適化を行いますが、パフォーマンスが向上しないプログラムもあります。
IA-32 アプリケーション	
-O3	ベクトル化オプション -ax{K|W|N|B|P} および -x{K|W|N|B|P} と -O3 を組み合わせて指定すると、-O2 よりも詳細にデータの依存性を分析します。そのためコンパイル時間が長くなる場合があります。

注

-fast オプションは、実行時のパフォーマンスを向上させる次のオプションを含めることで、プログラム全体にわたって実行速度を上げます:

- -O3 (最高速で高レベルの最適化)
- -ipo (複数ファイルにわたるプロシージャ間の最適化を有効にします)
- -static (共有ライブラリとリンクしないようにします)

-fast で設定されるオプションの 1 つを無効にするには、コマンドラインで -fast オプションの後に無効にするオプションを指定します。-fast で設定されるオプションは、バージョンによって変更になる場合があります。

特定のプロセッサ用に -fast の最適化を行うには、-x オプションのうちの 1 つを使用してください。
例:

```
prompt>icpc -fast -xW source_file.cpp
```

ループ変換

ループ変換ができるかどうかは、データの依存関係によって異なります。誘導変数の排除、定数伝播、コピー伝播、先行代入、不要コードの排除といった手法もあります。ループ変換の手法には次のものがあります。

- ループ正規化
- ループ逆転
- ループ交換/並べ替え
- ループ傾斜
- ループ分配
- ループ融合
- スカラ置換

以上は IA-32 アーキテクチャと Itanium® アーキテクチャの両方で使用できるループ変換ですが、これ以外にも Itanium アーキテクチャではコラプス手法が利用できます。

IVDEP ディレクティブによるループ・キャリー・メモリ依存の不在

Itanium® ベース・アプリケーションの場合、-ivdep_parallel オプションは IVDEP ディレクティブが指定されたループにループ・キャリー・メモリ依存が確実にないことを示します。この手法は、スパース・マトリックス・アプリケーションに役立ちます。例えば、次のループは ループ・キャリー依存がないことを示す IVDEP ディレクティブの他に -ivdep_parallel を必要とします。

例

```
#pragma ivdep

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

```
   e[ix[2][i]]=e[ix[2][i]]+1.0;
   e[ix[3][i]]=e[ix[3][i]]+2.0;
}
```

次の例では、このオプションと IVDEP ディレクティブを使用すると、a() への格納でループキャリー依存が確実にないことを示します。

例

```
#pragma ivdep

for(j=0; j<n; j++)
{
   a[b[j]]=a[b[j]]+1;
}
```

並列プログラミング

概要: 並列プログラミング

並列プログラミング用に、インテル® C++ コンパイラは、OpenMP* 2.0 API と自動並列化機能をサポートします。次の表は、OpenMP および自動並列化を行うオプションを列挙したものです。

オプション	説明
-openmp	OpenMP ディレクティブに基づいてマルチスレッド・コードを生成する処理を、パラレライザに許可します。デフォルト: オフ
-openmp_report{0|1|2}	OpenMP パラレライザの診断レベルを制御します。デフォルト: -openmp_report1
-openmp_stubs	シーケンシャル・モードで OpenMP プログラムのコンパイルを有効にします。OpenMP ディレクティブは無視され、スタブ OpenMP ライブラリがリンクされます。デフォルト: オフ
-parallel	自動並列化を有効にして、並列で安全に実行できるループのマルチスレッド・コードを生成します。デフォルト: オフ
-par_threshold{n}	並列でのループの実行が効果的である可能性に基づいてループの自動並列化のしきい値を設定します (n=0 から 100)。n=0 は "常に" を意味します。デフォルト: -Qpar_threshold75
-par_report{0|1|2|3}	自動パラレライザの診断レベルを制御します。 デフォルト: -par_report1

注

-openmp と -parallel の両方がコマンドラインで指定されると、-parallel オプションは、OpenMP ディレクティブを含まないルーチンでのみ有効となります。OpenMP ディレクティブを含むルーチンでは、-openmp オプションのみが有効です。

ベクトル化 (IA-32 のみ)

概要: ベクトル化

ベクトライザはインテル® C++ コンパイラのコンポーネントです。C++ コンパイラは、MMX®、SSE、SSE2 命令セットにある SIMD 命令を自動的に使用します。ベクトライザは並列に実行できるプログラムの演算子を検出します。その後、シーケンシャル・

プログラムを変換し、データ型によって 1 回の演算で 2、4、8、16 要素のいずれかを処理します。

ここには、インテル C++ コンパイラによるベクトル化のガイドライン、オプション解説、および具体例を収録しました (IA-32 のみ)。内容は次のとおりです。

- ベクトル化の機能および特徴の早見表 (クイック・リファレンス)
- ベクトル化制御用コンパイラ・スイッチの解説
- ベクトル化制御用 C++ 言語機能の解説
- ベクトル化の各段階についての解説と一般的なガイドライン
  - 自動ベクトル化
  - ユーザの介入によるベクトル化
- ベクトル化を行うときの問題点とその解決方法の具体例

ベクトライザのオプション

オプション	説明
`-ax{K|W|N|B|P}`	ベクトライザを有効にし、専用 IA-32 コードと汎用 IA-32 コードを生成します。通常は、汎用コードのほうが専用コードよりも処理速度は遅くなります。
`-x{K|W|N|B|P}`	ベクトライザをオンにし、プロセッサ専用に特化されたコードを生成します。
`-vec_report`*n*	次のようにベクトライザの診断メッセージのレベルを制御します: • *n* =0 診断情報を表示しない。 • *n* =1 ループのベクトル化が成功したことを示す診断メッセージを表示する (デフォルト)。 • *n* =2 *n* =1 のメッセージに加えて、ループのベクトル化が失敗したことを示す診断メッセージも表示する。 • *n* =3 *n* =2 のメッセージに加えて、判明した依存関係または想定される依存関係についての補足情報も表示する。

使用方法

`-vec_report{n}` オプションと一緒に `-c`、`-ipo` を、または `-vec_report{n}` オプションと一緒に `-c`、`-x{K|W|N|B|P}` または `-ax{K|W|N|B|P}` を使用すると、コンパイラは警告メッセージを表示し、レポートは生成されません。

前述のオプションを使用してレポートを生成するには、`-ipo_obj` オプションを追加する必要があります。`-c` および `-ipo_obj` の組み合わせは、1 つのファイルをコンパイルし、オブジェクト・コードを生成して、最終的にレポートが生成されます。

次のコマンドは、ベクトル化レポートを生成します:

- prompt>**icpc -x{K|W|N|B|P} -vec_report3 file.cpp**
- prompt>**icpc -x{K|W|N|B|P} -ipo -ipo_obj -vec_report3 file.cpp**
- prompt>**icpc -c -x{K|W|N|B||P} -ipo -ipo_obj -vec_report3 file.cpp**

次のコマンドは、ベクトル化レポートを生成しません:

- prompt>**icpc -c -x{K|W|M|B|P} -vec_report3 file.cpp**
- prompt>**icpc -x{K|W|N|B|P} -ipo -vec_report3 file.cpp**
- prompt>**icpc -c -x{K|W|N|B|P} -ipo -vec_report3 file.cpp**

ループの並列化とベクトル化

-parallel オプションと -x{K|W|N|B|P} オプションを組み合わせると、同一のコンパイルで、自動ループ並列化と自動ループベクトル化の両方を試みるようにコンパイラに指示します。多くの場合、コンパイラは、並列化には最も外側のループ、ベクトル化には最も内側のループを認識します。しかし、有効であると判断された場合、コンパイラは、同じループに並列化とベクトル化を適用します。

ループ並列化 (自動または OpenMP* ディレクティブのいずれか) の成功は、コンパイラにレポートされるループベクトル化におけるメッセージに影響する場合があるので注意してください。例えば、-vec_report2 オプションでは、ループのベクトル化が成功しなかったことが示されます。

ベクトル化プログラミングの基本となるガイドライン

ベクトル化コンパイラの目標は、自動的に SIMD (single-instruction multiple data) 処理を活用することです。次に示したガイドラインと制約条件をよく確認し、先の項に示すコードの具体例を見て、それらに照らしてコードの良否を判断し、最適なベクトル化を阻害する曖昧な部分が見つかったらそれを直してください。

ループ本体に関するガイドライン:

- 直列型コード (単一基本ブロック) を使用する。
- ベクトルデータだけを使用する。つまり、代入式の右辺には配列と不変式を使用してください。代入式の左辺には配列参照を使用してもかまいません。
- 代入文だけを使用する。

ループ本体内では避けたほうがよいもの:

- 関数呼び出し
- ベクトル化できない演算
- ベクトル化できる複数の型を同じループの中に混在させること
- データ依存性を持つループ出口条件

ベクトル化できるコードにする方法

ベクトル化可能なコードに変えるには、ループを変更しなければならない場合がよくあります。ただし、変更する部分は、ベクトル化に最低必要なところだけとし、他の部分まで変更しないようにしてください。特に、次の項目に注意してください:

- ループのアンロールはしないこと。これはコンパイラが自動的に行います。
- ループの本体内に文がいくつか含まれている場合は、そのループを複数の単文ループに分解しないこと。

制約事項

ハードウェア。コンパイラは、それを動かすハードウェアからいくつか制約を受けます。ストリーミング SIMD 拡張命令を実行する場合、ベクトルメモリ演算は、16 バイトでアライメントしたメモリ参照が優先するため、アクセス・ストライドが `1` に制限されます。つまり、コンパイラは、たとえループをベクトル化可能と抽象的に認めたとしても、別のターゲット・アーキテクチャに対してはベクトル化をしないかもしれません。

コードの書きかた。ソースコードの書きかたによっては最適化の妨げになるときもあります。例えば、グローバル・ポインタを使用する場合に、2 つのメモリ参照が別々の場所で行われるかどうかをコンパイラが判定できないという問題がよく起こります。そうなると、一部の変換処理は順序の並べ替えができなくなります。

コンパイラによる自動ベクトル化を阻害する多くの要因はループ構造の書きかたにあります。ループ本体にはキーワード、演算子、データ参照、およびメモリ演算が含まれており、それらが互いに作用し合うためループの動作がよく見えなくなるのです。

しかし、これらの制約を理解し、診断メッセージの読みかたを知れば、そうした既知の制約条件を克服して効率よくベクトル化できるようプログラムを修正できます。以降の各項では、ループ構造についてベクトライザの機能と制約条件を簡単に述べます。

データ依存性

データ依存性とは、連続したいくつかのループに含まれている各演算の実行順序を制限する関係のことです。ベクトル化を行うと各演算の実行順序が変わるため、自動ベクトライザはデータ依存性の解析が自由にできる何らかの仕組みを持っていなければなりません。例として、データ依存性を持つコードを下に示します。この例に示した配列の各要素の値は、その要素自体とその要素の前後の要素によって決まります。

データ依存性を持つループ

```
float data[N];
int i;

for (i=1; i<N-1; i++)
{
   data[i]=data[i-1]*0.25+data[i]*0.5+data[i+1]*0.25;
}
```

上の例に示したループはベクトル化しません。なぜなら、現在の要素 data[i] に書き込まれる内容は、その前の反復のときに 1 つ前の要素 data[i-1] にどんなデータが書き込まれたかによって決まるからです。この例は、わかりやすくするために最初の 2 回の反復で配列がどのようにアクセスするかを表したものです:

データ依存性を持つループをベクトル化したもの

```
for(i=0; i<100; i++)
a[i]=b[i];
アクセス パターン
read b[0]
write a[0]
read b[1]
write a[1]
i=1: READ data[0]
READ data[1]
READ data[2]
WRITE data[1]
i=2: READ data[1]
READ data[2]
READ data[3]
WRITE data[2]
```

上に示したループを通常どおり逐次実行する場合、2 回目の反復のときに読み取られる data[1] の値は 1 回目の反復のときに書き込まれたものになります。ベクトル化を行うためには、元のループのセマンティクスを変えることなく、対象となるすべての反復を並列に実行しなければなりません。

データ依存性の理論

データ依存性の解析とは、2 つのメモリアクセスの重なり合う条件を見つけることです。その条件は、1 つのプログラムの中で参照を 2 回行うと仮定した場合は、次の 2 つの事項によって規定されます:

- 参照するいくつかの変数が、メモリ内の同じ領域のエイリアスであるかどうか (つまり、互いに重複しているかどうか)。
- 配列参照の場合は、添字同士の関連性

C++ コンパイラのデータ依存性解析機構で解析をするときは、使用する時間とメモリ領域を次第に増やしながら判定する一連の検定 (判定法) を実行します。どれか 1 つの次元の中にでも独立性が認められれば、それによって依存関係が排除できるため、最初は 1 次元ずつ単純な検定をいくつか実行します。宣言済みの次元境界にまたがる可能性のある多次元配列参照については、線形化された形に変換してから各検定を適用できます。この単純な検定の中には、高速 GCD 検定と拡張限界検定があります。高速 GCD 検定は、ループ添字のすべての係数の最大公約数を求め、その最大公約数で定数項を割り切れない場合を「独立性あり」と判定します。拡張限界検定は、添字式の極値同士の重なり合っていないのを判定します (GCD = greatest common divisor、最大公約数)。

どの単純な検定でも独立性を証明できなかった場合は、最終的に Fourier-Motzkin 法の消去を用いた強力な階層型依存性解法を使用して、すべての次元におけるデータ依存性問題を解きます。

ループの構成要素

ループは、一般的な `for` や `while` 要素で構成することができます。ただし、ループは、入口が 1 つだけでかつ出口が 1 つだけでなければ、ベクトル化できません。

正しい使用方法

```
while(i<n)
{
   // If branch is inside body of loop

   a[i]=b[i]*c[i];
   if(a[i]<0.0)
   {
      a[i]=0.0;
   }
   i++;
}
```

誤った使用方法

```
while(i<n)
{
   if (condition) break;
   // 2nd exit.
   ++i;
}
```

ループ出口条件

ループ出口条件とは、1 つのループの反復回数を決める条件のことです。for ループの場合は、固定インデックスによって反復回数が決まっています。ループの反復回数は数えられるものでなければなりません。つまり、反復回数を指定するときは次のいずれかを使用しなければなりません。

- 定数
- ループ不変項
- 最も外側のループ添字の線形関数

ループの出口が計算結果によって左右されるようなループは、その反復回数を数えられません。次の例は、可算/不可算ループの構成要素を表しています。

正しい使用方法 (可算ループ)

```
// Exit condition specified by "N-1b+1"
count=N;

...

while(count!=1b)
{
   // 1b is not affected within loop
   a[i]=b[i]*x;
   b[i]=[i]+sqrt(d[i]);
   --count;
}
```

正しい使用方法 (可算ループ)

```
// Exit condition is "(n-m+2)/2"
i=0;
for(l=m; l<n; l+=2)
{
   a[i]=b[i]*x;
   b[i]=c[i]+sqrt(d[i]);
   ++i;
}
```

誤った使用方法 (不可算ループ)

```
i=0;

// Iterations dependent on a[i]
while(a[i]>0.0)
{
   a[i]=b[i]*c[i];
   ++i;
}
```

ベクトル化ループの種類

整数ループの場合、MMX® テクノロジ命令およびストリーミング SIMD 拡張命令は、32 ビット、16 ビット、および 8 ビットの整数データ型での算術演算と論理演算のほとんどに各種 SIMD 命令を提供しています。整数丸め演算の最終的な精度を保持するのであればベクトル化できるときがあります。例えば、最終的に保存された値が 16 ビット整数の場合は、32 ビット右シフト演算子はベクトル化されません。また、MMX 命令セットとストリーミング SIMD 拡張命令セットは完全には直交していないため (例えば、バイトシフトに対応していない)、実際にはすべての整数演算がベクトル化できるわけではありません。

32 ビット単精度および 64 ビット倍精度の浮動小数点数を使用して演算を行うループの場合、ストリーミング SIMD 拡張命令は、+、-、*、/ という算術演算子に SIMD 命令を提供しています。また、ストリーミング SIMD 拡張命令は、MIN、MAX という二項演算子、および SQRT という単項演算子に SIMD 命令を提供しています。その他いくつかの算術演算子 (三角関数 SIN、COS、TAN など) の SIMD 版は、インテル® C++ コンパイラに付属しているベクトル算術演算用ランタイム・ライブラリ収録のソフトウェアで対応しています。

ストリップ・マイニングとクリーンアップ

ループのセクション化として知られるストリップ・マイニングは、メモリのパフォーマンスを改善する手段を提供し、ループの SIMD エンコーディングを可能にするループ変換テクニックです。大きなループをより小さなセグメントやストリップに断片化することで、このテクニックは次の 2 つの方法でループ構造を変更します:

- データがアルゴリズムの異なるパスで再使用可能な場合、データ・キャッシュ中の一時的な空間が増加します。
- 各 ″ベクトル″ の長さ、または SIMD 演算ごとに実行される演算の数により、ループの反復数は減ります。ストリーミング SIMD 拡張命令の場合、このベクトルまたはストリップの長さは 4 回 (1 つのストリーミング SIMD 拡張単精度浮

動小数点 SIMD 演算が処理されるごとに 4 つの浮動小数点データ項目が）減ります。

ベクトライザに最初に導入される場合、このテクニックは指定されたベクトルマシンの最大ベクトル長以下のサイズで各ベクトル演算が完了したときに生成されるコードからなります。

コンパイラは、自動的にループをストリップ・マイニングし、クリーンアップ・ループを生成します。

ベクトル化前

```
i=0;
while(i<n)
{
   // Original loop code
   a[i]=b[i]+c[i];
   ++i;
}
```

ベクトル化後

```
// The vectorizer generates the following two loops
i=0;

while(i<(n-n%4))
{
   // Vector strip-mined loop
   // Subscript [i:i+3] denotes SIMD execution
   a[i:i+3]=b[i:i+3]+c[i:i+3];
   i=i+4;
}

while(i<n)
{
   // Scalar clean-up loop
   a[i]=b[i]+c[i];
   ++i;
}
```

ループ本体内の文

ベクトル化可能な演算は、浮動小数点データと整数データで異なります。

浮動小数点配列の演算

ループ本体内の文には float 型の演算(通常は配列)を使用できます。可能な算術演算は、加算、減算、乗算、除算、否定、平方根、最大値、および最小値です。`-xW` ま

たは -axW コンパイラ・オプションを使用して、Pentium® 4 プロセッサ・システムに合わせて最適化しない限り、倍精度型の演算はできません。

整数配列の演算

ループ本体内の文には、char、unsigned char、short、unsigned short、int、および unsigned int という各型を使用できます。sqrt や fabs といった関数を呼び出せます。算術演算に使用できるのは、加算、減算、ビット単位 AND、ビット単位 OR、ビット単位 XOR、除算 (16 ビットのみ)、乗算 (16 ビットのみ)、最小値、および最大値だけです。データ型を複数混在させられるのは、変換しても精度が失われない場合だけです。例えば、乗算演算子、シフト演算子、単項演算子は混在させられます。

その他の演算

上記の浮動小数点演算と整数演算以外の文は使用できません。特に、特殊なデータ型である __m64 と __m128 はベクトル化できないので注意してください。ループ本体に関数呼び出しを含めることはできません。ストリーミング SIMD 拡張命令の組込み関数 (_mm_add_ps) は使用できません。

言語サポートとディレクティブ

ここでは、コードのベクトル化に役立つ言語機能について説明します。declspec(align(n)) 宣言は、ハードウェアのアライメント境界を克服します。restrict 指示子とプラグマは、語彙スコープ、データの依存性、両義性の解決のために書きかたに関する問題を処理します。

言語サポート

特徴	説明
__declspec(align(n))	変数を *n* バイト境界にアライメントするようコンパイラに命令します。変数のアドレスは *address* mod n=0 です。
__declspec(align(n,off))	各 *n* バイト境界内でオフセットだけ離して変数を *n* バイト境界にアライメントするようコンパイラに命令します。変数のアドレスは *address* mod n=off です。
restrict	エイリアス仮定を行うとき一義化機構に自由度を持たせます。これによってベクトル化の度合いがより高くなります。
__assume_aligned(a,n)	配列 a が n バイト境界にアライメントされ

	ていると見なすようコンパイラに指示します。アライメント情報が取得できなかった場合に使用します。
`#pragma ivdep`	ベクトル依存性が存在していると推定されてもそれを無視するようコンパイラに命令します。
`#pragma vector{aligned|unaligned|always}`	ループをベクトル化するときの方法を指定し、効率ヒューリスティックについては無視したほうがよいことを示します。
`#pragma novector`	ループをベクトル化しないよう指定します。

マルチバージョン・コード

マルチバージョン・コードは、不明な値のポインタにより、データの依存性の分析がループの独立性を証明できなかった場合に備えて、コンパイラによって生成されます。この機能は、動的依存性のテストと呼ばれます。

プラグマ・スコープ

これらのプラグマは、プログラム中の後に続くループのみのベクトル化を制御しますが、コンパイラはネストされたループには適用しません。各ネストされたループは、その前にプラグマが適用されるように、固有のプラグマが必要です。プラグマ は、loop 文の前にのみ配置してください。

#pragma vector always

構文: `#pragma vector always`

定義: このプラグマは、ベクトル化するかどうかを決めるとき、効率ヒューリスティックを無効にするようコンパイラに命令します。`#pragma vector always` は、1 外のストライドまたはほとんどアライメントの合っていないメモリアクセスをベクトル化します。

例:

```
for(i = 0; i <= N; i++)
{
   a[32*i] = b[99*i];
}
```

#pragma ivdep

構文: `#pragma ivdep`

定義: このプラグマは、ベクトル依存性が存在していると推定されてもそれを無視するようコンパイラに命令します。正しいコードにするため、コンパイラは、想定される依存性を証明された依存性として扱います。これは、ベクトル化を行わないようにします。このプラグマは、その決定を無視します。推定されたループの依存性が安全で、無視できる場合にのみ使用してください。

この例のループは、k の値が不明なため、ivdep プラグマではベクトル化を行いません (ベクトル化は k<0 の場合、不正です)。

例:

```
#pragma ivdep
for (i = 0; i < m; i++)
{
   a[i] = a[i + k] * c;
}
```

#pragma vector

構文: #pragma vector{aligned | unaligned}

定義: vector ループプラグマは、ループのベクトル化が有効である場合、通常のヒューリスティックな採算性についての判断を無視して、ベクトル化を行います。アライメントされた (またはアライメントされていない) 指示子がこのプラグマで使用されると、ループは、アライメントされた (またはアライメントされていない) 演算を使用して、ベクトル化されます。アライメントされた指示子かアライメントされていない指示子のいずれかを指定します。

警告

アライメントされているものを引数に指定する場合、ループはこの命令を使用してベクトル化可能であることが確実でなければなりません。それ以外の場合、コンパイラは誤ったコードを生成します。

次の例に示したループは、アライメントされた指示子を使用して、アライメントされた命令でループがベクトル化されるように要求を出します。コンパイラは通常そうすることが安全であると証明できないように、配列が宣言されているためです。

例:

```
void foo (float *a)
{
   #pragma vector aligned
   for (i = 0; i < m; i++)
   {
      a[i] = a[i] * c;

}
```

コンパイラは、コンパイル時にデータ構造のアライメントがわからない場合に備えて、いくつかのアライメント手法を用意しています。簡単な例を次に示します (しかし、いくつかの他の方法もサポートします)。次の例のループにおいて、a のアライメントが不明な場合、コンパイラはプレリュードのループを生成し、ほとんどの場合に発生する配列参照がアライメントされたアドレスにヒットするまでループを繰返します。これにより、a のアライメント・プロパティが判明し、そのプロパティに応じてベクトルループが最適化されます。

アライメント手法の例

```
float *a;
// alignment unknown
for (i = 0; i < 100; i++)
{
   a[i] = a[i] + 1.0f;
}

// dynamic loop peeling
p = a & 0x0f;
if (p != 0)
{
   p = (16 - p) / 4;

   {
      a[i] = a[i] + 1.0f;
   }
}

// loop with a aligned (will be vectorized accordingly)
for (i = p; i < 100; i++)
{
   a[i] = a[i] + 1.0f;
}
```

#pragma novector

構文: `#pragma novector`

定義: `novector` ループプラグマは、ループのベクトル化が有効な場合でもループをベクトル化しないことを示します。この例では、反復回数 `(ub - lb)` が低すぎて、ベクトル化が無駄になるとわかっています。ループがベクトル化可能であると認識されても、`#pragma novector` を使用して、コンパイラにベクトル化しないように指示できます。

例:

```
void foo (int lb, int ub)
{
   #pragma novector
   for (j = lb; j < ub; j++)
   {
      a[j] = a[j] + b[j];
   }
}
```

#pragma vector nontemporal

構文: `#pragma vector nontemporal`

定義: `#pragma vector nontemporal` は、Pentium® 4 プロセッサ・ベースシステムのストリーミング・ストアをもたらします。生成されたアセンブリとのループ (float type) の例は、次のとおりです。N が大きくなると、Pentium 4 プロセッサ・システムでの非ストリーミング組込み関数のパフォーマンスが大幅に向上します。

例:

```
#pragma vector nontemporal
for (i = 0; i < N; i++)
   a[i] = 1;
   .B1.2:
movntps XMMWORD PTR  a[eax], xmm0
movntps XMMWORD PTR _a[eax+16], xmm0
add eax, 32
cmp eax, 4096
jl .B1.2
```

動的依存性のテスト例

```
float *p, *q;
for (i = L; I <= U; i++)
{
   p[i] = q[i];
}
...
pL = p * 4*L;
pH = p + 4*U;
qL = q + 4*L;
qH = q + 4*U;
if (pH < qL || pL > qH)
{
   // loop without data dependence
   for (i = L; i <= U; i++)
   {
      p[i] = q[i];
   } else {
   for (i = L; i <= U; i++)
   {
      p[i] = q[i];
   }
}
```

ベクトル化の具体例

このセクションでは、ベクトル・プログラミングでよく起こる問題について簡単な例をいくつか挙げて説明します。

引数のエイリアシング: ベクトルコピー

下の例に示したループはベクトル・コピーを行うものですが、これはベクトル化されます。その理由は、コンパイラが dest[i] と src[i] との区別が付くのを証明できるからです。

ベクトル化可能なコピー操作 (区別が付くのを証明されないため)

```
void vec copy(float *dest, float *src, int len)
{
   int i;
   for(i=0; i<len; i++;)
   {
      dest[i]=src[i];
   }
}
```

下の例に示した restrict キーワードは、各ポインタが別々のオブジェクトを参照しているのを示します。したがって、マルチ・バージョン・コードを生成しなくてもベクトル化できます。

ベクトル化できるだけの違いがあることを証明するために restrict を使用する

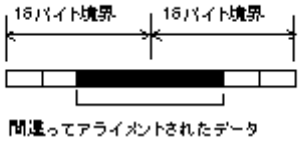
```
void vec copy(float *restrict dest, float *restrict src, int len)
{
   int i;
   for(i=0; i<len; i++)
   {
      dest[i]=src[i];
   }
}
```

データのアライメント

16 バイト以上のデータ構造体または配列は、その基底アドレスが 16 の倍数になる方法で各構造体や配列要素の先頭をアライメントしてください。

下の図は、アライメントの合っていないデータが原因でデータ・キャッシュ・ユニット (DCU) を分割した場合の影響を示したものです。アライメントの合っていないデータをロードすると 16 バイト境界にまたがるため、メモリアクセスが 1 回余分に発生し、その結果、6～12 サイクルのストールが発生します。データのアライメントの合っているのがわかっている場合や、アライメントされているものとして指定した場合は、このストールを避けられます。

16 バイト境界にまたがる間違ってアライメントされたデータ



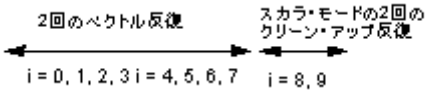
例えば、要素 a[0] と b[0] が 16 バイト境界にアライメントされているのがわかっている場合は、アライメント・オプション (#pragma vector aligned) をオンにすると、次のループはベクトル化できます:

ポインタのアライメントが確認済みの場合

```
float *a, *b;
int i;

for(int i=0; i<10; i++)
{
   a[i]=b[i];
}
```

ベクトル化すると、ループはこのように実行されます:

ベクトルおよびスカラのクリーンアップ反復処理



ベクトル反復処理である a[0:3] = b[0:3] と a[4:7] = b[4:7] は、要素 a[0] と b[0] (同様に a[4] と b[4]) が 16 バイトでアライメントされている場合には、アライメントの合った移動命令を使用して両方とも実装できます。

警告

不正なアライメント・オプションでベクトライザを指定すると、コンパイラの動作は予測がつかなくなります。特に、アライメントの合っていないデータを、アライメントの合っている移動命令で処理すると、不正命令例外が発生します。

データのアライメント例

下の例は、アライメントの合っていないメモリ命令でしかベクトル化しないループを示したものです。コンパイラはこのローカル配列をアライメントできますが、コンパイル時には lb の値がわからないため、アライメントが正しいかどうかの判断はできません。

コンパイル時に変数の値がわからないためにアライメントされないループ

```
void f(int lb)
{
   float z2[N], a2[N], y2[N], x2;
   for(i=lb; i<N; i++)
   {
      a2[i]=a2[i]*x2+y2[i];
   }
}
```

lb が 4 の倍数であることがわかっているときは、下の例に示したように #pragma vector aligned を使用してループをアライメントできます。

変数の値が 4 の倍数であるという前提でアライメントを行う

```
void f(int lb)
{
   float z2[N], a2[N], y2[N], x2;
   assert(lb%4==0);

   #pragma vector aligned

   for(i=lb; i<N; i++)
   {
      a2[i]=a2[i]*x2+y2[i];
   }
}
```

ループ交換と添字: マトリックス乗算

一般に、マトリックス乗算 (行列積) は下の例のように記述します:

一般的なマトリックス乗算

```
for(i=0; i<N; i++)
{
    for(j=0; j<n; j++)
    {
        for(k=0; k<n; k++)
        {
            c[i][j]=c[i][j]+a[i][k]*b[k][j];
        }
    }
}
```

b[k][j] の用法は、ストライドが 1 ではないため、通常はベクトル化できません。ただし、下の例に示すように、そのループを交換すると、すべてのストライドが 1 になります。

警告

依存関係のあるときは交換できないこともあり、交換してしまうと実行結果が異なる場合があります。

ストライドが 1 のマトリックス乗算

```
for(i = 0; i<N; i++)
{
    for(k=0; k<n; k++)
    {
        for(j=0; j<n; j++)
        {
            c[i][j]=c[i][j]+a[i][k]*b[k][j];
        }
    }
}
```

自動並列化

概要: 自動並列化

インテル® C++ コンパイラの自動並列化機能は、入力プログラムのシーケンシャル部分を同等のマルチスレッド・コードに自動的に変換します。自動パラレライザは、プログラムのループのデータフローを分析して、安全かつ効率的に並列実行可能なループに対するマルチスレッド・コードを生成します。これにより、SMP (対称型マルチプロセッサ) システムの並列アーキテクチャを活用できます。

自動並列化は次のようなユーザの負担を軽減します:

- ワークシェアリング候補であるループを見つけなければならない。
- 正しい並列実行を確認するためにデータフロー分析を行う。
- OpenMP ディレクティブのプログラミングに必要な場合、スレッドコード生成のデータをパーティショニングする。

並列ランタイム・サポートは、ループの反復修正、スレッド・スケジューリング、および同期化の詳細を処理するような、OpenMP* と同じランタイム機能を提供します。

OpenMP ディレクティブはシリアル・アプリケーションを素早く並列アプリケーションに変換することができますが、プログラマは、並列処理を含む適切なコンパイラ・ディレクティブを追加するアプリケーション・コードの特定部分を明示的に識別する必要があります。`-parallel` オプションで起動された自動並列化は、並列処理を含むループ構造を自動的に識別します。コンパイル中、コンパイラは、並列処理のためにコード・

シーケンスを別々のスレッドに自動的に分解しようと試みます。他にプログラマにかかる負荷はありません。

次の例は、2 つのスレッドで同時に実行できるようにループの反復を分割する方法を示します。

オリジナルの連続コード

```
for (i=1; i<100; i++)
{
  a[i] = a[i] + b[i] * c[i];
}
```

変換された並列コード

```
/* Thread 1 */
for (i=1; i<50; i++)
{
  a[i] = a[i] + b[i] * c[i];
}

/* Thread 2 */
for (i=50; i<100; i++)
{
  a[i] = a[i] + b[i] * c[i];
}
```

自動並列化のプログラミング

自動並列化機能は、(parallel for ディレクティブにより) ワークシェアリング構造などの OpenMP* のいくつかのコンセプトを取り入れています。ここでは、自動並列化について説明します。

効率的な自動並列化の使用方法のガイドライン

次の場合、ループは並列化が可能です:

- ループがコンパイル時にカウント可能な場合。これは、ループの実行回数 (ループ反復回数) を表す式が、ループに入る直前に生成されることを意味します。
- FLOW (WRITE の後の READ)、OUTPUT (READ の後の WRITE)、または ANTI (READ の後の WRITE) ループ・キャリー・データ依存性がない場合。同じメモリの場所が、ループの異なる反復で参照されるときに、ループ・キャリー・データ依存が起こります。コンパイラの判断で、推測されたループキャリー依存性がランタイム依存性のテストで解決されると、ループは並列化されることがあります。

コンパイラは、コンパイル時に定数ではないループ・パラメータを持つ `parallel for` ループで、実行するメリットを検証するためにランタイム・テストを生成します。

コーディング・ガイドライン

次のコーディング・ガイドラインにより自動並列化の威力と効率を強化することができます。

- 可能な限りループの反復回数を明確化してください。特に反復回数が既知の場合は定数を使用し、ループ・パラメータはローカル変数に保存してください。
- コンパイラがキャリー依存データと判断する可能性のある構造（関数呼び出し、不明瞭な間接参照、グローバル参照など）をループ本体内に配置しないでください。

自動並列化のデータフロー

自動並列化の処理では、コンパイラは次のステップを実行します。

1. データフローの解析
2. ループの分類
3. 依存性の解析
4. 高度な並列化
5. データのパーティショニング
6. マルチスレッド・コードの生成

これらのステップには次のものが含まれます:

- データフローの解析: プログラムを通してデータのフローを計算します。
- ループの分類: しきい値解析で示されるように 正確さと効率に基づいて並列化のループの候補を決定します。
- 依存性の解析: 各ループのネストで参照における依存性の解析を計算します。
- 高度な並列化
  - 依存性のグラフを解析し、並列で実行できるループを決定します。
  - ランタイムの依存性を計算します。
- データのパーティショニング: `shared`、`private`、および `firstprivate` のアクセスのタイプに基づいて、データ参照とパーティションを検査します。
- マルチスレッド・コードの生成:
  - ループのパラメータを修正します。
  - スレッドタスクごとに入口/出口を生成します。
  - スレッド生成と同期化の並列ランタイム・ルーチンへの呼び出しを生成します。

自動並列化: 有効、オプション、および環境変数

自動パラレライザを有効にするには、-parallel オプションを使用します。-parallel オプションは、並列で安全に実行できる並列ループを検出して自動的にこれらのループのマルチスレッド・コードを生成します。次に、自動並列化を使用するコマンドの例を示します:

```
prompt>icpc -c -parallel prog.cpp
```

自動並列化のオプション

-O2 (または-O3) 最適化オプションがオン (デフォルトは -O2) の場合、-parallel オプションは自動並列化を有効にします。-parallel オプションは、並列で安全に実行できる並列ループを検出して自動的にこれらのループのマルチスレッド・コードを生成します。

オプション	説明
-parallel	自動パラレライザを有効にします。
-parallel_threshold{1-100}	自動並列化に必要な作業しきい値を制御します。詳細は、後のサブセクションを参照してください。
-par_report{1|2|3}	自動並列化の診断メッセージを制御します。詳細は、後のサブセクションを参照してください。

自動並列化の環境変数

変数	説明	デフォルト
OMP_NUM_THREADS	使用されるスレッド数を制御します。	実行ファイルを生成する際にシステムに現在搭載されているプロセッサ数
OMP_SCHEDULE	ランタイム・スケジューリングのタイプを指定します。	static

自動並列化のしきい値と診断

しきい値制御

-par_threshold{n} オプションは、並列ループ実行の有効性に基づいて、ループの自動並列化のしきい値を設定します。n の値は 0 から 100 まで設定できます。デフォルト値は 75 です。このオプションは、コンパイル時に計算量が確定できないループに使用します。しきい値は、通常、ループ反復回数がコンパイル時に不明なときに関係します。

-par_threshold{n} オプションには次のバージョンと機能があります:

- デフォルト: -par_threshold はコマンドラインでは、指定されません。これは、-par_threshold0 が指定されるときと同様です。ループは、計算量に関わらず自動並列化を行います。つまり、常に並列化します。
- -par_threshold100 – ループは並列化実行が有効であることが確実な場合にのみ自動並列化されます。
- 1 から 99 の値は、有効な速度の向上の可能性の比率を表します。例えば、n=50 の場合、並列実行された場合にコードの速度が向上する可能性が 50 パーセントの場合に並列化を行います。
- デフォルト値は、n=75 (または -par_threshold75) です。-par_threshold を数字の指定なしで、コマンドラインで使用すると、デフォルト値は 75 になります。

コンパイラは、作成された複数のスレッドのオーバーヘッドとスレッド間を共有できるワーク量のバランスをとろうとするヒューリスティックスを適用します。

診断

-par_report{0|1|2|3} オプションは、自動パラレライザの診断レベル 0、1、2、3 を次のように制御します:

- -par_report0 = 診断情報を表示しません。
- -par_report1 = ループが正常に自動並列化されたことを示します (デフォルト)。並列ループに対して "LOOP AUTO-PARALLELIZED" メッセージを出力します。
- -par_report2 = 正常に自動並列化されたループおよび不正に自動並列化されたループを表示します。
- -par_report3 = 2 に加えて、自動並列化を妨げる実証された依存および推測された依存についての追加情報を示します。

並列化診断レポートの例

次の例は、-par_report3 で生成された出力結果です:

```
prompt>icpc -c -parallel -par_report3 prog.cpp
```

サンプル出力

```
program prog
procedure: prog
serial loop: line 5: not a parallel candidate due to
statement at line 6
serial loop: line 9
flow data dependence from line 10 to line 10, due to "a"
12 Lines Compiled
```

プログラム prog.cpp は次のとおりです:

prog.cpp のサンプル

```
/* Assumed side effects */

for (i=1; i<10000; i++)
{
  a[i] = foo(i);
}

/* Actual dependence */

for (i=1; i<10000; i++)
{
  a[i] = a[i-1] + i;
}
```

トラブルシューティングのヒント

- コンパイラが十分な計算量がないことを推定したかどうかを確認するには、-par_threshold0 を使用します。
- 診断結果を表示するには、-par_report3 を使用します。
- 推定される関数の呼び出しへの副作用を回避するには、-ipo を使用します。

OpenMP* による並列化

概要: OpenMP* による並列化

インテル® C++ コンパイラは、OpenMP* C++ バージョン 2.0 API 仕様をサポートします。OpenMP は、次の主な機能を持った対称型マルチプロセッシング (SMP) を提供します:

- 反復のパーティショニング、データの共用、スレッドのスケジューリング、および同期化に関する下位の詳細レベルを処理して、ユーザの負担を軽減します。
- 共有メモリ、マルチプロセッサ・システムの場合に得られるパフォーマンスを提供します。

インテル C++ コンパイラは、ソース・プログラムのユーザの OpenMP ディレクティブの指定に従って、コード変換を実行し、マルチ・スレッド・コードを生成して、既存のソフトウェアへスレッドを追加しやすくします。インテル・コンパイラは、現在の業界標準の OpenMP ディレクティブのすべてに対応しています。ただし、WORKSHARE および OpenMP ディレクティブの注釈のある並列実行プログラムのコンパイラは除きます。さらに、インテル C++ コンパイラは、ランタイム・ライブラリ・ルーチンおよび環境変数を含む OpenMP C++ バージョン 2.0 仕様にインテル独自の拡張機能を提供します。

注

コンパイラの他の高度機能と同じように、OpenMP ディレクティブを効果的に使用し、プログラムの予期しない動作を避けるためには、OpenMP ディレクティブの機能を正しく理解する必要があります。

インテル C++ コンパイラの OpenMP 機能の全オプションについては、「概要: 並列プログラミング」を参照してください。

OpenMP 標準の詳細については、Web サイト http://www.openmp.org をご覧ください。OpenMP* C++ バージョン 2.0 API 仕様については、http://www.openmp.org/specs/ を参照してください。

OpenMP による並列処理

OpenMP でコンパイルするには、OpenMP ディレクティブでコードを注釈するプログラムを準備する必要があります。インテル C++ コンパイラは、はじめにアプリケーションを処理して、コードのマルチスレッド・バージョンを生成してからコードをコンパイルします。その出力は、並列領域または構造を実行するスレッドによって実装される並列処理の実行プログラムです。

プロセッサのランタイム・チェック

ループを並列化するとき、インテル・コンパイラのループ・パラレライザ、OpenMP は指定されたプロセッサ向けの最適なセットの構成を決定しようとします。プログラムの実行時に、どの IA32 プロセッサの OpenMP がループを最適化するべきかを決定するチェックが行われます。詳細は、「プロセッサ固有のランタイム・チェック、IA-32 システム」を参照してください。

パフォーマンス分析

プログラムのパフォーマンス分析には、パフォーマンス情報を表示する「インテル® VTune™ パフォーマンス・アナライザ」を使用してください。コードのどの部分が最も多く実行時間を必要としているか、また並列パフォーマンスの問題個所の特定に関する詳細情報が得られます。

並列処理スレッドモデル

ここでは、並列化されたプログラム処理の説明と、並列プログラミングで使用される用語の定義を追加説明します。

実行フロー

前述したように、OpenMP* の C++ API コンパイラ・ディレクティブを持つプログラムは、単一のプロセスとして実行を開始します。これは、**マスタ**スレッドの実行と呼ばれます。最初の**並列構造**が検出されるまで、マスタスレッドは、シーケンシャルに実行します。

OpenMP C++ API では、`#pragma omp parallel` ディレクティブは、並列構造を定義します。マスタスレッドが並列構造を検出すると、そのマスタスレッドがチームのマスタとなるように、スレッドの**チーム**を作成します。並列構造に囲まれたプログラム文は、チーム内の各スレッドごとに並列して実行されます。これらの文は、囲まれた文の中から呼び出されるルーチンを含みます。

構造内で字句的に囲まれた文は、構造の**静的範囲**を定義します。**動的範囲**は、構造内から呼び出されるルーチンと同様に静的範囲を含みます。`#pragma omp parallel` ディレクティブが終了すると、チーム内のスレッドは同期化され、そのチームは消滅して、マスタスレッドのみが実行し続けます。チーム内の他のスレッドは、待機状態になります。単一プログラム内で並列構造は何回でも指定できます。結果として、プログラム実行中に、スレッドのチームは何度も生成され消滅します。

孤立ディレクティブの使用

並列構造内で呼び出されたルーチンで、ディレクティブを使用することができます。並列構造の字句範囲ではなく、動的範囲のディレクティブは、孤立ディレクティブと呼ばれます。孤立ディレクティブは、プログラムのシーケンシャル・バージョンに最小限の変更を行うだけで、プログラムの主要部分を並列に実行できます。この機能を使用すると、プログラムの最上位レベルで並列構造をコーディングでき、ディレクティブを使用して呼び出されるすべてのルーチンの実行を制御することができます。
例:

```
int main(void)
{
   ...
   #pragma omp parallel
   {
      phase1();
   }
}

void phase1(void)
{
   ...
   #pragma omp for private(i) shared(n)
   for(i=0; i < n; i++)
   {
      some_work(i);
   }
}
```

並列範囲が字句的に指定されていないので、これが孤立ディレクティブとなります。

データ環境ディレクティブ

データ環境ディレクティブは、並列構造の実行中にデータ環境を制御します。並列およびワークシェアリング構造内でデータ環境を制御できます。ディレクティブとディレクティブのデータ環境節を使用して次のことが可能です:

- THREADPRIVATE ディレクティブを使用して、スコープ変数をプライベート化します。
- THREADPRIVATE ディレクティブの節を使用して、データスコープ属性を制御します。データスコープ属性節:
  - COPYIN
  - DEFAULT
  - PRIVATE
  - FIRSTPRIVATE
  - LASTPRIVATE
  - REDUCTION
  - SHARED

複数のディレクティブ節を使用して、変数のデータスコープ属性を指定した構造の継続期間中にその属性を制御することができます。ディレクティブでデータスコープ属性節を指定しない場合、ディレクティブに影響を受ける変数のデフォルトは SHARED になります。

並列処理モデルの疑似コード

一般的な OpenMP ディレクティブをいくつか使用した疑似プログラムのサンプルには、次のものがあります。この例ではまた、シリアル領域と並列領域の相違も示しています。

```
main() {                 // Begin serial execution
 ...                     // Only the master thread executes
#pragma omp parallel     // Begin a Parallel Construct, form
 {                       // a team. This is Replicated Code
  ...                    // (each team member executes
  ...                    // the same code)
                         //
#pragma omp              // Begin a Worksharing Construct
sections
 {                       //
  #pragma omp section    // One unit of work
   {...}                 //
```

```
  #pragma omp section    // Another unit of work
   {...}                 //
 }                       // Wait until both units of work complete
 ...                     // More Replicated Code
                         //
 #pragma omp for nowait  // Begin a Worksharing Construct;
  for(...) {             // each iteration is unit of work
                         //
   ...                   // Work is distributed among the team
                         members
                         //
 }                       // End of Worksharing Construct;
                         // nowait was specified, so
                         // threads proceed
                         //
 #pragma omp critical    // Begin a Critical Section
 {                       //
  ...                    // Replicated Code, but only one
                         // thread can execute it at a
 }                       // given time
 ...                     // More Replicated Code
                         //
 #pragma omp barrier     // Wait for all team members to arrive
 ...                     // More Replicated Code
                         //
}                        // End of Parallel Construct;
                         // disband team and continue
                         // serial execution
                         //
...                      // Possibly more Parallel constructs
                         //
}                        // End serial execution
```

OpenMP、ディレクティブ形式、および診断でのコンパイル

OpenMP* モードでインテル® C++ コンパイラを実行するには、`-openmp` オプションでコンパイラを起動します:

```
prompt>icpc -openmp file.cpp
```

マルチスレッド・コードを起動する前に、OpenMP 環境変数 OMP_NUM_THREADS で使用するスレッドの数を設定できます。詳細については、「OpenMP の環境変数」を参照してください。

-openmp オプション

-openmp オプションは、パラレライザが OpenMP ディレクティブに基づいてマルチ・スレッド・コードを生成できるようにします。このコードは、単一プロセッサ・システムとマルチプロセッサ・システムのいずれでも並列実行が可能です。-openmp オプションは、-Od (最適化なし) と -O1、-O2 (デフォルト) および -O3 の最適化レベルで動作します。-openmp と -O0 を指定すると、OpenMP アプリケーションのデバッグに役立ちます。

OpenMP ディレクティブ形式と構造

OpenMP ディレクティブの形式は次のとおりです:

```
#pragma omp directive-name [clause, ...] newline
```

各アイテムの意味は次のとおりです:

- #pragma omp -- すべての OpenMP ディレクティブに必須。
- directive-name -- 有効な OpenMP ディレクティブ。pragma の後および節の前に出力する必要があります。
- clause -- オプション。clause は順番に関係なく指定でき、制限されていない限り必要に応じて繰り返すことができます。
- newline -- 必須。このディレクティブに囲まれた構造ブロックを続行します。

OpenMP 診断

-openmp_report{0|1|2} オプションは、OpenMP パラレライザの診断レベル 0、1、2 を次のように制御します:

- -openmp_report0 = 診断情報を表示しません。
- -openmp_report1 = 正常に並列化されたループ、領域、およびセクションを示す診断を表示します。
- -openmp_report2 = 上記の -openmp_report1 の診断に加えて、正常に処理された MASTER 構造、SINGLE 構造、CRITICAL 構造、ORDERED 構造、ATOMIC ディレクティブなどを示す診断を表示します。

デフォルトは -openmp_report1 です。

OpenMP* ディレクティブと節

OpenMP ディレクティブ

ディレクティブ名	説明
parallel	並列実行領域を定義します。
for	関連付けられたループの反復が並列実行される領域を指定する、反復的なワークシェアリングの構造を識別します。
sections	チーム内のスレッド間で分割される一連の構造を指定する、非反復的なワークシェアリングの構造を識別します。
single	対応する構造化ブロックがチーム内の 1 つのスレッドだけで実行されるように指定する構造を識別します。
parallel for	1 つの for ディレクティブを含む並列領域のショートカット。OpenMP ディレクティブ parallel または for の直後には、for 文を続けなければなりません。parallel または for ディレクティブと for 文の間に、他の文または OpenMP ディレクティブがあると、インテル® C++ コンパイラは構文エラーを出力します。
parallel sections	1 つの sections ディレクティブを含む並列領域を指定するショートカット形式。
master	チームの master スレッドで実行する構造ブロックを指定する構成体を示します。
critical[lock]	関連する構造ブロックの実行を一度に 1 スレッドだけに制限する構成体を示します。
barrier	チーム内のすべてのスレッドを同期化します。
atomic	特定のメモリ・ロケーションをアトミックに更新します。
flush	"クロススレッド" シーケンス・ポイントを指定します。このポイントでは、チーム内のすべてのスレッドから見たメモリ内の特定のオブジェクトの状態の整合性が保たれるように、プログラム上で保証する必要があります。
ordered	ordered ディレクティブに続く構造ブロックを、シーケンシャル・ループ内で反復が実行される順序で実行します。
threadprivate	指定された名前付きのファイル有効範囲変数または名前空間の有効範囲変数を特定のスレッドに対してプライベートにし、そのスレッド内ではファイル有効範囲を参照可能にします。

OpenMP の節

節	説明
private	チーム内の各スレッドに対して private になるように変数を宣言します。
firstprivate	private 節で指定される機能のスーパセットを指定します。

lastprivate	private 節で指定される機能のスーパセットを指定します。
shared	チーム内のすべてのスレッドで変数を共用します。
default	変数のデータ有効範囲属性を設定できます。
reduction	スカラ変数の削減を実行します。
ordered	ordered ディレクティブに続く構造ブロックを、シーケンシャル・ループ内で反復が実行される順序で実行します。
if	if(scalar_logical_expression) 節が存在する場合、scalar_logical_expression 節が真である場合にのみ、囲まれたコードブロックは並列に実行されます。 それ以外の場合は、コードブロックは直列に実行されます。
schedule	for ループの反復がチームのスレッド間でどのように分割するかを指定します。
copyin	並列領域を実行しているチーム内の各スレッドの threadprivate 変数に、同じ名前を割り当てます。

OpenMP* のサポート・ライブラリ

OpenMP* をサポートするインテル® C++ コンパイラは、製品サポート・ライブラリ libguide.lib を提供します。このライブラリを使用して、アプリケーションを異なる実行モードで実行することができます。これは、既にチューニングされているアプリケーションによる通常実行またはパフォーマンスが重要な実行において使用します。

注

libguide.lib ライブラリは、デバッグすることが難しいパフォーマンス問題を回避するために、コマンドライン・オプションに関係なく、ダイナミックにリンクされます。

実行モード

OpenMP をサポートするインテル・コンパイラは、ランタイム時に指定した実行モードでアプリケーションを実行することができます。このライブラリは、シリアル (serial)、ターンアラウンド (turnaround) およびスループット (throughput) モードをサポートします。これらのモードは、実行時に KMP_LIBRARY 環境変数を使用することによって選択されます。

シリアル

シリアルモードは、並列アプリケーションをシングル・プロセッサ上で強制的に実行します。

ターンアラウンド

すべてのプロセッサが、プログラムの全実行に対し排他的に割り当てられる専用 (バッチまたはシングルユーザ) 並列環境では、常にすべてのプロセッサを効果的に利用することが最も重要です。ターンアラウンド・モードは、並列計算を行うすべてのプロセッサをアクティブな状態で維持して、単一ジョブの実行時間を最小限に抑えるよう設計されています。作業スレッドは、追加の並列作業を他のスレッドにわたすことなく、アクティブな状態で待機します。

注

過剰なシステムリソースの割り当てを避けてください。過剰なシステムリソースの割り当ては、スレッドが多すぎるか、または実行時に利用可能なプロセッサが少なすぎる場合に発生します。システムリソースが過剰に割り当てられると、このモードはパフォーマンスの低下を起こします。この問題が発生した場合、スループット・モードを使用してください。

スループット

並列マシン上のロードが一定ではない、またはジョブ・ストリームが予測できないマルチユーザ環境下では、スループット用にデザインおよびチューニングする方が良い場合もあります。これにより、複数のジョブを同時に実行した際の合計時間を最小限に抑えることができます。このモードでは、作業スレッドは追加の並行作業の待機中、他のスレッドへ作業を渡します。

スループット・モードは、プログラムにその実行環境を認知させ (つまりシステムの読み込み)、リソースの使用を調整することで、動的環境における効率の良い実行を行えるよう設計されています。スループット・モードはデフォルトです。

OpenMP* の環境変数

このトピックでは、OpenMP* の環境変数 (OMP_ プリフィックス付き) およびインテル固有の環境変数 (KMP_ プリフィックス付き) を説明します。

標準環境変数

変数	説明	デフォルト
OMP_SCHEDULE	ランタイム・スケジュールの型とチャンク・サイズを設定します。	STATIC (ブロックサイズの指定なし)
OMP_NUM_THREADS	実行時に使用するスレッド数を設定します。	プロセッサの数

OMP_DYNAMIC	スレッド数の動的な調整を有効 (TRUE) または無効 (FALSE) にします。	FALSE
OMP_NESTED	ネストされた並列処理を有効 (TRUE) または無効 (FALSE) にします。	FALSE

インテル拡張環境変数

環境変数	説明	デフォルト
KMP_LIBRARY	OpenMP ランタイム・ライブラリ・スループットを選択します。この変数の値には、実行モードを示す serial、turnaround または throughput があります。この変数が指定されなかった場合、デフォルト値の throughput が使用されます。	throughput (実行モード)
KMP_STACKSIZE	各並行スレッドがプライベート・スタックとして使用するバイト数を設定します。オプションの b、k、m、g または t サフィックスを使用して、確保するバイト数をバイト、キロバイト、メガバイト、ギガバイトまたはテラバイトで指定してください。	IA-32: 2m Itanium® コンパイラ: 4m

OpenMP* ランタイム・ライブラリ・ルーチン

OpenMP* では、並列モードでプログラムを管理するためのランタイム・ライブラリ・ルーチンをいくつか用意しています。これらの関数の多くは、デフォルトとして設定できる環境変数を持っています。これらの環境変数はランタイム・ライブラリ関数によって、動的に変更できます。いかなる場合も、ランタイム・ライブラリ・ルーチンが呼び出されると、それに対応する環境変数は無効になります。

次の表は、これらランタイム・ライブラリ・ルーチンとのインターフェイスを明記したものです。ルーチン名はユーザ名前空間に保存しています。コンパイラをインストールした INCLUDE ディレクトリに omp.h および omp_lib.h ヘッダファイルがあります。

この表中の関数に使用する 2 つの異なるロック (lock) である omp_lock_kind と omp_nest_lock_kind には、定義がいくつかあります:

実行環境ルーチン

関数	説明
omp_set_num_threads(*nthreads*)	後続の並列領域に使用するスレッド数を設定します。
omp_get_num_threads()	現在の並列領域に使用されているスレッドの数を返します。
omp_get_max_threads()	並列実行に利用可能なスレッドの最大数を返しま

	す。
omp_get_thread_num()	コードのこのセクションを現在実行しているスレッドはどれか、その固有番号を返します。
omp_get_num_procs()	プログラムに利用可能なプロセッサの数を返します。
omp_in_parallel()	並列で実行する並列領域の動的範囲内で呼び出された場合は TRUE を返します。そうでない場合は FALSE を返します。
omp_set_dynamic(*dynamic_threads*)	並列領域の実行に使用するスレッド数の動的な調整を有効または無効にします。*dynamic_threads* が TRUE の場合は、ダイナミック・スレッドは有効です。*dynamic_threads* が FALSE の場合は、ダイナミック・スレッドは無効です。Dynamics threads はデフォルトでは無効です。
omp_get_dynamic()	動的なスレッド調整が有効の場合は、TRUE を返します。そうでない場合、FALSE を返します。
omp_set_nested(*nested*)	ネストされた並列処理を有効または無効にします *nested* が TRUE の場合は、ネストされた並列処理は有効です。*nested* が FALSE の場合は、ネストされた並列処理は無効です。ネストされた並列処理はデフォルトでは無効です。
omp_get_nested()	ネストされた並列処理が有効な場合は、TRUE を返します。そうでない場合は、FALSE を返します。

ロックルーチン

関数	説明
omp_init_lock(*lock*)	後続の呼び出しに使用する *lock* に関連付けられたロックを初期化します。
omp_destroy_lock(*lock*)	*lock* に関連付けられたロックを未定義にします。
omp_set_lock(*lock*)	*lock* に関連付けられているロックが使用可能な状態になるまで実行中のスレッドを強制的に待機させます。ロックが使用可能になると、スレッドにはそのロックの所有権が与えられます。
omp_unset_lock(*lock*)	*lock* に関連付けられているロックの所有権から実行スレッドを解放します。*lock* に関連付けられたロックを実行中のスレッドが所有していない場合の動作は不定です。
omp_test_lock(*lock*	*lock* に関連付けられているロックを設定しようと試みます。成功した場合は、TRUE を返します。そうでない場合は、FALSE を返します。
omp_init_nest_lock(*lock*)	後続の呼び出しに使用する *lock* に関連付けられたネストされたロックを初期化します。

omp_destroy_nest_lock(*lock*)	*lock* に関連付けられたネストされたロックを未定義にします。
omp_set_nest_lock(*lock*)	*lock* に関連付けられているネストされたロックが使用可能な状態になるまで実行中のスレッドを強制的に待機させます。ロックが使用可能になると、スレッドにはそのネストされたロックの所有権が与えられます。
omp_unset_nest_lock(*lock*)	ネストしているカウント数がゼロの場合は、*lock* に関連付けられたネストされたロックの所有権から実行中のスレッドを解放します。*lock* に関連付けられたネストされたロックを実行中のスレッドが所有していない場合の動作は不定です。
omp_test_nest_lock(*lock*)	*lock* に関連付けられているネストされたロックを設定しようと試みます。成功した場合はネスト数を返し、失敗した場合は 0 を返します。

タイミング・ルーチン

関数	説明
omp_get_wtime()	任意の参照時間から経過したウォールクロック時間 (秒) に等しい倍精度値を返します。参照時間は、プログラム実行中には変更されません。
omp_get_wtick()	連続するクロック刻みの間隔の秒数に等しい倍精度値を返します。

OpenMP* の使用例

次の例に、OpenMP 機能の使用方法を示します。

単純な差分演算子

この例は、各繰返しごとにワーク量が異なる単純な並列ループを示したものです。負荷のバランスをとるために、ダイナミック・スケジューリングを使用しています。並列領域の最後に暗黙的なバリアがあるため、for に nowait が含まれています。

```
void for 1 (float a[], float b[], int n)
{
  int i, j;
  #pragma omp parallel shared(a,b,n) private(i,j)
  {
      #pragma omp for schedule(dynamic,1) nowait
          for(i = 1; i < n; i++)
          {
             for(j = 0; j <= i; j++)
             b[j + n*i] = (a[j + n*i] + a[j + n*(i-1)])/2.0;
          }
   }
}
```

2 つの差分演算子

下記の例では、fork/join のオーバーヘッドを減らすために融合される 2 つの並列ループを使用します。2 つ目のループで使用するすべてのデータは最初のループで使用されるすべてのデータと異なるため、最初の `for` に `nowait` が含まれています。

```
void for_2 (float a[], float b[], float c[], \
float d[], int n, int m)
{
  int i, j;
  #pragma omp parallel shared(a,b,c,d,n,m) private(i,j)
  {
    #pragma omp for schedule(dynamic,1) nowait
    for(i = 1; i < n; i++)
    {
      for(j = 0; j <= i; j++)
      b[j + n*i] = ( a[j + n*i] + a[j + n*(i-1)] )/2.0;
    }

    #pragma omp for schedule(dynamic,1) nowait
    for(i = 1; i < m; i++)
    {
      for(j = 0; j <= i; j++)
      d[j + m*i] = ( c[j + m*i] + c[j + m*(i-1)] )/2.0;

  }
}
```

OpenMP* に追加されたインテル拡張機能

概要: インテルのワークキューイング・モデル

ワークキューイング・モデルでは、OpenMP* モデルにサポートされる制御構造の範囲を超えた制御構造を並列化できます。同時に OpenMP で定義されるフレームワークに合うようにします。特に、ワークキューイング・モデルは、ワークシェアリング構造の開始時に作業単位が事前計算されないように指定する柔軟性のあるメカニズムです。 `single`、`for` および `sections` 構造では、構造が実行を開始する時点で、実行可能なすべての作業単位が判明しています。ワークキューイング・プラグマ taskq と task は、環境 (taskq) と作業単位 (task) を別々に指定することによって、この制限を緩和します。

インテル拡張機能

インテル® C++ コンパイラは、OpenMP* ランタイム・ライブラリへの拡張機能として、次の関数グループをサポートします:

- 並列スレッドのスタックサイズの取得と設定

- メモリの割り当て

ここで説明するインテル拡張機能は、ライブラリ・コードとアプリケーションが目的どおりに機能することを確認する低レベルのデバッグに使用できます。これらの関数を使用するには、プログラムをシーケンシャルに実行する `-openmp_stubs` コマンドライン・オプションを使用しなければならないため、充分注意して使用してください。これらの関数はまた、一般的に他のベンダの OpenMP 互換コンパイラには認識されません。これらのコンパイラでは、リンクの段階で失敗します。

注

以下の関数は、プリプロセッサ・ディレクティブ `#include <omp.h>` を必要とします。

スタックサイズ

多くの場合、ディレクティブは拡張命令の代わりに使用されます。たとえば、並列スレッドのスタックサイズは、`kmp_set_stacksize_s()` 関数ではなく、`KMP_STACKSIZE` 環境変数を使用して設定します。

注

インテル拡張機能へのランタイムの呼び出しは、対応する環境変数の設定よりも優先します。下の「スタックサイズ」表でスタックサイズ関数の定義を参照してください。

メモリの割り当て

インテル C++ コンパイラは、OpenMP ランタイム・ライブラリに対する拡張機能として、メモリ割り当て関数を実装しています。そのため、スレッドは各スレッドにローカルなヒープからメモリを割り当てることが可能です。これらの関数は、`kmp_malloc()`、`kmp_calloc()` および `kmp_realloc()` です。これらの関数によって割り当てられたメモリは、`kmp_free()` 関数によって解放する必要があります。あるスレッドによってメモリを割り当て、別のスレッドでメモリを `kmp_free()` を呼び出しても不正な処理ではありませんが、このような処理によってパフォーマンスが多少低下します。下の「メモリの割り当て」表でこれらの関数の定義を参照してください。

スタックサイズ

関数	説明
`kmp_get_stacksize_s()`	各並列スレッドがプライベート・スタックとして使用するバイト数を返します。この値は、最初の並列領域の前に `kmp_set_stacksize_s()` で変更するか、または `KMP_STACKSIZE` 環境変数で変更できます。

kmp_get_stacksize()	この関数は、下位互換性のみ提供します。異なるインテル® プロセッサとの互換性には kmp_get_stacksize_s() を使用します。
kmp_set_stacksize_s(*size*)	各並列スレッドがプライベート・スタックとして使用するバイト数を *size* に設定します。この値は、KMP_STACKSIZE 環境変数で設定することもできます。kmp_set_stacksize_s() を有効にするには、プログラムの最初の (動的に実行された) 並列領域の先頭の前に呼び出す必要があります。
kmp_set_stacksize(*size*)	この関数は、下位互換性のみ提供します。異なるインテル・プロセッサとの互換性には kmp_set_stacksize_s() を使用します。

メモリの割り当て

関数	説明
kmp_malloc(*size*)	スレッド・ローカル・ヒープから *size* バイトのメモリブロックを割り当てます。
kmp_calloc(*nelem, elsize*)	スレッド・ローカル・ヒープからサイズ *elsize* の *nelem* 要素の配列を割り当てます。
kmp_realloc(*ptr, size*)	スレッド・ローカル・ヒープからアドレス *ptr* および *size* バイトにメモリブロックを再割り当てします。
kmp_free(*ptr*)	スレッド・ローカル・ヒープからアドレス *ptr* のメモリブロックを解放します。メモリは、以前に kmp_malloc()、kmp_calloc()、または kmp_realloc() に割り当てられている必要があります。

ワークキューイング・構造

taskq プラグマ

taskq プラグマは、囲まれた作業 (タスク) 単位が実行される環境を指定します。最初に、taskq プラグマを実行するすべてのスレッド間の中から、1 つのスレッドが選択されます。概念的には、taskq プラグマは、選択したスレッドを実行する空のキューを生成し、次に、taskq ブロックの内側がコード・シングルスレッドで実行されます。他のすべてのスレッドは、この概念キューに作業がキューイングされるのを待ちます。task プラグマは、潜在的に異なるスレッドで実行される作業単位を指定します。taskq ブロック内に task プラグマが存在すると、task ブロックの内側のコードは、taskq に関連付けられている概念キューにキューイングされます。キューイングされたすべての作業が終了し、taskq ブロックの最後に達すると、概念キューはなくなります。

制御構造体

多くの制御構造体は、異なる作業反復と作業生成のパターンを表しているため、ワークキューイング・モデルで並列化可能です。一般的なケース

- `while` ループ
- C++ 反復子
- 再帰関数

while ループ

`while` ループの各反復における計算が独立している場合、ループ全体が `taskq` プラグマに対する環境になります。`while` ループ本体内の文が、`task` プラグマで指定される作業単位になります。`while` ループの条件と制御変数への修正は、`task` ブロックの外側に置かれ、データを制御変数に依存させるようシーケンシャルに実行されます。

C++ 反復子

C++ STL (標準テンプレート・ライブラリ) 反復子は、前述した `while` ループに非常に類似しています。STL にストアされるデータの演算は、すべてのデータに対する反復動作から独立しているからです。データが独立している演算の場合、その演算は、作業の反復がシーケンシャルであれば、並列に処理されます。このタイプの `while` ループ並列処理は、ループの標準 OpenMP* の for ループワークシェアリングを一般化したものです。for ループのワークシェアリングでは、ループの増分演算が反復子で、ループの本体が作業単位です。しかし、`for` ループ反復子の変数は、大抵クローズされたフォームを持つため、その変数は並列に処理することができ、シーケンシャルなステップを回避できます。

再帰関数

再帰関数もまた、並列に反復処理を指定するために使用できます。そのメカニズムは、`sections` プラグマを使用する並列処理を指定するのに類似していますが、より柔軟性があります。`taskq` と `task` プラグマ間に任意のコードを置くことができ、関数の再帰的ネストで、`taskq` キューの概念ツリーをビルドできるためです。`taskq` プラグマの再帰的ネストは、ネストされた OpenMP 並列領域のように動作する OpenMP のワークシェアリング構造の概念を拡張したものです。 ネストされた並列領域のように、ネストされた各ワークキューイング・構造は新しいインスタンスで 1 つのスレッドに検出されます。 しかし主な相違点は、ネストされたワークキューイング・構造は新しいスレッドやチームを生成しないことです。むしろチームからスレッドを再利用します。これにより、動的環境で、マルチ・アルゴリズム並列処理を容易にし、並列処理の各レベルでスレッドの数を確保する必要がなくなり、トップレベルだけで済ますことができます。 その時点から、大量の作業量が内側のレベルで発生すると、外側のレベルから

アイドルスレッドは、その作業の終了を支援することができます。例えば、1 つのスレッドに各インカミング・リクエストを処理させ、多くのスレッドがインカミング・リクエストを待機している状態は、サーバ環境では非常に一般的です。特定のリクエストにおいて、スレッドが処理を開始する時点では、そのサイズが明らかではないことがあります。スレッドが、ネストされたワークキューイング・構造を使用し、内側の構造が開始された後にリクエストのスコープが大きくなると、外側の構造のスレッドは、リクエストを終了するために内側の構造に簡単に移動できます。

ワークキューイング・モデルはシーケンシャルなセマンティクスを保持するように設計されているため、同期化は taskq ブロックのセマンティクスでは固有のものです。taskq 構造を検出したスレッドの taskq ブロックの完了時に暗黙的なチームバリアがあるため、taskq ブロック内で指定されたすべてのタスクの実行が終了したことを確認できます。この taskq バリアは、オリジナル・プログラムのシーケンシャル・セマンティクスを実行します。OpenMP ワークシェアリング構造のように、依存性が存在しないこと、またはタスクブロック間、あるいはタスクブロックのコードと、タスクブロック外の taskq ブロックのコード間で依存性が適切に同期されることを確認する必要があります。

文法、セマンティクスおよび構文は、OpenMP* ワークシェアリング構造と同じように設計されています。OpenMP ワークシェアリング構造で有効な構文のほとんどは、ワークキューイング・プラグマでも、適切な意味を持ちます。

taskq 構造

```
#pragma intel omp taskq [clause[[,]clause]...]
     structured-block
```

clause (構文) は次のいずれかです:

- private (variable-list)
- firstprivate (variable-list)
- lastprivate (variable-list)
- reduction (operator : variable-list)
- ordered
- nowait

private

private 構文は、taskq の variable-list にある各オブジェクトの default-constructed バージョンの private を生成します。囲まれた各タスクで、captureprivate も含みます。各変数に参照されるオリジナルのオブジェクトは、構造に入る時点で中間値を持ちます。構造の動的範囲内では、オブジェクトは変更してはいけません。そして、構造から出る時点で中間値を持ちます。

firstprivate

`firstprivate` 構文は、`taskq` の `variable-list` に各オブジェクトにある copy-constructed バージョンの private を生成します。囲まれた各タスクで、`captureprivate` も含みます。各変数に参照されるオリジナルのオブジェクトは、構造の動的範囲内では、変更してはいけません。そして、構造から出る時点で中間値を持ちます。

lastprivate

`lastprivate` 構文は、`taskq` の `variable-list` にある各オブジェクトの default-constructed バージョンの private を生成します。囲まれた各タスクで、`captureprivate` も含みます。各変数に参照されるオリジナルのオブジェクトは、構造に入る時点で中間値を持ちます。構造の動的範囲内では、オブジェクトは変更してはいけません。そして、タスクの実行が終了した後に、オブジェクトは囲まれた最後のタスクからのオブジェクトの値をコピー割り当てされます。

reduction

`reduction` 構文は、`variable-list` にある各オブジェクトの囲まれたタスク・構造で与えられた演算子を持つリダクション演算を実行します。`operator` と `variable-list` は、OpenMP 仕様と同じように定義されます。

ordered

`ordered` 構文は、オリジナルのシーケンシャル実行順序で、囲まれた `task` 構造で、ordered 構造を実行します。`ordered` と結合される `taskq` ディレクティブには `ordered` 構文がなければなりません。

nowait

`nowait` 構文は、`taskq` の最後にある暗黙的なバリアを削除します。`taskq` 構造にキューされたすべての `task` 構造が処理される前に、スレッドは taskq 構造を終了できます。

task 構造

```
#pragma intel omp task [clause[[,]clause]...]
structured-block
```

`clause` (構文) は次のいずれかです:

- `private( variable-list )`
- `captureprivate( variable-list )`

private

private 構文は、task の variable-list にある各オブジェクトの default-constructed バージョンの private を生成します。変数に参照されるオリジナルのオブジェクトは、構造に入る時点で中間値を持ちます。構造の動的範囲内では、オブジェクトは変更してはいけません。そして、構造から出る時点で中間値を持ちます。

captureprivate

captureprivate 構文は、task がエンキューされた時点で、task の variable-list に各オブジェクトにある copy-constructed バージョンの private を生成します。各変数に参照されるオリジナルのオブジェクトは、その値を保持しますが、task 構造の動的範囲内では、変更してはいけません。

parallel と taskq 構造の組み合わせ

```
#pragma intel omp parallel taskq [clause[[,]clause]...]
structured-block
```

clause (構文) は次のいずれかです:

- if(scalar-expression)
- num_threads(integer-expression)
- copyin(variable-list)
- default(shared | none)
- shared(variable-list)
- private(variable-list)
- firstprivate(variable-list)
- lastprivate(variable-list)
- reduction(operator : variable-list)
- ordered

clause (構文) は、上記の OpenMP の parallel 構造または taskq 構造と同様です。

関数例

下記の test1 関数は、ワークキューイング・モデルを使用する並列化処理を示します。parallel taskq プラグマを持つループと task プラグマを持つループ本体の作業を付けることによって並列処理を記述できます。parallel taskq プラグマは、囲まれた task プラグマに指定された作業単位をエンキューする while ループの環境を指定します。従って、ループの制御構造とエンキューは、シングルスレッドで実行され、チーム内の他のスレッドは taskq キューからの作業をデキューし、実行します。captureprivate 構文は、各タスクがエンキューされる時点、つまり、シーケンシャ

ル・セマンティクスが保存される時点で、リンクポインタ p の private コピーが確実に取り込まれるようにします。

```
void test1(LIST p)
{
  #pragma intel omp parallel taskq shared(p)

    while (p != NULL)
    {
      #pragma intel omp task captureprivate(p)

        do work1(p);
      }
      p = p->next;
    }
  }
}
```

最適化サポート機能

概要: 最適化サポート機能

このセクションでは、ソースコードを直接最適化できるインテル® C++ コンパイラの言語拡張機能について説明します。パフォーマンスの強化や分析に活用できるインテルの拡張ディレクティブおよびライブラリ・ルーチンによる最適化の例を示します。

コンパイラ・ディレクティブ

コンパイラ・ディレクティブ

このセクションでは、次の場合に使用する言語拡張ディレクティブを説明します。

- ソフトウェアのパイプライン化
- ループカウントとループ分配
- ループのアンロール
- プリフェッチ
- ベクトル化

Itanium® ベース・アプリケーションのパイプライン化

swp および noswp ディレクティブは、ループにソフトウェアのパイプライン化を行うかどうかを指示します。swp ディレクティブは、データ依存性を使用しません。しかし、プロファイル・カウントまたは lop-sided コントロール・フローに基づいた手法を上書きします。このディレクティブの構文は次のとおりです:

```
#pragma swp

#pragma noswp
```

swp ディレクティブの例

```
#pragma swp
for (i=0; i<m ; i++)
{
   if (a[i]==0)
   {
      b[i]=a[i]+1;
   }
   else
```

```
    {
        b[i]=a[i]*2;
    }
}
```

swp ディレクティブで起動されるソフトウェアのパイプライン化の最適化は、最も内側のループをスケジューリングする命令を適用します。また、ループ内で命令が異なるステージに分割され、増大した命令レベルの並列処理を許可します。これで、長い待ち時間の演算による影響を減らし、結果として、より速いループを実行します。ソフトウェアのパイプライン化に選択されるループは、インライン化されないプロシージャ呼び出しを含まない、常に最も内側のループです。最適化は、完全にアンロールされたループを最も内側のループとしてもはや認識しないため、完全にアンロールするループは、追加のループを最も内側のループにすることができます。最適化のレポートをリクエスト、および表示して、ソフトウェアのパイプライン化が適用されたかどうかを確認できます (「最適化レポートの生成」を参照してください)。

ループカウントとループ分配

loop count (n) ディレクティブ

loop count (n) ディレクティブはループカウントが n になるように指定します。このディレクティブの構文は次のとおりです:

```
#pragma loop count (n)
```

n は整数定数です。loop count の値はソフトウェアのパイプライン化、ベクトル化およびループ変換に使用される手法に影響を与えます。

loop count (n) ディレクティブの例

```
#pragma loop count (10000)

for(i=0; i<m; i++)
{
    //swp likely to occur in this loop
    a[i]=b[i]+1.2;
}
```

distribute point ディレクティブ

distribute point ディレクティブはループ分配の実行優先度をコンパイラに指示します。このディレクティブの構文は次のとおりです:

```
#pragma distribute point
```

ループ分配は、大きなループを小さなループに分配することがあります。これは、より多くのループにおいてソフトウェアのパイプライン化を有効にします。ディレクティブをループの内側に置く場合、分配はディレクティブの後で行われ、あらゆるループキャリーの依存性が無視されます。ディレクティブをループの前に置く場合、コンパイラは分配する場所を決定し、データ依存性を監視します。1 つの distribute ディレクティブのみが、ループの内側に置かれる際にサポートされます。

distribute point ディレクティブの例

```
#pragma distribute point

for(i=1; i<m; i++)
{
   b[i]=a[i]+1;

   ...

   //Compiler will automatically
   //decide where to distribute.
   //Data dependency is observed.

   c[i]=a[i]+b[i];

   ...

   d[i]=c[i]+1;
}

for(i=1; i<m; i++)
{
   b[i]=a[i]+1;

   ...

   #pragma distribute point

   //Distribution will start here,
   //ignoring all loop-carried dependency.

   sub(a,n);
   c[i]=a[i]+b[i];

   ...

   d[i]=c[i]+1;
}
```

ループのアンロールのサポート

unroll ディレクティブ

`unroll` ディレクティブ (`unroll(n)|nounroll`) は、コンパイラに、カウントされたループをアンロールする回数を伝えます。このディレクティブの構文は次のとおりです:

```
#pragma unroll

#pragma unroll(n)

#pragma nounroll
```

n は 0 から 255 までの整数定数です。unroll ディレクティブは、各 for ループが動作する for 文の前になければなりません。n が指定されると、最適化機構はループを n 回アンロールします。n が省略されるか、n が有効範囲外の場合、最適化機構はループをアンロールする回数を割り当てます。unroll ディレクティブは、コマンドラインから行われるループ・アンロールの設定を変更します。ディレクティブは、最も内側のネストされたループにのみ適用されます。外側のループに適用された場合、無視されます。コンパイラは、n とループカウンタを比較することによって、正しいコードを生成します。

unroll ディレクティブの例

```
#pragma unroll(4)

for(i=1; i<m; i++)
{
   b[i]=a[i]+1;
   d[i]=c[i]+1;
}
```

プリフェッチのサポート

prefetch ディレクティブ

prefetch と noprefetch ディレクティブは、データ・プリフェッチがメモリ参照に生成されるか、されないかを指定します。これは、コンパイラが使用する手法に影響を与えます。このディレクティブの構文は次のとおりです:

```
#pragma noprefetch

#pragma prefetch

#pragma prefetch a,b
```

ループの前に prefetch a を置いて、ループ内で式 a[j] を使用する場合、コンパイラはループ内の a[j+d] のプリフェッチを挿入します。d はコンパイラによって決定されます。このディレクティブは -O3 オプションがオンの場合にサポートされます。

prefetch ディレクティブの例

```
#pragma noprefetch b
#pragma prefetch a

for(i=0; i<m; i++)
{
   a[i]=b[i]+1;
}
```

ベクトル化のサポート (IA-32)

vector ディレクティブは、プログラム中の後に続くループのベクトル化を制御しますが、コンパイラはネストされたループには適用しません。ネストされたそれぞれのループは、その前に、固有のディレクティブが必要です。vector ディレクティブは、loop 文の前に配置してください。

vector always ディレクティブ

vector always ディレクティブは、ベクトル化するかどうかを決定する際、効率ヒューリスティックを無効にするようコンパイラに命令します。そして 1 以外のストライドまたはほとんどアライメントの合っていないメモリアクセスをベクトル化します。

vector always ディレクティブの例

```
#pragma vector always

for(i=0; i<=N; i++)
{
   a[32*i]=b[99*i];
}
```

ivdep ディレクティブ

ivdep ディレクティブは、ベクトル依存性が存在していると推定されてもそれを無視するようコンパイラに命令します。正しいコードにするため、コンパイラは、想定される依存性を証明された依存性として扱います。これは、ベクトル化を行わないようにします。このディレクティブは、その決定を無視します。推定されたループの依存性が安全で、無視できる場合にのみ ivdep を使用してください。下記の例のループは、k の値が不明なため、ivdep ではベクトル化を行いません (ベクトル化は k<0 の場合、不正です)。

ivdep ディレクティブの例

```
#pragma ivdep
```

```
for(i=0; i<m; i++)
{
   a[i]=a[i+k]*c;
}
```

vector aligned ディレクティブ

vector aligned ディレクティブは、ループのベクトル化が可能である限り、ベクトル化によるメリットの存否についての通常のヒューリスティックな判断を無視してベクトル化を行います。aligned または unaligned 指示子が使用されると、ループは、aligned または unaligned 演算を使用して、ベクトル化されます。aligned または unaligned のいずれか 1 つを指定します。

警告

引数として aligned を指定する場合、ループはこの命令を使用してベクトル化可能であることが確実でなければなりません。それ以外の場合、コンパイラは誤ったコードを生成します。次の例に示すループは、コンパイラは通常それが安全であると証明できないように配列が宣言されているため、aligned 指示子を使用して、aligned 命令でループがベクトル化されるように要求を出します。

vector aligned ディレクティブの例

```
#void foo(float *a)
{
   #pragma vector aligned
   for(i=0; i<m; i++)
   {
      a[i]=a[i]*c;
   }
}
```

コンパイラは、コンパイル時にデータ構造のアライメントがわからない場合に備えて、いくつかのアライメント手法を持っています。以下に簡単な例を次に示します (但し、他の方法もサポートされています)。次の例のループにおいて、a のアライメントが不明な場合、コンパイラはプレリュードのループを生成し、ほとんどの場合に発生する配列参照がアライメントされたアドレスにヒットするまでループを繰返します。これにより、a のアライメント・プロパティが判明し、そのプロパティに応じてベクトルループが最適化されます。

アライメント手法の例

```
float *a;

//Alignment unknown
for(i=0; i<100; i++)
```

```
{
   a[i]=a[i]+1.0f;
}

//Dynamic loop peeling
p=a & 0x0f;

{
   p=(16-p)/4;
   for(i=0; i<p; i++)
   {
      a[i]=a[i]+1.0f;
   }
}

//Loop with a aligned.
//Will be vectorized accordingly.
for(i=p; i<100; i++)
{
   a[i]=a[i]+1.0f;
}
```

novector ディレクティブ

novector ディレクティブは、ループのベクトル化が有効であっても、ベクトル化を行わないよう指定します。この例では、反復回数 (ub - lb) が低すぎて、ベクトル化が無駄になるとわかっています。novector を使用して、ループがベクトル化可能であると認識されても、コンパイラにベクトル化しないように指示できます。

novector ディレクティブの例

```
void foo(int lb, int ub)
{
   #pragma novector
   for(j=lb; j<ub; j++)
   {
      a[j]=a[j]+b[j];
   }
```

最適化機構レポートの作成

インテル® C++コンパイラは、最適化レポートを生成、管理するオプションを提供します。

- -opt_report は、最適化レポートを生成して stderr に送ります。デフォルトでは、コンパイラは最適化レポートを生成しません。
- -opt_report_file*filename* は、最適化レポートを作成し、*filename* で指定されたファイルに送ります。

- -opt_report_level{*min*|*med*|*max*} は、最適化レポートの詳細レベルを指定します。*min* 引数は概要レポートを、*max* 引数は完全なレポートを作成します。デフォルトは、-opt_report_level*min* です。
- -opt_report_routine*fileroutine_substring* は、名前の一部に *substring* を含むすべてのルーチンからレポートを作成します。指定されない場合、すべてのルーチンからのレポートが生成されます。デフォルトでは、コンパイラはすべてのルーチンのレポートを生成します。

レポートを作成する最適化の指定

コンパイラは、-opt_report_phase*phase* オプションの *phase* 引数で指定される最適化機構のレポートを作成できます。複数の最適化機構のレポートを作成するために同じコマンドライン上に複数回、オプションを使用できます。現在、次の最適化機構レポートがサポートされています。

最適化機構の論理名	最適化機構のフルネーム
ipo	Interprocedural Optimizer (プロシージャ間の最適化)
hlo	High Level Optimizer (高レベル最適化)
ilo	Intermediate Language Scalar Optimizer (中間言語スカラ最適化機構)
ecg	Code Generator (コード・ジェネレータ)
omp	OpenMP
all	すべてのフェーズ

上記の最適化機構の論理名の 1 つが指定されるとその最適化機構からのすべてのレポートが作成されます。

例えば、-opt_report_phase*ipo* -opt_report_phase*ecg* では、プロシージャ間の最適化およびコードの生成からレポートが作成されます。

各最適化機構は、特定の最適化を行うことができます。これらの各最適化は、論理名の 1 つがプリフィックスになります。
例:

Optimizer_optimization	フルネーム
ipo_inline	Interprocedural Optimizer (プロシージャ間の最適化)、inline expansion of functions (関数のインライン展開)
ipo_constant_propagation	Interprocedural Optimizer (プロシージャ間の最適化)、

	constant propagation (定数伝播)
ipo_function_reorder	Interprocedural Optimizer (プロシージャ間の最適化)、function reorder (関数の再順序化)
ilo_constant_propagation	Intermediate Language Scalar Optimizer (中間言語スカラ最適化機構)、constant propagation (定数の伝播)
ilo_copy_propagation	Intermediate Language Scalar Optimizer (中間言語スカラ最適化機構)、copy propagation (コピー伝播)
ecg_software_pipelining	Code Generator (コード・ジェネレータ)、software pipelining (ソフトウェアのパイプライン化)

指定された最適化機構のプリフィックスと一致するすべての最適化レポートが作成されます。例えば、-opt_report_phase ilo_co が指定された場合、定数伝播およびコピー伝播の両方からのレポートが作成されます。

利用可能なレポート生成

-opt_report_help オプションは、レポートの作成に利用可能な最適化機構の論理名をリストします。

アプリケーションの時間測定

アプリケーションの実行速度は、パフォーマンスを測る 1 つの目安になります。アプリケーションの速度を測るときは、次の環境を考慮してください:

- 他のユーザが誰もアクティブではないときに、プログラムの時間測定を行います。時間測定中、1 つまたは複数の CPU 集中型プロセスが起動していると、結果に影響します。
- 最も正確な結果を得るには、毎回、同じ条件でプログラムを実行するようにしてください。特に、同一プログラムの以前のバージョンと比較する際は注意してください。可能ならば、同じシステム (プロセッサ・モデル、メモリ、オペレーティング・システムのバージョンなど) で実行してください。
- システムを変更する必要がある場合、両方のシステムでプログラムの同一バージョンの速度を測ります。こうすることによって、システムによる速度の違いを把握できます。
- 数秒以下で実行されるプログラムの場合は、数回測定し、正しい結果であることを確認してください。外部プログラムをロードするような一部のオーバヘッドのあるプログラムは、短い時間に大きな影響を与えます。

- プログラムが多くのテキストを表示する場合、プログラムの出力をリダイレクトすることを考慮してください。プログラムの出力をリダイレクトすると、画面 I/O が減少するため、レポートされる時間が変わります。

次に、プログラムの時間測定のモデルを示します:

```
/* Sample Timing */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
   clock t start, finish;
   long loop;
   double  duration, loop calc;
   start = clock();
   for(loop=0; loop <= 2000; loop++)
   {
      loop calc = 123.456 * 789;

      //printf() inculded to facilitate example
      printf("\nThe value of loop is: %d", loop);

   finish = clock();
   duration = (double)(finish - start)/CLOCKS PER SEC;
   printf("\n%2.3f seconds\n", duration);
}
```

コンパイラの制限

コンパイラの制限

次の表に、コンパイラが処理できる各アイテムのサイズまたは数の限界を示します。表に示したすべての容量は、テスト済みの値です。実際の値は、表の値より大きくなる可能性があります。

アイテム	テスト済みの値
制御構造のネスト (ブロックネスト)	512
条件付きコンパイルのネスト	512
宣言子の修飾子	512
カッコのネストレベル	512
有効文字数 (内部識別子)	2048
外部識別子名の長さ	64K
外部識別子/ファイルの数	128K
1 つのブロック内の識別子の数	2048
同時に定義されるマクロの数	128K
関数呼び出しのパラメータの数	512
マクロ 1 つ当たりのパラメータの数	512
文字列内の文字数	128K
オブジェクト内のバイト数	512K
インクルード・ファイルのネストの深さ	512
スイッチ内のケースラベル	32K
1 つの構造体または共用体内のメンバ数	32K
1 つの列挙子内の列挙定数	8192
構造体のネストレベル	320
配列のサイズ	2 GB

主要なファイル

IA-32 コンパイラの主要なファイルの概要

次の表に、IA-32 版コンパイラ用にインストールされるファイルの一覧と簡単な説明を示します。

/bin ファイル

ファイル	説明
codecov	コード・カバレッジ・ツール
iccvars.sh	環境変数を設定するバッチファイル
icc.cfg	コマンドラインから使用する設定ファイル
icc icpc	インテル® C++ コンパイラ
profmerge	プロファイルに基づく最適化に使用するユーティリティ
proforder	プロファイルに基づく最適化に使用するユーティリティ
tselect	テスト・プライオリタイゼーション・ツール
xiar	プロシージャ間の最適化に使用するツール
xild	プロシージャ間の最適化に使用するツール

/include ファイル

ファイル	説明
dvec.h	クラス・ライブラリ用の SSE 2 組込み関数
emm_func.h	SSE2 組込み関数用のヘッダファイル (emmintrin.h で使用)
emmintrin.h	SSE2 組込み関数用の主ヘッダファイル
float.h	標準 float.h の IEEE 754 バージョン
fvec.h	クラス・ライブラリ用の SSE 組込み関数
iso646.h	標準ヘッダファイル
ivec.h	クラス・ライブラリ用の MMX® 命令の組込み関数
limits.h	標準ヘッダファイル

mathf.h	従来のインテル® 数値演算ライブラリ用の主ヘッダファイル
mathimf.h	現在のインテル数値演算ライブラリ用の主ヘッダファイル
mmintrin.h	MMX 命令用の組込み関数
omp.h	OpenMP* 用の主ヘッダファイル
omp_lib.h	OpenMP 用のヘッダファイル
pgouser.h	プロファイルに基づく最適化でインストルメンテーション・コンパイルに使用するファイル
pmmintrin.h	HT テクノロジ インテル® Pentium® 4 プロセッサ (SSE3 対応) 用の主ヘッダファイル
proto.h	
sse2mmx.h	ストリーミング SIMD 拡張命令 2 の組込み関数用の主ヘッダファイル
stdarg.h	標準 stdarg.h の置換ヘッダ
stdbool.h	_Bool キーワードの定義
stddef.h	標準ヘッダファイル
syslimits.h	
varargs.h	標準 varargs.h の置換ヘッダ
xarg.h	stdargs.h と varargs.h で使用されるヘッダファイル
xmm_func.h.h	ストリーミング SIMD 拡張命令用のヘッダファイル
xmm_utils.h	ストリーミング SIMD 拡張命令用のファイル
xmmintrin.h	ストリーミング SIMD 拡張命令の組込み関数用の主ヘッダファイル

/lib ファイル

ライブラリ	説明
libguide.a libguide.so	OpenMP* 用
libguide_stats.a libguide_stats.so	パフォーマンス統計とプロファイル情報を含むパラレライザ・ツール用の OpenMP スタティック・ライブラリ
libompstub.a	OpenMP の未使用時に OpenMP サブルーチンの参照を解決するライブラリ
libsvml.a	SVML (Short Vector Mathematical Library)
libirc.a	PGO および CPU ディスパッチ用にインテルが提供するライブラリ
libircmt.a	libirc.a のマルチスレッド・バージョン

libimf.a	インテル数値演算ライブラリ
libimf.so	インテル数値演算ライブラリ
libcprts.a libcprts.so libcprts.so.3	Dinkumware* C++ ライブラリ
libunwind.a libunwind.so libunwind.so.3	Unwinder ライブラリ
libcxa.a libcxa.so libcxa.so.3	C++ 機能のインテル・ランタイム・サポート
libcxaguard.a libcxaguard.so libcxaguard.so.3	-cxxlib-gcc オプションとともに相互運用性のサポートに使用されます。 「gcc との相互運用性」を参照してください。

Itanium® コンパイラの主要なファイルの概要

次の表に、Itanium® コンパイラ用にインストールされるファイルの一覧と簡単な説明を示します。

/bin ファイル

ファイル	説明
codecov	コード・カバレッジ・ツール
iccvars.sh	環境変数を設定するバッチファイル
icc.cfg	コマンドラインから使用する設定ファイル
icc icpc	インテル® C++ コンパイラ
profmerge	プロファイルに基づく最適化に使用するユーティリティ
proforder	プロファイルに基づく最適化に使用するユーティリティ
tselect	テスト・プライオリタイゼーション・ツール
xiar	プロシージャ間の最適化に使用するツール
xild	プロシージャ間の最適化に使用するツール

/include ファイル

ファイル	説明
emmintrin.h	SSE2 組込み関数用の主ヘッダファイル
float.h	標準 float.h の IEEE 754 バージョン
fvec.h	クラス・ライブラリ用の SSE 組込み関数
ia64intrin.h	
ia64regs.h	標準ヘッダファイル
iso646.h	標準ヘッダファイル
ivec.h	クラス・ライブラリ用の MMX® 命令の組込み関数
limits.h	標準ヘッダファイル
mathimf.h	現在のインテル® 数値演算ライブラリ用の主ヘッダファイル
mmintrin.h	MMX 命令用の組込み関数
omp.h	OpenMP* 用の主ヘッダファイル
pgouser.h	プロファイルに基づく最適化でインストルメンテーション・コンパイルに使用するファイル
proto.h	
sse2mmx.h	ストリーミング SIMD 拡張命令 2 の組込み関数用の主ヘッダファイル
stdarg.h	標準 stdarg.h の置換ヘッダ
stdbool.h	_Bool キーワードの定義
stddef.h	標準ヘッダファイル
syslimits.h	
varargs.h	標準 varargs.h の置換ヘッダ
xarg.h	stdargs.h と varargs.h で使用されるヘッダファイル
xmmintrin.h	ストリーミング SIMD 拡張命令の組込み関数用の主ヘッダファイル

/lib ファイル

ファイル	説明
libcprts.a	C++標準言語ライブラリ
libcxa.so	I/O データの位置を示す C++ 言語ライブラリ
libirc.a	インテル固有のライブラリ (最適化)

libm.a	数値演算ライブラリ
libguide.a	OpenMP ライブラリ
libguide.so	共用 OpenMP ライブラリ
libmofl.a	インテル・アセンブラが使用する、複数オブジェクト・フォーマット・ライブラリ
libmofl.so	インテル・アセンブラが使用する、共用複数オブジェクト・フォーマット・ライブラリ
libunwinder.a	Unwinder ライブラリ
libintrins.a	組込み関数ライブラリ

診断およびメッセージ

概要: 診断およびメッセージ

本書では、コンパイラから出力する各種メッセージについて説明します。メッセージの種類としては、開始メッセージ (サインオン・メッセージ) および、リマーク、警告、エラーのそれぞれに関係した診断メッセージがあります。診断メッセージの場合は必ず、エラーを含むソース行も一緒に標準出力に出力します。

本書では、診断メッセージの重要度の制御方法についても説明します。

診断メッセージ

オプション	説明
-w0	エラーを表示します (-w と同じ)
-w1	警告とエラーを表示します (デフォルト)
-w2	リマーク、警告、エラーを表示します

言語診断

ソースファイルの処理中に報告された診断結果を示すメッセージです。この診断結果は次の形式で出力します。

```
filename (linenum): type [#nn]: message
```

filename	現在処理しているソースファイルの名前です。
linenum	問題が検出されたソース行の番号です。
type	診断メッセージの重要度です。warning (警告)、remark (リマーク)、error (エラー)、catastrophic error (致命的エラー) のいずれかです。
[#nn]	エラー (または警告) メッセージの番号です。ハードエラーまたは致命的エラーに番号は付きません。
message	診断結果の説明文です。

警告メッセージの例を次に示します:

```
tantst.cpp(3): warning #328: Local variable "increment"
never used.
```

標準的なエラーに関する内部エラー・メッセージを表示する場合があります。コンパイル時に内部エラーが表示されたときは、インテルのカスタマサポートまでご連絡ください。内部エラー・メッセージは次の形式で出力されます:

```
FATAL COMPILER ERROR: message
```

lint のコメントを使用した警告メッセージの出力停止

UNIX の `lint` プログラムは、C/C++ プログラムの中にバグ、移植不能部分、無駄の多い部分がないかどうかを検出します。このコンパイラでは、以下に示した、`lint` 固有のコメントが 3 つ認識されます:

1. `/*ARGSUSED*/`
2. `/*NOTREACHED*/`
3. `/*VARARGS*/`

`lint` プログラムと同じように、このコンパイラでも、ソースの特定の位置にこれらのコメントが書き込んであれば、何らかの条件に関する警告を表示しないようにできます。

警告メッセージの出力停止方法とリマーク・メッセージの出力方法

警告メッセージを非表示にするときやリマーク・メッセージ機能を有効にするには、前処理やコンパイルの各フェーズで `-w` オプションか `-Wn` オプションを使用します。このオプションは、次のいずれかの引数とともに入力できます:

オプション	説明
`-w0`	エラーを表示します (`-w` と同じ)
`-w1`	警告とエラーを表示します (デフォルト)
`-w2`	リマーク、警告、エラーを表示します

不正な箇所がコードに含まれていても、それについて既に把握しているときや実際には問題のないとき (K&R の C 構造など) にまで警告メッセージを表示させたくない場

合があります。例えば、次のコマンドを実行すると、newprog.cpp がコンパイルされ、コンパイラ・エラーは表示されますが、警告メッセージは出力されません:

```
prompt>icpc -W0 newprog.cpp
```

特定の診断結果を示すには、-ww、-we、または -wd オプションを使用します。

オプション	説明
-wwL1[L2,...,Ln]	L1 から LN までの診断結果の重要度を警告に変更します。
-weL1[L2,...,Ln]	L1 から LN までの診断結果の重要度をエラーに変更します。
-wdL1[L2,...,Ln]	L1 から LN までの診断を禁止します。

例

```
/* test.c */

int main()
{
   int x=0;
}
```

-Wall オプション (すべての警告を有効にする) を使用して上記の test.c をコンパイルした場合、コンパイラは警告 #177 を出力します:

```
prompt>icc -Wall test.c

remark #177: variable 'x' was declared but never referenced
```

警告 #177 を出力しないようにするには、-wd オプションを使用します:

```
prompt>icc -Wall -wd177 test.c
```

同様に、-we オプションを使用するとコンパイル時にエラーが出力されます:

```
prompt>icc -Wall -we177 test.c

error #177: variable 'x' was declared but never referenced

compilation aborted for test.c
```

エラーの出力個数の制限

エラー・メッセージがいくつ表示したらコンパイラを終了するのかは、-wn*n* オプションで指定します。デフォルトでは、エラー数が 100 を超えるとコンパイルの処理が止まります。

オプション	説明
-wn*n*/*i*	*n* で指定した個数を超すエラー・メッセージが出力されるとコンパイルの処理が止まります。リマーク・メッセージと警告メッセージはこの個数には入りません。

例えば、次のコマンドラインを実行した場合は、ファイル a.cpp のコンパイル中にエラー・メッセージの出力個数が 50 個を超えたときにコンパイルの処理が止まります。

```
prompt>icpc -wn50 -c a.cpp
```

リマーク・メッセージ

C/C++ で一般的によく使用されるが実際には規約に違反している恐れのある用法を報告するメッセージです。「警告メッセージの出力停止方法とリマーク・メッセージの出力方法」で説明したように、-W オプションをレベル 4 に指定しない限り、リマーク・メッセージは出力も表示もされません。リマーク・メッセージが出ても、変換もリンクも停止しません。リマーク・メッセージを表示しても出力ファイルは影響を受けません。次に、リマーク・メッセージの代表例をいくつか示します:

- function declared implicitly
- type qualifiers are meaningless in this declaration
- controlling expression is constant

インテル® 数値演算ライブラリ

概要: インテル® 数値演算ライブラリ

インテル® C++ コンパイラには、高度に最適化された正確な数学関数を含む数値演算ソフトウェア・ライブラリが含まれています。これらの関数は、科学やグラフィック・アプリケーションに広く使用されます。同様に、浮動小数点演算を多用するプログラムにも使用されます。C99 `_Complex` データ型のサポートは、`-c99` コンパイラ・オプションを使用することにより含まれます。`mathimf.h` ヘッダファイルには、ライブラリ関数のプロトタイプが含まれています。「インテル® 数値演算ライブラリの使用」を参照してください。利用可能な関数の一覧は、このセクションの「関数一覧」を参照してください。

IA-32 と Itanium® ベース・システムの数値演算ライブラリ

アプリケーションにリンクされる数値演算ライブラリは、指定されたコンパイルまたはリンケージ・オプションに依存します。下の表を参照してください:

ライブラリ	説明
`libimf.a`	デフォルトの静的数値演算ライブラリです。
`libimf.so`	デフォルトの共用数値演算ライブラリです。

「数値演算ライブラリ」も参照してください。

インテル® 数値演算ライブラリの使用

インテル® 数値演算ライブラリを使用するには、プログラムに、ヘッダファイル `mathimf.h` をインクルードしてください。下記に、数値演算ライブラリを使用した 2 つのプログラム例を示します。

実関数の使用例

```
// real math.c

#include <stdio.h>
#include <mathimf.h>

int main() {

float  fp32bits;
```

```
double fp64bits;
long double fp80bits;
long double pi_by_four = 3.141592653589793238/4.0;

// pi/4 radians is about 45 degrees.

fp32bits = (float) pi by four;   // float approximation to pi/4
fp64bits = (double) pi by four;  // double approximation to pi/4
fp80bits = pi by four;           // long double (extended)
approximation to pi/4

// The sin(pi/4) is known to be 1/sqrt(2) or approximately .7071067

printf("When x = %8.8f, sinf(x) = %8.8f \n", fp32bits,
sinf(fp32bits));
printf("When x = %16.16f, sin(x) = %16.16f \n", fp64bits,
sin(fp64bits));
printf("When x = %20.20Lf, sinl(x) = %20.20f \n", fp80bits,
sinl(fp80bits));

return 0;
```

上記のプログラム例では、long double データ型を含むため、-long_double コンパイラ・オプションを必ず含めるようにしてください。

prompt>**icc -long_double real_math.c**

a.out の出力は次のようになります:

```
When x = 0.78539816, sinf(x) = 0.70710678
When x = 0.7853981633974483, sin(x) = 0.7071067811865475
When x = 0.78539816339744827900, sinl(x) = 0.70710678118654750275
```

複素数関数の使用例

```
// complex math.c

#include <stdio.h>
#include <mathimf.h>

int main()
{

float   Complex c32in,c32out;
double  Complex c64in,c64out;
double pi by four= 3.141592653589793238/4.0;

c64in = 1.0 +   I  * pi by four;

// Create the double precision complex number 1 + (pi/4) * i
// where i is the imaginary unit.

c32in = (float  Complex) c64in;
```

```
// Create the float complex value from the double complex value.

c64out = cexp(c64in);
c32out = cexpf(c32in);

// Call the complex exponential,
// cexp(z) = cexp(x+iy) = e^ (x + i y) = e^x * (cos(y) + i sin(y))

printf("When z = %7.7f + %7.7f i, cexpf(z) = %7.7f + %7.7f i \n"
,crealf(c32in),cimagf(c32in),crealf(c32out),cimagf(c32out));
printf("When z = %12.12f + %12.12f i, cexp(z) = %12.12f + %12.12f i
\n"
,creal(c64in),cimag(c64in),creal(c64out),cimagf(c64out));

return 0;
}
```

prompt>**icc complex_math.c**

a.out の出力は次のようになります:

```
When z = 1.0000000 + 0.7853982 i, cexpf(z) = 1.9221154 + 1.9221156 i
When z = 1.000000000000 + 0.785398163397 i, cexp(z) = 1.922115514080
+ 1.922115514080 i
```

注

_Complex データ型は、C プログラムではサポートされていますが、C++ プログラムではサポートされていません。

例外条件

未定義の結果になる引数を使用して数値関数を呼び出した場合、エラー番号がシステム変数 errno に割り当てられます。数値関数エラーは通常、領域エラーまたは範囲エラーです。

領域エラーは、関数で領域外の引数が使用された場合に発生します。例えば、acos の引数は -1 から +1 の間のみと定義されています。このため、acos(-2) や acos(3) を評価すると、領域エラーが発生し、戻り値は QNaN になります。

範囲エラーは、数学上は有効な引数を使用して、関数の値が浮動小数点データ型で表現可能な値の範囲を超えた場合に発生します。例えば、exp(1000) を評価すると、範囲エラーが発生し、戻り値は INF になります。

領域または範囲エラーが発生すると、次の値が errno に割り当てられます:

- 領域エラー (EDOM): errno = 33
- 範囲エラー (ERANGE): errno = 34

次の例は、EDOM および ERANGE エラーの errno の読み方を示したものです。

```
// errno.c

#include <errno.h>
#include <mathimf.h>
#include <stdio.h>

int main(void)
{
   double neg_one=-1.0;
   double zero=0.0;

   // The natural log of a negative number is considered a domain
error - EDOM
   printf("log(%e) = %e and errno(EDOM) = %d
\n",neg_one,log(neg_one),errno);

   // The natural log of zero is considered a range error - ERANGE
   printf("log(%e) = %e and errno(ERANGE) = %d
\n",zero,log(zero),errno);
}
```

errno.c の出力は次のようになります:

```
log(-1.000000e+00) = nan and errno(EDOM) = 33
log(0.000000e+00) = -inf and errno(ERANGE) = 34
```

このセクションの数値関数では、適用可能な場合、errno の対応する値が表示されます。

その他の考慮事項

コンパイラによって、自動的にインライン化される数値演算関数もあります。実際にインライン化される関数は、使用するベクトル化またはプロセッサ固有のコンパイラ・オプションに依存し、異なります。詳細は、「関数のインライン展開の条件」を参照してください。

デフォルトの精度制御または丸めモードを変更すると、いくつかの算術関数で返される結果に影響する場合があります。「浮動小数点演算の精度」を参照してください。

使用するデータ型によるいくつかの重要なコンパイラ・オプションは次のとおりです。

- -long_double: long double データ型 (80 ビット浮動小数点) のサポートが必要なプログラムをコンパイルする場合は、このオプションを使用します。このオプションがなくても、コンパイルは成功しますが、long double データ型は、double データ型にマップされます。
- -c99: _Complex データ型のサポートが必要なプログラムをコンパイルする場合は、このオプションを使用します

数値演算関数

三角関数

インテル® 数値演算ライブラリは、次の三角関数をサポートします:

ACOS

説明: `acos` 関数は、範囲 [-1,1] の `x` の弧度 [0,pi] 範囲内で、`x` の逆余弦の主値を返します。

errno: EDOM、|x| > 1 の場合

呼び出しインターフェイス:

```
double acos(double x);
long double acosl(long double x);
float acosf(float x);
```

ACOSD

説明: `acosd` 関数は、範囲 [-1,1] の `x` の角度 [0,180] の範囲で、`x` の逆余弦の主値を返します。

errno: EDOM、|x| > 1 の場合

呼び出しインターフェイス:

```
double acosd(double x);
long double acosdl(long double x);
float acosdf(float x);
```

ASIN

説明: `asin` 関数は、範囲 [-1,1] の `x` の弧度 [-pi/2, +pi/2] 範囲内で、`x` の逆正弦の主値を返します。

errno: EDOM、|x| > 1 の場合

呼び出しインターフェイス:

```
double asin(double x);
long double asinl(long double x);
float asinf(float x);
```

ASIND

説明: asind 関数は、範囲 [-1,1] の x の角度 [-90,90] の範囲で、x の逆正弦の主値を返します。

errno: EDOM、|x| > 1 の場合

呼び出しインターフェイス:

```
double asind(double x);
long double asindl(long double x);
float asindf(float x);
```

ATAN

説明: atan 関数は、弧度 [-pi/2, +pi/2] の範囲内で、x の逆正接の主値を返します。

呼び出しインターフェイス:

```
double atan(double x);
long double atanl(long double x);
float atanf(float x);
```

ATAN2

説明: atan2 関数は、弧度 [-pi, +pi] の範囲内で、y/x の逆正接の主値を返します。

errno: EDOM、x = 0 および y=0 の場合

呼び出しインターフェイス:

```
double atan2(double x, double y);
long double atan2l(long double x, long double y);
float atan2f(float x, float y);
```

ATAND

説明: atand 関数は、角度 [-90, 90] の範囲で、x の逆正接の主値を返します。

呼び出しインターフェイス:

```
double atand(double x);
long double atandl(long double x);
float atandf(float x);
```

ATAN2D

説明: atand2 関数は、角度 [-180, +180] の範囲内で、y/x の逆正接の主値を返します。

errno: EDOM、x = 0 の場合

呼び出しインターフェイス:

```
double atan2d(double x, double y);
long double atan2dl(long double x, long double
y);
float atan2df(float x, float y);
```

COS

説明: cos 関数は、弧度法で x の余弦を返します。この関数は、Itanium® コンパイラによってインライン化されます。

呼び出しインターフェイス:

```
double cos(double x);
long double cosl(long double x);
float cosf(float x);
```

COSD

説明: cosd 関数は、角度法で x の余弦を返します。

呼び出しインターフェイス:

```
double cosd(double x);
long double cosdl(long double x);
float cosdf(float x);
```

COT

説明: cot 関数は、弧度法で x の余接を返します。

errno: ERANGE、オーバーフロー状態の場合

呼び出しインターフェイス:

```
double cot(double x);
long double cotl(long double x);
float cotf(float x);
```

COTD

説明: cotd 関数は、角度法で x の余接を返します。

errno: ERANGE、オーバーフロー状態の場合

呼び出しインターフェイス:

```
double cotd(double x);
long double cotdl(long double x);
float cotdf(float x);
```

SIN

説明: sin 関数は、弧度法で x の正弦を返します。この関数は、Itanium コンパイラによってインライン化されます。

呼び出しインターフェイス:

```
double sin(double x);
long double sinl(long double x);
float sinf(float x);
```

SINCOS

説明: sincos 関数は、弧度法で x の正弦と余弦の両方を返します。この関数は、Itanium コンパイラによってインライン化されます。

呼び出しインターフェイス:

```
void sincos(double x, double *sinval, double
*cosval);
void sincosl(long double x, long double *sinval,
long double *cosval);
void sincosf(float x, float *sinval, float
*cosval);
```

SINCOSD

説明: sincosd 関数は、角度法で x の正弦と余弦の両方を返します。

呼び出しインターフェイス:

```
void sincosd(double x, double *sinval, double
*cosval);
void sincosdl(long double x, long double *sinval,
long double *cosval);
void sincosdf(float x, float *sinval, float
*cosval);
```

SIND

説明: sind 関数は、角度法で x の正弦を計算します。

呼び出しインターフェイス:

```
double sind(double x);
long double sindl(long double x);
float sindf(float x);
```

TAN

説明: tan 関数は、弧度法で x の正接を返します。

呼び出しインターフェイス:

```
double tan(double x);
long double tanl(long double x);
float tanf(float x);
```

TAND

説明: tand 関数は、角度法で x の正接を返します。

errno: ERANGE、オーバーフロー状態の場合

呼び出しインターフェイス:

```
double tand(double x);
long double tandl(long double x);
float tandf(float x);
```

双曲線関数

インテル® 数値演算ライブラリは、次の双曲線関数をサポートします:

ACOSH

説明: acosh 関数は、x の逆双曲線余弦を返します。

errno: EDOM、x < 1 の場合

呼び出しインターフェイス:

```
double acosh(double x);
long double acoshl(long double x);
float acoshf(float x);
```

ASINH

説明: asinh 関数は、x の逆双曲線正弦を返します。

呼び出しインターフェイス:

```
double asinh(double x);
long double asinhl(long double x);
float asinhf(float x);
```

ATANH

説明: atanh 関数は、x の逆双曲線正接を返します。

errno: EDOM、x < 1 の場合
errno: ERANGE、x = 1 の場合

呼び出しインターフェイス:

```
double atanh(double x);
long double atanhl(long double x);
float atanhf(float x);
```

COSH

説明: cosh 関数は、x の双曲線余弦、$(e^x + e^{-x})/2$ を返します。

errno: ERANGE、オーバーフロー状態の場合

呼び出しインターフェイス:

```
double cosh(double x);
long double coshl(long double x);
float coshf(float x);
```

SINH

説明: sinh 関数は、x の双曲線正弦、$(e^x - e^{-x})/2$ を返します。

errno: ERANGE、オーバーフロー状態の場合

呼び出しインターフェイス:

```
double sinh(double x);
long double sinhl(long double x);
float sinhf(float x);
```

SINHCOSH

説明: sinhcosh 関数は、x の双曲線正弦と双曲線余弦の両方を返します。

errno: ERANGE、オーバーフロー状態の場合

呼び出しインターフェイス:

```
void sinhcosh(double x, float *sinval, float
*cosval);
void sinhcoshl(long double x, long double *sinval,
long double *cosval);
void sinhcoshf(float x, float *sinval, float
*cosval);
```

TANH

説明: tanh 関数は、x の双曲線正接、$(e^x - e^{-x}) / (e^x + e^{-x})$ を返します。

呼び出しインターフェイス:

```
double tanh(double x);
long double tanhl(long double x);
float tanhf(float x);
```

指数関数

インテル® 数値演算ライブラリは、次の指数関数をサポートします:

CBRT

説明: cbrt 関数は、x の立方根を返します。

Calling interface:

```
double cbrt(double x);
long double cbrtl(long double x);
float cbrtf(float x);
```

EXP

説明: exp 関数は、e を x 乗した値、e^x を返します。この関数は、Itanium® コンパイラによってインライン化されます。

errno: ERANGE、アンダーフローおよびオーバーフロー状態の場合

呼び出しインターフェイス:

```
double exp(double x);
long double expl(long double x);
float expf(float x);
```

EXP10

説明: exp10 関数は、10 を x 乗した値、10^x を返します。

errno: ERANGE、アンダーフローおよびオーバーフロー状態の場合

呼び出しインターフェイス:

```
double exp10(double x);
long double exp10l(long double x);
float exp10f(float x);
```

EXP2

説明: exp2 関数は、2 を x 乗した値、2^x を返します。

errno: ERANGE、アンダーフローおよびオーバーフロー状態の場合

呼び出しインターフェイス:

```
double exp2(double x);
long double exp2l(long double x);
float exp2f(float x);
```

EXPM1

説明: expm1 関数は、e を x 乗した値から 1 をマイナスした値、$e^x - 1$ を返します。

errno: ERANGE、オーバーフロー状態の場合

呼び出しインターフェイス:

```
double expm1(double x);
long double expm1l(long double x);
float expm1f(float x);
```

FREXP

説明: frexp 関数は、x の浮動小数点数を、2 のべき乗を乗じた [1/2, 1) 範囲で符号付き正規化分数に変換します。符号付き正規化分数が返され、その整数の指数部分を exp に格納します。

呼び出しインターフェイス:

```
double frexp(double x, int *exp);
long double frexp(long double x, int *exp);
float frexpf(float x, int *exp);
```

HYPOT

説明: hypot 関数は、$(x^2 + y^2)$ の平方根を返します。

errno: ERANGE、オーバーフロー状態の場合

呼び出しインターフェイス:

```
double hypot(double x, double y);
long double hypotl(long double x, long double y);
float hypotf(float x, float y);
```

ILOGB

説明: ilogb 関数は、x のべき指数を符号付 int 値として返します。

errno: ERANGE、x = 0 の場合

呼び出しインターフェイス:

```
int ilogb(double x);
int ilogbl(long double x);
int ilogbf(float x);
```

LDEXP

説明: ldexp 関数は $x*2^{exp}$ を返します。exp は整数値です。

errno: ERANGE、アンダーフローおよびオーバーフロー状態の場合

呼び出しインターフェイス:

```
double ldexp(double x, int exp);
long double ldexpl(long double x, int exp);
float ldexpf(float x, int exp);
```

LOG

説明: log 関数は、x の自然対数、ln(x) を返します。この関数は、Itanium コンパイラによってインライン化されます。

errno: EDOM、x < 0 の場合
errno: ERANGE、x = 0 の場合

呼び出しインターフェイス:

```
double log(double x);
long double logl(long double x);
float logf(float x);
```

LOG10

説明: log10 関数は、基数 10 の x の対数、$\log_{10}(x)$ を返します。この関数は、Itanium コンパイラによってインライン化されます。

errno: EDOM、x < 0 の場合
errno: ERANGE、x = 0 の場合

呼び出しインターフェイス:

```
double log10(double x);
long double log10l(long double x);
float log10f(float x);
```

LOG1P

説明: log1p 関数は、(x+1) の自然対数、ln(x + 1) を返します。

errno: EDOM、x < -1 の場合
errno: ERANGE、x = -1 の場合

呼び出しインターフェイス:

```
double log1p(double x);
long double log1pl(long double x);
float log1pf(float x);
```

LOG2

説明: log2 関数は、基数 2 の x の対数、$\log_2(x)$ を返します。

errno: EDOM、x < 0 の場合
errno: ERANGE、x = 0 の場合

呼び出しインターフェイス:

```
double log2(double x);
long double log2l(long double x);
float log2f(float x;
```

LOGB

説明: logb 関数は、x の符号付き指数を返します。

errno: EDOM、x = 0 の場合

呼び出しインターフェイス:

```
double logb(double x);
long double logbl(long double x);
float logbf(float x);
```

POW

説明: pow 関数は、x を y 乗した値、x^y を返します。

呼び出しインターフェイス:

errno: EDOM、x = 0 および y < 0 の場合
errno: EDOM、x < 0 および y が整数ではない場合
errno: ERANGE、オーバーフロー状態の場合

```
double pow(double x, double y);
long double powl(double x, double y);
float powf(float x, float y);
```

SCALB

説明: scalb 関数は、$x*2^y$ を返します。y は浮動小数点値です。

errno: ERANGE、アンダーフローおよびオーバーフロー状態の場合

呼び出しインターフェイス:

```
double scalb(double x, double y);
long double scalbl(long double x, long double y);
float scalbf(float x, float y);
```

SCALBN

説明: scalbn 関数は、$x*2^n$ を返します。n は整数値です。

errno: ERANGE、アンダーフローおよびオーバーフロー状態の場合

呼び出しインターフェイス:

```
double scalbn(double x, int n);
long double scalbnl (long double x, int n);
float scalbnf(float x, int n);
```

SCALBLN

説明: scalbln 関数は、$x*2^n$ を返します。n は long 整数値です。

errno: ERANGE、アンダーフローおよびオーバーフロー状態の場合

呼び出しインターフェイス:

```
double scalbln(double x, long int n);
long double scalblnl (long double x, long int n);
float scalblnf(float x, long int n);
```

SQRT

説明: sqrt 関数は、正確な丸め平方根を返します。

errno: EDOM、x < 0 の場合

呼び出しインターフェイス:

```
double sqrt(double x);
long double sqrtl(long double x);
float sqrtf(float x);
```

特殊関数

インテル® 数値演算ライブラリは、次の特殊関数をサポートしています:

ANNUITY

説明: annuity 関数は、一定期間による一定額の支払い (年金) の原価係数を計算します。$(1 - (1+x)^{(-y)}) / x$ の x は利率、y は期間です。

errno: ERANGE、アンダーフローおよびオーバーフロー状態の場合

呼び出しインターフェイス:

```
double annuity(double x, double y);
long double annuity(double x, double y);
float annuityf(float x, double y);
```

COMPOUND

説明: compound 関数は、複利係数を計算します。$(1+x)^y$ で x は利率、y は期間です。

errno: ERANGE、アンダーフローおよびオーバーフロー状態の場合

呼び出しインターフェイス:

```
double compound(double x, double y);
long double compound(double x, double y);
float compoundf(float x, double y);
```

ERF

説明: erf 関数は、誤差関数の値を返します。

呼び出しインターフェイス:

```
double erf(double x);
long double erfl(long double x);
float erff(float x);
```

ERFC

説明: erfc 関数は、相補誤差関数の値を返します。

errno: ERANGE、アンダーフロー状態の場合

呼び出しインターフェイス:

```
double erfc(double x);
long double erfcl(long double x);
float erfcf(float x);
```

GAMMA

説明: gamma 関数は、gamma の絶対値の対数値を返します。

errno: ERANGE、オーバーフロー状態の場合

呼び出しインターフェイス:

```
double gamma(double x);
float gammaf(float x);
```

GAMMA_R

説明: gamma_r 関数は、gamma の絶対値の対数値を返します。gamma 関数の符号は、整数 signgam で返されます。

呼び出しインターフェイス:

```
double gamma_r(double x, int *signgam);
float gammaf_r(float x, int *signgam);
```

J0

説明: 0 次の x のベッセル関数 (第 1 種) を計算します。

呼び出しインターフェイス:

```
double j0(double x);
float j0f(float x);
```

J1

説明: 1 次の x のベッセル関数 (第 1 種) を計算します。

呼び出しインターフェイス:

```
double j1(double x);
float j1f(float x);
```

JN

説明: n 次の x のベッセル関数 (第 1 種) を計算します。

呼び出しインターフェイス:

```
double jn(int n, double x);
float jnf(int n, float x);
```

LGAMMA

説明: lgamma 関数は、gamma の絶対値の対数値を返します。

errno: ERANGE、オーバーフロー状態の場合

呼び出しインターフェイス:

```
double lgamma(double x);
long double lgammal(long double x);
float lgammaf(float x);
```

LGAMMA_R

説明: `lgamma_r` 関数は、gamma の絶対値の対数値を返します。`gamma` 関数の符号は、整数 `signgam` で返されます。

errno: ERANGE、オーバーフロー状態、x=0 または負の整数の場合

呼び出しインターフェイス:

```
double lgamma_r(double x, int *signgam);
long double lgamma_r(double x, int *signgam);
float lgammaf_r(float x, int *signgam);
```

TGAMMA

説明: `tgamma` 関数は、`x` の gamma 関数を計算します。

errno: EDOM、x=0 または負の整数の場合

呼び出しインターフェイス:

```
double tgamma(double x);
long double tgammal(long double x);
float tgammaf(float x);
```

Y0

説明: 0 次の `x` のベッセル関数 (第 2 種) を計算します。

errno: EDOM、x <= 0 の場合

呼び出しインターフェイス:

```
double y0(double x);
float y0f(float x);
```

Y1

説明: 1 次の `x` のベッセル関数 (第 2 種) を計算します。

errno: EDOM、x <= 0 の場合

呼び出しインターフェイス:

```
double y1(double x);
float y1f(float x);
```

YN

説明: `n` 次の `x` のベッセル関数 (第 2 種) を計算します。

errno: EDOM、x <= 0 の場合

呼び出しインターフェイス:

```
double yn(int n, double x);
float ynf(int n, float x);
```

丸め関数

インテル® 数値演算ライブラリは、次の丸め関数をサポートします:

CEIL

説明: `ceil` 関数は、`x` より小さくない値で、最も小さい整数値を浮動小数点数として返します。この関数は、Itanium® コンパイラによってインライン化されます。

呼び出しインターフェイス:

```
double ceil(double x);
long double ceill(long double x);
float ceilf(float x);
```

FLOOR

説明: `floor` 関数は、`x` より大きくない値で、最も大きい整数値を浮動小数点値として返します。この関数は、Itanium コンパイラによってインライン化されます。

呼び出しインターフェイス:

```
double floor(double x);
long double floorl(long double x);
float floorf(float x);
```

LLRINT

説明: `llrint` 関数は、現在の丸め方向を使用して、`long long int` として丸めた整数値を返します。

errno: ERANGE、値が非常に大きい場合

呼び出しインターフェイス:

```
long long int llrint(double x);
long long int llrintl(long double x);
long long int llrintf(float x);
```

LLROUND

説明: `llround` 関数は、`long long int` として丸めた整数値を返します。

errno: ERANGE、値が非常に大きい場合

呼び出しインターフェイス:

```
long long int llround(double x);
long long int llroundl(long double x);
long long int llroundf(float x);
```

LRINT

説明: `lrint` 関数は、現在の丸め方向を使用して、`long int` として丸めた整数値を返します。

呼び出しインターフェイス:

```
long int lrint(double x);
long int lrintl(long double x);
long int lrintf(float x);
```

LROUND

説明: `lround` 関数は、`long int` として丸めた整数値を返します。中間の場合は 0 から離れて丸められます。

errno: ERANGE、値が非常に大きい場合

呼び出しインターフェイス:

```
long int lround(double x);
long int lroundl(long double x);
long int lroundf(float x);
```

MODF

説明: modf 関数は、x の符号付き小数点部分の値を返し、*iptr に浮動小数点形式で整数部分をストアします。

呼び出しインターフェイス:

```
double modf(double x, double *iptr);
long double modfl(long double x, long double
*iptr);
float modff(float x, float *iptr);
```

NEARBYINT

説明: nearbyint 関数は、現在の丸め方向を使用して、浮動小数点数として丸めた整数値を返します。

呼び出しインターフェイス:

```
double nearbyint(double x);
long double nearbyintl(long double x);
float nearbyintf(float x);
```

RINT

説明: rint 関数は、現在の丸め方向を使用して、浮動小数点数として丸めた整数値を返します。

呼び出しインターフェイス:

```
double rint(double x);
long double rintl(long double x);
float rintf(float x);
```

ROUND

説明: round 関数は、浮動小数点数として丸めた整数値を返します。中間の場合は 0 から離れて丸められます。

呼び出しインターフェイス:

```
double round(double x);
long double roundl(long double x);
float roundf(float x);
```

TRUNC

説明: `trunc` 関数は、浮動小数点数として切り捨てられた整数値を返します。

呼び出しインターフェイス:

```
double trunc(double x);
long double truncl(long double x);
float truncf(float x);
```

剰余関数

インテル® 数値演算ライブラリは、次の剰余関数をサポートしています:

FMOD

説明: `fmod` 関数は、整数 `n` の値 `x-n*y` を返します。`y` が非ゼロの場合、その結果は、`x` と同じ符号を持ち、`y` の絶対値より小さい値になります。

errno: EDOM、x = 0 の場合

呼び出しインターフェイス:

```
double fmod(double x, double y);
long double fmodl(long double x, long double y);
float fmodf(float x, float y);
```

REMAINDER

説明: `remainder` 関数は、IEEE 標準によって求められる `x REM y` の値を返します。

呼び出しインターフェイス:

```
double remainder(double x, double y);
long double remainderl(long double x, long double
y);
float remainderf(float x, float y);
```

REMQUO

説明: remquo 関数は、x REM y の値を返します。quo で指定されたオブジェクトで、関数は、符号が x/y の符号で絶対値が x/y の整数の商の絶対値への合同剰余 2^{24} の値を返します。n は 3 以上の実装定義された整数です。

呼び出しインターフェイス:

```
double remquo(double x, double y, int *quo);
long double remquol(long double x, long double y,
int *quo);
float remquof(float x, float y, int *quo);
```

その他の関数

インテル® 数値演算ライブラリは、次に示すその他の関数をサポートします:

COPYSIGN

説明: copysign 関数は、x の絶対値と y の符号を返します。

呼び出しインターフェイス:

```
double copysign(double x, double y);
long double copysignl(long double x, long double
y);
float copysignf(float x, float y);
```

FABS

説明: fabs 関数は、x の絶対値を返します。

呼び出しインターフェイス:

```
double fabs(double x);
long double fabsl(long double x);
float fabsf(float x);
```

FDIM

説明: fdim 関数は、正の異なる値、x-y (x >y の場合) または +0 (x？y の場合) を返します。

errno: ERANGE、値が非常に大きい場合

呼び出しインターフェイス:

```
double fdim(double x, double y);
long double fdiml(long double x, long double y);
float fdimf(float x, float y);
```

FMA

説明: fma 関数は、(x*y)+z を返します。

呼び出しインターフェイス:

```
double fma(double x, double y, long double z);
long double fmal(long double x, long double y,
long double z);
float fmaf(float x, float y, long double z);
```

FMAX

説明: fmax 関数は、引数の最大値を返します。

呼び出しインターフェイス:

```
double fmax(double x, double y);
long double fmaxl(long double x, long double y);
float fmaxf(float x, float y);
```

FMIN

説明: fmin 関数は、引数の最小値を返します。

呼び出しインターフェイス:

```
double fmin(double x, double y);
long double fminl(long double x, long double y);
float fminf(float x, float y);
```

FPCLASSIFY

説明: fpclassify 関数は、その引数の値に適した番号分類マクロの値を返します。

呼び出しインターフェイス:

```
double fpclassify(double x);
long double fpclassifyl(long double x);
float fpclassifyf(float x);
```

ISFINITE

説明: finite 関数は、x が NaN または +/- 無限大ではない場合、1 を返します。それ以外の場合は、0 を返します。

呼び出しインターフェイス:

```
int isfinite(double x);
int isfinitel(long double x);
int isfinitef(float x);
```

ISGREATER

説明: isgreater 関数は、x が y よりも大きい場合、1 を返します。この関数は、無効な浮動小数点例外を増やしません。

呼び出しインターフェイス:

```
int isgreater(double x, double y);
int isgreaterl(long double x, long double y);
int isgreaterf(float x, float y);
```

ISGREATEREQUAL

説明: isgreaterequal 関数は、x が y 以上の場合、1 を返します。この関数は、無効な浮動小数点例外を増やしません。

呼び出しインターフェイス:

```
int isgreaterequal(double x, double y);
int isgreaterequall(long double x, long double
y);
int isgreaterequalf(float x, float y);
```

ISINF

説明: isinf 関数は、引数が無限値を持つ場合のみ、非ゼロ値を返します。

呼び出しインターフェイス:

```
int isinf(double x);
int isinfl(long double x);
int isinff(float x);
```

ISLESS

説明: isless 関数は、x が y よりも小さい場合、1 を返します。この関数は、無効な浮動小数点例外を増やしません。

呼び出しインターフェイス:

```
int isless(double x, double y);
int islessl(long double x, long double y);
int islessf(float x, float y);
```

ISLESSEQUAL

説明: islessequal 関数は、x が y 以下の場合、1 を返します。この関数は、無効な浮動小数点例外を増やしません。

呼び出しインターフェイス:

```
int islessequal(double x, double y);
int islessequall(long double x, long double y);
int islessequalf(float x, float y);
```

ISLESSGREATER

説明: islessgreater 関数は、x が y よりも小さいまたは大きい場合、1 を返します。この関数は、無効な浮動小数点例外を増やしません。

呼び出しインターフェイス:

```
int islessgreater(double x, double y);
int islessgreaterl(long double x, long double y);
int islessgreaterf(float x, float y);
```

ISNAN

説明: isnan 関数は、x が NaN 値を持つ場合のみ、非ゼロ値を返します。

呼び出しインターフェイス:

```
int isnan(double x);
int isnanl(long double x);
int isnanf(float x);
```

ISNORMAL

説明: isnormal 関数は、x が平均値の場合のみ、非ゼロ値を返します。

呼び出しインターフェイス:

```
int isnormal(double x);
int isnormall(long double x);
int isnormalf(float x);
```

ISUNORDERED

説明: isunordered 関数は、x または y のいずれかが NaN の場合、1 を返します。この関数は、無効な浮動小数点例外を増やしません。

呼び出しインターフェイス:

```
int isunordered(double x, double y);
int isunorderedl(long double x, long double y);
int isunorderedf(float x, float y);
```

NEXTAFTER

説明: nextafter 関数は、y 方向での x の次に表現可能な値を指定した形式で返します。

errno: ERANGE、値が非常に大きい場合

呼び出しインターフェイス:

```
double nextafter(double x, double y);
long double nextafterl(long double x, long double
y);
float nextafterf(float x, float y);
```

NEXTTOWARD

説明: nexttoward 関数は、y 方向での x の次に表現可能な値を指定した形式で返します。x と y が等しい場合、関数のタイプに変換した y を返します。

errno: ERANGE、値が非常に大きい場合

呼び出しインターフェイス:

```
double nexttoward(double x, double y);
long double nexttowardl(long double x, long
double y);
float nexttowardf(float x, float y);
```

SIGNBIT

説明: signbit 関数は、x の符号が負の場合のみ、非ゼロ値を返します。

呼び出しインターフェイス:

```
int signbit(double x);
int signbitl(long double x);
int signbitf(float x);
```

SIGNIFICAND

説明: significand 関数は、[1,2) の範囲で x の仮数を返します。x が 0、NaN、または +/- 無限大と等しい場合、オリジナルの x が返されます。

呼び出しインターフェイス:

```
double significand(double x);
long double significandl(long double x);
float significandf(float x);
```

複素数関数

インテル® 数値演算ライブラリは、次の複素数関数をサポートします:

CABS

説明: cabs 関数は、z の複素数絶対値を返します。

呼び出しインターフェイス:

```
double cabs(double _Complex z);
long double cabs(long double _Complex z);
float cabsf(float _Complex z);
```

CACOS

説明: cacos 関数は、z の複素数逆余弦を返します。

呼び出しインターフェイス:

```
double _Complex cacos(double _Complex z);
long double _Complex cacosl(long double _Complex
```

```
z);
float _Complex cacosf(float _Complex z);
```

CACOSH

説明: cacosh 関数は、z の複素数逆双曲線余弦を返します。

呼び出しインターフェイス:

```
double _Complex cacosh(double _Complex z);
long double _Complex cacoshl(long double _Complex
z);
float _Complex cacoshf(float _Complex z);
```

CARG

説明: carg 関数は、[-pi, +pi] の範囲で引数の値を返します。

呼び出しインターフェイス:

```
double carg(double _Complex z);
long double cargl(long double _Complex z);
float cargf(float _Complex z);
```

CASIN

説明: casin 関数は、z の複素数逆正弦を返します。

呼び出しインターフェイス:

```
double _Complex casin(double _Complex z);
long double _Complex casinl(long double _Complex
z);
float _Complex casinf(float _Complex z);
```

CASINH

説明: casinh 関数は、z の複素数逆双曲線正弦を返します。

呼び出しインターフェイス:

```
double _Complex casinh(double _Complex z);
long double _Complex casinhl(long double _Complex
z);
float _Complex casinhf(float _Complex z);
```

CATAN

説明: catan 関数は、z の複素数逆正接を返します。

呼び出しインターフェイス:

```
double _Complex catan(double _Complex z);
long double _Complex catanl(long double _Complex
z);
float _Complex catanf(float _Complex z);
```

CATANH

説明: catanh 関数は、z の複素数逆双曲線正接を返します。

呼び出しインターフェイス:

```
double _Complex catanh(double _Complex z);
long double _Complex catanhl(long double _Complex
z);
float _Complex catanhf(float _Complex z);
```

CCOS

説明: ccos 関数は、z の複素数余弦を返します。

呼び出しインターフェイス:

```
double _Complex ccos(double _Complex z);
long double _Complex ccosl(long double _Complex
z);
float _Complex ccosf(float _Complex z);
```

CCOSH

説明: ccosh 関数は、z の複素数双曲線余弦を返します。

呼び出しインターフェイス:

```
double _Complex ccosh(double _Complex z);
long double _Complex ccoshl(long double _Complex
z);
float _Complex ccoshf(float _Complex z);
```

CEXP

説明: cexp 関数は、e^z を計算します。

呼び出しインターフェイス:

```
double _Complex cexp(double _Complex z);
long double _Complex cexpl(long double _Complex
z);
float _Complex cexpf(float _Complex z);
```

CEXP10

説明: cexp10 関数は、10^z を計算します。

呼び出しインターフェイス:

```
double _Complex cexp10(double _Complex z);
long double _Complex cexp10l(long double _Complex
z);
float _Complex cexp10f(float _Complex z);
```

CIMAG

説明: cimag 関数は、z の虚数部分値を返します。

呼び出しインターフェイス:

```
double cimag(double _Complex z);
long double cimag(long double _Complex z);
float cimagf(float _Complex z);
```

CIS

説明: cis 関数は、弧度法で z の余弦と正弦 (複素数値) を返します。

呼び出しインターフェイス:

```
double _Complex cis(double z);
long double _Complex cis(long double z);
float _Complex cis(float z);
```

CISD

説明: cis 関数は、角度法で z の余弦と正弦 (複素数値) を返します。

呼び出しインターフェイス:

```
double _Complex cis(double z);
long double _Complex cis(long double z);
float _Complex cis(float z);
```

CLOG

説明: clog 関数は、z の複素数自然対数を返します。

呼び出しインターフェイス:

```
double _Complex clog(double _Complex z);
long double _Complex clogl(long double _Complex
z);
float _Complex clogf(float _Complex z);
```

CLOG2

説明: clog2 関数は、基数 2 の z の複素数対数を返します。

呼び出しインターフェイス:

```
double _Complex clog2(double _Complex z);
long double _Complex clog2l(long double _Complex
z);
float _Complex clog2f(float _Complex z);
```

CONJ

説明: conj 関数は、虚数部の正弦を反転して、z の共役複素数を返します。

呼び出しインターフェイス:

```
double _Complex conj(double _Complex z);
long double _Complex conj(long double _Complex
z);
float _Complex conjf(float _Complex z);
```

CPOW

説明: cpow 関数は、複素数べき乗関数 x^y を返します。

呼び出しインターフェイス:

```
double _Complex cpow(double _Complex x, double
_Complex y);
long double _Complex cpowl(long double _Complex x,
double _Complex y);
float _Complex cpowf(float _Complex x, float
_Complex y);
```

CPROJ

説明: cproj 関数は、リーマン球上での z の投影を返します。

呼び出しインターフェイス:

```
double _Complex cproj(double _Complex z);
long double _Complex cproj(long double _Complex
z);
float _Complex cprojf(float _Complex z);
```

CREAL

説明: creal 関数は、z の実数部の値を返します。

呼び出しインターフェイス:

```
double creal(double _Complex z);
long double creal(long double _Complex z);
float crealf(float _Complex z);
```

CSIN

説明: csin 関数は、z の複素数正弦を返します。

呼び出しインターフェイス:

```
double _Complex csin(double _Complex z);
long double _Complex csinl(long double _Complex
z);
float _Complex csinf(float _Complex z);
```

CSINH

説明: csinh 関数は、z の複素数双曲線正弦を返します。

呼び出しインターフェイス:

```
double _Complex csinh(double _Complex z);
long double _Complex csinl(long double _Complex
z);
float _Complex csinhf(float _Complex z);
```

CSQRT

説明: csqrt 関数は、z の複素数平方根を返します。

呼び出しインターフェイス:

```
double _Complex csqrt(double _Complex z);
long double _Complex csqrtl(long double _Complex
```

```
z);
float _Complex csqrtf(float _Complex z);
```

CTAN

説明: ctan 関数は、z の複素数正接を返します。

呼び出しインターフェイス:

```
double _Complex ctan(double _Complex z);
long double _Complex ctanl(long double _Complex
z);
float _Complex ctanf(float _Complex z);
```

CTANH

説明: ctanh 関数は、z の複素数双曲線正接を返します。

呼び出しインターフェイス:

```
double _Complex ctanh(double _Complex z);
long double _Complex ctanhl(long double _Complex
z);
float _Complex ctanhf(float _Complex z);
```

C99 マクロ

インテル® 数値演算ライブラリと mathimf.h ヘッダファイルは、次の C99 マクロをサポートします:

```
int fpclassify(x);

int isfinite(x);

int isgreater(x, y);

int isgreaterequal(x, y);

int isinf(x);

int isless(x, y);

int islessequal(x, y);

int islessgreater(x, y);

int isnan(x);
```

```
int isnormal(x);
int isunordered(x, y);
int signbit(x);
```

「その他の関数」も参照してください。

インテル® C++ 組込み関数リファレンス

はじめに

概要: 組込み関数

インテル® Pentium® 4 プロセッサおよびその他のプロセッサには、最適化されたマルチメディア・アプリケーションの開発用の命令を用意しています。これらの命令は、以前に実装した命令を拡張したものです。これらの命令は、SIMD (Single Instruction, Multiple Data) テクノロジを使用します。SIMD 命令を使用してデータ要素を並列処理すると、大量のマルチメディア・データ・ストリームを処理するアプリケーションのパフォーマンスが大きく向上します。インテル® Itanium® プロセッサもまた、そうした命令をサポートします。

これらの命令を使用する最も直接的な方法は、ソースコード内でインライン・アセンブリ言語命令を使用することです。しかし、これは時間のかかる作業です。また、一部のコンパイラは、アセンブリ言語のインライン・プログラミングに対応していません。そこで、インテルでは、組込み関数と呼ばれる API 拡張機能セットを使用する方法を用意しました。

組込み関数は、ハードウェア・レジスタを直接操作するのではなく、C の関数呼び出しと C の変数の構文を使用できる、コーディング用の拡張機能です。組込み関数によって、プログラマは、アセンブリ言語のプログラミングとレジスタの管理を行う必要がなくなります。また、コンパイラは、命令のスケジューリングを最適化して、実行ファイルの処理速度を上げられます。

さらに、プログラマは、Itanium プロセッサ向けのネイティブ組込み関数を使用して、C および C++ 言語の標準的な構文では生成できない Itanium 命令を利用できます。また、インテル® C++ コンパイラは、すべての IA-32 プラットフォームおよび Itanium ベースのプラットフォーム上で動作する汎用組込み関数もサポートします。

組込み関数の詳細については、次の参考資料を参照してください:

『IA-32 インテル® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、中巻: 命令セット・リファレンス』インテル社、資料番号 245471J

各インテル・プロセッサの組込み関数への対応

プロセッサ:	MMX® テクノロジの組込み関数	ストリーミング SIMD 拡張命令	ストリーミング SIMD 拡張命令 2	Itanium プロセッサ命令
Itanium プロセッサ	X	X	N/A	X
Pentium 4 プロセッサ	X	X	X	N/A
Pentium III プロセッサ	X	X	N/A	N/A
Pentium II プロセッサ	X	N/A	N/A	N/A
MMX テクノロジ Pentium プロセッサ	X	N/A	N/A	N/A
Pentium Pro プロセッサ	N/A	N/A	N/A	N/A
Pentium プロセッサ	N/A	N/A	N/A	N/A

組込み関数を使用するメリット

組込み関数を使用する大きなメリットは、従来のコーディング手法では使用できない重要な機能を利用できることです。組込み関数を使用すれば、アセンブリ言語の代わりに、C の関数呼び出しと変数の構文を使用してコーディングできます。ほとんどの MMX® テクノロジ、ストリーミング SIMD 拡張命令およびストリーミング SIMD 拡張命令 2 の組込み関数は、これらの命令を直接使用する C の関数に対応します。これによって、プログラマは、レジスタを管理する必要がなくなり、コンパイラは命令のスケジューリングを最適化できます。

MMX テクノロジ命令の組込み関数とストリーミング SIMD 拡張命令の組込み関数では、次の新しい機能を使用します:

- 新しいレジスタ--最大 128 ビットのパックドデータの最適な SIMD 処理が可能です。
- 新しいデータ型--最大 16 要素のデータを 1 つのレジスタにパックできます。

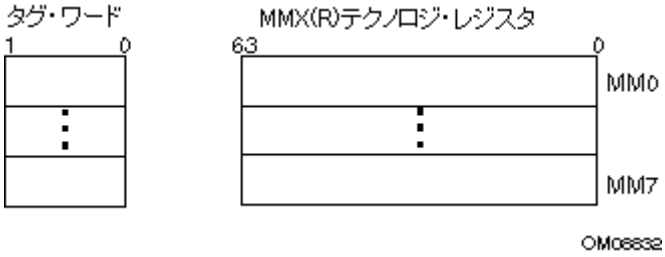
ストリーミング SIMD 拡張命令 2 の組込み関数は、IA-32 についてのみ定義されており、Itanium® ベースのシステムについては定義されていません。ストリーミング SIMD 拡張命令 2 は、128 ビットデータ (2 個の 64 ビット倍精度浮動小数点値) を操

作します。Itanium アーキテクチャでは、ストリーミング SIMD 拡張命令 2 をサポートしていないため、Itanium ベースのシステム上では、ストリーミング SIMD 拡張命令 2 は実行できません。

新しいレジスタ

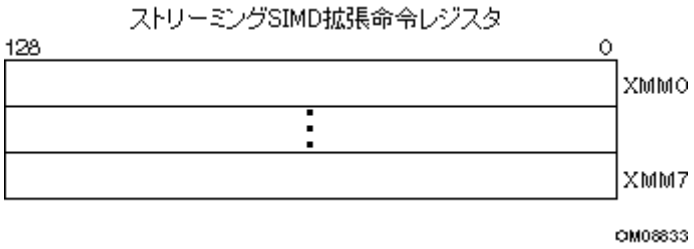
新しいプロセッサ・アーキテクチャには、新しいレジスタセットが追加されています。MMX 命令は、8 個の 64 ビットレジスタ (mm0～mm7) を使用します。これらのレジスタは、浮動小数点スタックレジスタに別名を付けて使用されます。

MMX テクノロジ・レジスタ



ストリーミング SIMD 拡張命令レジスタ

ストリーミング SIMD 拡張命令レジスタは、8 個の 128 ビットレジスタ (xmm0～xmm7) を使用します。



これらの新しいデータレジスタによって、データ要素の並列処理が可能になります。各レジスタが複数のデータ要素を保持できるため、プロセッサは複数のデータ要素を同時に処理できます。このような処理方法は、SIMD (Single Instruction, Multiple Data) 処理と呼ばれます。

新しい拡張命令セットのそれぞれの計算命令とデータ操作命令について、その命令を直接に実行する C 組込み関数を用意しています。これにより、プログラマは、レジスタの管理とアセンブリ言語のプログラミングを行う必要がなくなります。また、コンパイラは、命令のスケジューリングを最適化して、実行ファイルの処理速度を上げられます。

注

MMX テクノロジ・レジスタと XMM レジスタは、それぞれ MMX テクノロジの組込み関数とストリーミング SIMD 拡張命令/ストリーミング SIMD 拡張命令 2 の組込み関数を実行するために、IA-32 プラットフォーム上で使用される SIMD レジスタです。Itanium ベースのプラットフォーム上では、MMX テクノロジの組込み関数とストリーミング SIMD 拡張命令の組込み関数は、64 ビット汎用レジスタと、80 ビット浮動小数点レジスタの 64 ビットの仮数部を使用します。

データ型

組込み関数は、4 つの新しい C データ型をオペランドとして使用します。4 つのデータ型は、組込み関数に対するオペランドとして使用される新しいレジスタを表しています。次の表に、プロセッサごとに、これらの新しいデータ型が使用可能かどうかを示します。

新しいデータ型への対応

新しいデータ型	MMX テクノロジ	ストリーミング SIMD 拡張命令	拡張命令 2	Itanium プロセッサ
__m64	X	X	X	X
__m128	N/A	X	X	X
__m128d	N/A	N/A	X	X
__m128i	N/A	N/A	X	X

__m64 データ型

__m64 データ型は、MMX テクノロジの組込み関数に使用される MMX テクノロジ・レジスタの内容を表します。__m64 データ型は、8 個の 8 ビット値、4 個の 16 ビット値、2 個の 32 ビット値、または 1 個の 64 ビット値を保持できます。

__m128 データ型

__m128 データ型は、ストリーミング SIMD 拡張命令の組込み関数に使用するストリーミング SIMD 拡張命令レジスタの内容を表します。__m128 データ型は、4 つの 32 ビット浮動小数点値を保持できます。

__m128d データ型は、2 つの 64 ビット浮動小数点値を保持できます。

__m128i データ型は、16 個の 8 ビット整数値、8 個の 16 ビット整数値、4 個の 32 ビット整数値、または 2 個の 64 ビット整数値を保持できます。

コンパイラは、__m128 型のローカルデータとグローバル・データのアライメントを、スタック上の 16 バイト境界に合わせます。integer 型、float 型、または double 型の配列のアライメントを合わせるには、declspec 文を使用します。

新しいデータ型を使用する際のガイドライン

これらの新しいデータ型は、基本的な ANSI C データ型ではありません。このため、次のような使用上の制限があります:

- 新しいデータ型は、代入文の左辺または右辺で、戻り値またはパラメータとして使用してください。他の算術式 (″+″、″-″など) にこのデータ型を使用することはできません。
- 新しいデータ型は、バイト要素/構造にアクセスするための共用体など、共用体などの集合体のオブジェクトとして使用してください。
- 新しいデータ型は、本書で説明する組込み関数でのみ使用してください。新しいデータ型は、代入文の両辺で、関数呼び出しに渡すパラメータおよび関数呼び出しからの戻り値として使用できます。

命名と使用する構文

ほとんどの組込み関数名は、次の表記規則に従います:

_mm_<intrin_op>_<suffix>

<intrin_op>	組込み関数の基本操作を示します。例えば、加算の場合は add、減算の場合は sub になります。
<suffix>	命令の操作対象となるデータの型を示します。各サフィックスの最初の 1 文字または 2 文字は、データがパックドデータ (p)、拡張パックドデータ (ep)、またはスカラデータ (s) であることを示します。その他の文字は、次のとおりデータ型を示します: • s 単精度浮動小数点値 • d 倍精度浮動小数点値 • i128 符号付き 128 ビット整数 • i64 符号付き 64 ビット整数 • u64 符号なし 64 ビット整数 • i32 符号付き 32 ビット整数 • u32 符号なし 32 ビット整数 • i16 符号付き 16 ビット整数 • u16 符号なし 16 ビット整数 • i8 符号付き 8 ビット整数 • u8 符号なし 8 ビット整数

変数名を付加した数字は、パックされたオブジェクトの要素を示します。例えば、r0 は r の最下位ワードです。一部の組込み関数は、2 つ以上の命令で実行するため、"複合組込み関数" と呼ばれます。

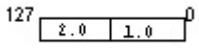
パックされた値は、右から左の順序で表現し、最下位の値がスカラ操作に使用されます。次の操作の例について考えます:

```
double a[2] = {1.0, 2.0};

__m128d t = _mm_load_pd(a);
```

上の操作の結果は、次のそれぞれの操作と同じになります:

```
__m128d t = _mm_set_pd(2.0, 1.0);

__m128d t = _mm_setr_pd(1.0, 2.0);
```

つまり、値 t を保持する xmm レジスタは、次のようになります:



"スカラ" 要素は 1.0 です。一部の組込み関数では、命令の性質上、引数として即値 (定数整数リテラル) を指定しなければなりません。

組込み関数の構文

コードの中で組込み関数を使用するには、次の構文の行を挿入します:

```
data_type intrinsic_name (parameters)
```

各項の意味は次のとおりです。

data_type	戻りデータ型。void、int、__m64、__m128、__m128d、__m128i、__int64 のうちのいずれかです。すべてのインテル® アーキテクチャでサポートする組込み関数は、組込み関数の構文の定義に従って、その他のデータ型を返す場合があります。
intrinsic_name	この名前は、実際の命令をインライン展開する代わりに C++ コード内で使用できる関数として機能します。
parameters	各組込み関数が要求するパラメータを表します。

すべての IA プロセッサでサポートされる組込み関数

概要: すべての IA プロセッサでサポートされる組込み関数

このセクションで説明する組込み関数は、すべての IA-32 プラットフォームおよび Itanium® ベースのプラットフォーム上で動作します。これらの組込み関数は、プログラマにとって便利なものです。これらの組込み関数は、次のグループに分類されます:

- 整数演算に関連する組込み関数
- 浮動小数点に関連する組込み関数
- 文字列とブロックのコピーに関連する組込み関数
- その他

整数演算に関連する組込み関数

組込み関数	説明
`int abs(int)`	整数の絶対値を返します。
`long labs(long)`	long 型整数の絶対値を返します。
`unsigned long _lrotl(unsigned long value, int shift)`	符号なし long 型整数の各ビットを左にローテートします。
`unsigned long _lrotr(unsigned long value, int shift)`	符号なし long 型整数の各ビットを右にローテートします。
`unsigned int __rotl(unsigned int value, int shift)`	符号なし整数の各ビットを左にローテートします。
`unsigned int __rotr(unsigned int value, int shift)`	符号なし整数の各ビットを右にローテートします。

注

ローテート組込み関数内で定数シフト値を渡すと、パフォーマンスが向上します。

浮動小数点に関連する組込み関数

組込み関数	説明
`double fabs(double)`	浮動小数点値の絶対値を返します。
`double log(double)`	倍精度を使用して、自然対数 ln(x), x>0 を返します。

float logf(float)	単精度を使用して、自然対数 ln(x), x>0 を返します。
double log10(double)	倍精度を使用して、10 を基数とする対数 log10(x), x>0 を返します。
float log10f(float)	単精度を使用して、10 を基数とする対数 log10(x), x>0 を返します。
double exp(double)	倍精度を使用して、指数関数を返します。
float expf(float)	単精度を使用して、指数関数を返します。
double pow(double, double)	倍精度を使用して、x を y 乗した値を返します。
float powf(float, float)	単精度を使用して、x を y 乗した値を返します。
double sin(double)	倍精度を使用して、x の正弦を返します。
float sinf(float)	単精度を使用して、x の正弦を返します。
double cos(double)	倍精度を使用して、x の余弦を返します。
float cosf(float)	単精度を使用して、x の余弦を返します。
double tan(double)	倍精度を使用して、x の正接を返します。
float tanf(float)	単精度を使用して、x の正接を返します。
double acos(double)	倍精度を使用して、x の逆余弦を返します。
float acosf(float)	単精度を使用して、x の逆余弦を返します。
double acosh(double)	倍精度を使用して、引数の逆双曲線余弦を計算します。
float acoshf(float)	単精度を使用して、引数の逆双曲線余弦を計算します。
double asin(double)	倍精度を使用して、引数の逆正弦を計算します。
float asinf(float)	単精度を使用して、引数の逆正弦を計算します。
double asinh(double)	倍精度を使用して、引数の逆双曲線正弦を計算します。
float asinhf(float)	単精度を使用して、引数の逆双曲線正弦を計算します。
double atan(double)	倍精度を使用して、引数の逆正接を計算します。
float atanf(float)	単精度を使用して、引数の逆正接を計算します。
double atanh(double)	倍精度を使用して、引数の逆双曲線正接を計算します。
float atanhf(float)	単精度を使用して、引数の逆双曲線正接を計算します。
float cabs(double)**	複素数の絶対値を計算します。
double ceil(double)	引数より小さくない倍精度引数の整数値のうち、最も小さい値を計算します。
float ceilf(float)	引数より小さくない単精度引数の整数値のうち、最も小さい値を計算します。
double cosh(double)	倍精度引数の双曲線余弦を計算します。

float coshf(float)	単精度引数の双曲線余弦を計算します。
float fabsf(float)	単精度引数の絶対値を計算します。
double floor(double)	引数より大きくない倍精度引数の整数値のうち、最も大きい値を計算します。
float floorf(float)	引数より大きくない単精度引数の整数値のうち、最も大きい値を計算します。
double fmod(double)	倍精度を使用して、第 1 の引数を第 2 の引数で割ったときの浮動小数点剰余を計算します。
float fmodf(float)	単精度を使用して、第 1 の引数を第 2 の引数で割ったときの浮動小数点剰余を計算します。
double hypot(double, double)	倍精度を使用して、直角三角形の斜辺の長さを計算します。
float hypotf(float)	単精度を使用して、直角三角形の斜辺の長さを計算します。
double rint(double)	IEEE 丸めモードを使用して、倍精度で表現される整数値を計算します。
float rintf(float)	IEEE 丸めモードを使用して、単精度で表現される整数値を計算します。
double sinh(double)	倍精度引数の双曲線正弦を計算します。
float sinhf(float)	単精度引数の双曲線正弦を計算します。
float sqrtf(float)	単精度引数の平方根を計算します。
double tanh(double)	倍精度引数の双曲線正接を計算します。
float tanhf(float)	単精度引数の双曲線正接を計算します。

* Itanium® ベースのシステムではサポートしていません。

** この場合の double は、2 個の単精度 (32 ビット浮動小数点) 要素 (実数部と虚数部) で構成される複素数です。

文字列とブロックのコピーに関連する組込み関数

次の関数は、Itanium® ベースのプラットフォーム上では組込み関数としてサポートしていません。

組込み関数	説明
char *_strset(char *, _int32)	文字列のすべての文字を固定値に設定し

	ます。
void *memcmp(const void *cs, const void *ct, size_t n)	メモリの 2 つの領域を比較します。cs<ct の場合は <0、cs=ct の場合は 0、cs>ct の場合は >0 を返します。
void *memcpy(void *s, const void *ct, size_t n)	メモリからコピーします。s を返します。
void *memset(void * s, int c, size_t n)	メモリを固定値に設定します。s を返します。
char *strcat(char * s, const char * ct)	文字列に追加します。s を返します。
int *strcmp(const char *, const char *)	2 つの文字列を比較します。cs<ct の場合は <0、cs=ct の場合は 0、cs>ct の場合は >0 を返します。
char *strcpy(char * s, const char * ct)	文字列をコピーします。s を返します。
size_t strlen(const char * cs)	文字列 cs の長さを返します。
int strncmp(char *, char *, int)	指定した文字数だけ、2 つの文字列を比較します。
int strncpy(char *, char *, int)	指定した文字数だけ、文字列をコピーします。

組込み関数

次の組込み関数は IA-32 および Itanium® アーキテクチャで共通です。

組込み関数	説明
void *_alloca(int)	バッファを割り当てます。
int _setjmp(jmp_buf)*	setjmp() の高速版。終了処理が省略されます。呼び出し先セーブのレジスタ、スタックポインタ、およびリターンアドレスを保存します。

_exception_code(void)	例外コードを返します。
_exception_info(void)	例外情報を返します。
_abnormal_termination(void)	終了ハンドラ以外では起動できません。対応する最後に試行する領域が予定より早く終了したために終了ハンドラが起動された場合は、TRUE を返します。
void _enable()	割り込みを有効にします。
void _disable()	割り込みを無効にします。
int _in_byte(int)	IA-32 命令 IN に対応付けられる組込み関数。引数で指定されたポートからデータバイトを転送します。
int _in_dword(int)	IA-32 命令 IN に対応付けられる組込み関数。引数で指定されたポートからダブルワードを転送します。
int _in_word(int)	IA-32 命令 IN に対応付けられる組込み関数。引数で指定されたポートからワードを転送します。
int _inp(int)	_in_byte と同じです。
int _inpd(int)	_in_dword と同じです。
int _inpw(int)	_in_word と同じです。
int _out_byte(int, int)	IA-32 命令 OUT に対応付けられる組込み関数。第 2 の引数内のデータバイトを、第 1 の引数で指定するポートに転送します。
int _out_dword(int, int)	IA-32 命令 OUT に対応付けられる組込み関数。第 2 の引数内のダブルワードを、第 1 の引数で指定するポートに転送します。
int _out_word(int, int)	IA-32 命令 OUT に対応付けられる組込み関数。第 2 の引数内のワードを、第 1 の引数で指定するポートに転送します。
int _outp(int, int)	_out_byte と同じです。
int _outpd(int, int)	_out_dword と同じです。
int _outpw(int, int)	_out_word と同じです。

MMX® テクノロジの組込み関数

MMX® テクノロジのサポート

MMX® テクノロジは、インテル® アーキテクチャ (IA) 命令セットを拡張したものです。MMX テクノロジ命令セットには、57 のオペコード、64 ビット・クワッドワード・データ型、

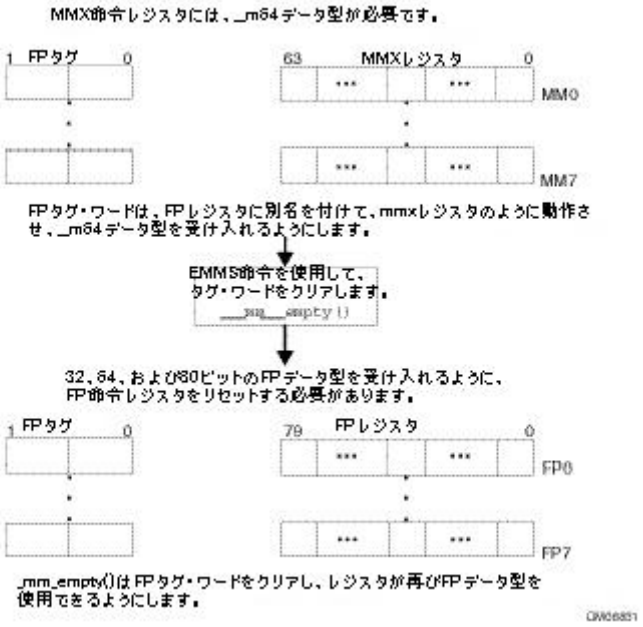
および 8 個の 64 ビットレジスタが追加されました。各レジスタは、レジスタ名 mm0～mm7 を使用して、直接にアドレス指定できます。

MMX テクノロジの組込み関数のプロトタイプは、ヘッダファイル mmintrin.h 内にあります。

EMMS 命令: 必要な理由

EMMS 命令には、レジスタを一度空にして、別のデータに使用できるようにする働きがあります。例えば、MMX® テクノロジ命令は、FP レジスタ内の FP タグワードを自動的に有効にして、__m64 型を使用できるようにします。これによって、FP レジスタセットがリセットされ、MMX テクノロジ・レジスタセットとして別名が付けられます。再び FP レジスタセットとして使用できるようにするには、EMMS 命令か _mm_empty() 組込み関数を使用して、レジスタの状態をリセットします。

MMX テクノロジ命令の実行後、EMMS 命令でレジスタをリセットする理由



警告

MMX 命令の実行後、マルチメディア・ステートを空にせずに浮動小数点命令を実行すると、予期しない動作が行われたり、パフォーマンスが低下する場合があります。

EMMS を使用する際のガイドライン

次に、EMMS を使用する際のガイドラインを示します:

- Itanium® ベース・システム上では使用しないでください。EMMS 組込み関数がサポートされている場合でも、Itanium ベースのシステム上には、MMX® 命令やストリーミング SIMD 拡張命令用の特殊なレジスタ (またはオーバーレイ) はありません。
- MMX 命令の実行後、次に浮動小数点 (FP) 命令を実行する (例えば、float、double、または long double 型データを計算する) 場合は、その前に _mm_empty( ) を使用してください。インテル® C++ コンパイラを使用して MMX 命令を生成したコードでは、次のすべての状況に注意してください:
  - MMX テクノロジ組込み関数を使用する場合
  - __m64 データ型を使用する、ストリーミング SIMD 拡張命令の整数組込み関数を使用する場合
  - __m64 データ型の変数を参照する場合
  - インライン・アセンブリによって MMX 命令を使用する場合
- MMX 命令の前に _mm_empty() を使用しないでください。MMX 命令の前に _mm_empty() を使用すると、何のメリットもない操作 (no-op) が行われます。
- FP 命令を使用する操作と MMX 命令を使用する操作は、別々の関数に分けてください。そうすれば、性能に大きな影響を及ぼすループの中で、マルチメディア・ステートを空にする必要はありません。
- 実行時に __m64 および FP データ型を初期化するときは、_mm_empty() を使用してください。これによって、データ型が切り替わる間に、レジスタが確実にリセットされます。
- 次の「正しい使用方法」のコーディング例を参照してください。

誤った使用方法	正しい使用方法
`__m64 x = _m_paddd(y, z);` `float f = init();`	`__m64 x = _m_paddd(y, z);` `float f = (_mm_empty(), init());`

EMMS の詳細については、http://www.intel.co.jp/jp/developer/ Web サイトにアクセスして、EMMS を検索してください。

MMX® テクノロジの一般的な組込み関数

MMX® テクノロジの組込み関数のプロトタイプは、ヘッダファイル mmintrin.h 内にあります。

組込み関数の名前	別名	対応する命令	操作	符号	飽和
_m_empty	_mm_empty	EMMS	MM ステートを空にする	--	--
_m_from_int	_mm_cvtsi32_si64	MOVD	int からの変換	--	--
_m_to_int	_mm_cvtsi64_si32	MOVD	int からの変換	--	--
_m_packsswb	_mm_packs_pi16	PACKSSWB	Pack	符号付き	可
_m_packssdw	_mm_packs_pi32	PACKSSDW	Pack	符号付き	可
_m_packuswb	_mm_packs_pu16	PACKUSWB	Pack	符号なし	可
_m_punpckhbw	_mm_unpackhi_pi8	PUNPCKHBW	インターリーブ	--	--
_m_punpckhwd	_mm_unpackhi_pi16	PUNPCKHWD	インターリーブ	--	--
_m_punpckhdq	_mm_unpackhi_pi32	PUNPCKHDQ	インターリーブ	--	--
_m_punpcklbw	_mm_unpacklo_pi8	PUNPCKLBW	インターリーブ	--	--
_m_punpcklwd	_mm_unpacklo_pi16	PUNPCKLWD	インターリーブ	--	--
_m_punpckldq	_mm_unpacklo_pi32	PUNPCKLDQ	インターリーブ	--	--

```
void _m_empty(void)
```

マルチメディア・ステートを空にします。

```
__m64 _m_from_int(int i)
```

整数オブジェクト i を、64 ビットの __m64 オブジェクトに変換します。整数値は 0 で拡張して 64 ビットに変換します。

```
int _m_to_int(__m64 m)
```

__m64 オブジェクト m の下位 32 ビットを、整数に変換します。

```
__m64 _m_packsswb(__m64 m1, __m64 m2)
```

符号付き飽和処理を使用して、m1 の 4 つの 16 ビット値を、結果の下位 4 つの 8 ビット値にパックします。符号付き飽和処理を使用して、m2 の 4 つの 16 ビット値を、結果の上位 4 つの 8 ビット値にパックします。

```
__m64 _m_packssdw(__m64 m1, __m64 m2)
```

符号付き飽和処理を使用して、m1 の 2 つの 32 ビット値を、結果の下位 2 つの 16 ビット値にパックします。符号付き飽和処理を使用して、m2 の 2 つの 32 ビット値を、結果の上位 2 つの 16 ビット値にパックします。

```
__m64 _m_packuswb(__m64 m1, __m64 m2)
```

符号なし飽和処理を使用して、m1 の 4 つの 16 ビット値を、結果の下位 4 つの 8 ビット値にパックします。符号なし飽和処理を使用して、m2 の 4 つの 16 ビット値を、結果の上位 4 つの 8 ビット値にパックします。

```
__m64 _m_punpckhbw(__m64 m1, __m64 m2)
```

m1 の上位半分の 4 つの 8 ビット値と、m2 の上位半分の 4 つの値をインターリーブ (交互に配置) します。インターリーブは m1 のデータから始めます。

```
__m64 _m_punpckhwd(__m64 m1, __m64 m2)
```

m1 の上位半分の 2 つの 16 ビット値と、m2 の上位半分の 2 つの値をインターリーブします。インターリーブは m1 のデータから始めます。

```
__m64 _m_punpckhdq(__m64 m1, __m64 m2)
```

m1 の上位半分の 32 ビット値と、m2 の上位半分の 32 ビット値をインターリーブします。インターリーブは m1 のデータから始めます。

```
__m64 _m_punpcklbw(__m64 m1, __m64 m2)
```

m1 の下位半分の 4 つの 8 ビット値と、m2 の下位半分の 4 つの値をインターリーブします。インターリーブは m1 のデータから始めます。

```
__m64 _m_punpcklwd(__m64 m1, __m64 m2)
```

m1 の下位半分の 2 つの 16 ビット値と、m2 の下位半分の 2 つの値をインターリーブします。インターリーブは m1 のデータから始めます。

```
__m64 _m_punpckldq(__m64 m1, __m64 m2)
```

m1 の下位半分の 32 ビット値と、m2 の下位半分の 32 ビット値をインターリーブします。インターリーブは m1 のデータから始めます。

MMX® テクノロジのパックド算術演算組込み関数

MMX® テクノロジの組込み関数のプロトタイプは、ヘッダファイル mmintrin.h 内にあります。

組込み関数の名前	別名	対応する命令	操作	符号	引数 値の数/ビット数	結果 値の数/ビット数
_m_paddb	_mm_add_pi8	PADDB	加算	--	8/8	8/8
_m_paddw	_mm_add_pi16	PADDW	加算	--	4/16	4/16
_m_paddd	_mm_add_pi32	PADDD	加算	--	2/32	2/32
_m_paddsb	_mm_adds_pi8	PADDSB	加算	符号あり	8/8	8/8
_m_paddsw	_mm_adds_pi16	PADDSW	加算	符号あり	4/16	4/16
_m_paddusb	_mm_adds_pu8	PADDUSB	加算	符号なし	8/8	8/8
_m_paddusw	_mm_adds_pu16	PADDUSW	加算	符号なし	4/16	4/16
_m_psubb	_mm_sub_pi8	PSUBB	減算	--	8/8	8/8
_m_psubw	_mm_sub_pi16	PSUBW	減算	--	4/16	4/16
_m_psubd	_mm_sub_pi32	PSUBD	減算	--	2/32	2/32
_m_psubsb	_mm_subs_pi8	PSUBSB	減算	符号あり	8/8	8/8
_m_psubsw	_mm_subs_pi16	PSUBSW	減算	符号あり	4/16	4/16
_m_psubusb	_mm_subs_pu8	PSUBUSB	減算	符号なし	8/8	8/8
_m_psubusw	_mm_subs_pu16	PSUBUSW	減算	符号なし	4/16	4/16
_m_pmaddwd	_mm_madd_pi16	PMADDWD	乗算	--	4/16	2/32
_m_pmulhw	_mm_mulhi_pi16	PMULHW	乗算	符号あり	4/16	4/16 (上位)
_m_pmullw	_mm_mullo_pi16	PMULLW	乗算	--	4/16	4/16 (下位)

```
__m64 _m_paddb(__m64 m1, __m64 m2)
```

m1 の 8 つの 8 ビット値を、m2 の 8 つの 8 ビット値に加算します。

```
__m64 _m_paddw(__m64 m1, __m64 m2)
```

m1 の 4 つの 16 ビット値を、m2 の 4 つの 16 ビット値に加算します。

`__m64 _m_paddd(__m64 m1, __m64 m2)`

`m1` の 2 つの 32 ビット値を、`m2` の 2 つの 32 ビット値に加算します。

`__m64 _m_paddsb(__m64 m1, __m64 m2)`

飽和演算を使用して、`m1` の 8 つの符号付き 8 ビット値を、`m2` の 8 つの符号付き 8 ビット値に加算します。

`__m64 _m_paddsw(__m64 m1, __m64 m2)`

飽和演算を使用して、`m1` の 4 つの符号付き 16 ビット値を、`m2` の 4 つの符号付き 16 ビット値に加算します。

`__m64 _m_paddusb(__m64 m1, __m64 m2)`

飽和演算を使用して、`m1` の 8 つの符号なし 8 ビット値を、`m2` の 8 つの符号なし 8 ビット値に加算します。

`__m64 _m_paddusw(__m64 m1, __m64 m2)`

飽和演算を使用して、`m1` の 4 つの符号なし 16 ビット値を、`m2` の 4 つの符号なし 16 ビット値に加算します。

`__m64 _m_psubb(__m64 m1, __m64 m2)`

`m1` の 8 つの 8 ビット値から、`m2` の 8 つの 8 ビット値を引きます。

`__m64 _m_psubw(__m64 m1, __m64 m2)`

`m1` の 4 つの 16 ビット値から、`m2` の 4 つの 16 ビット値を引きます。

`__m64 _m_psubd(__m64 m1, __m64 m2)`

`m1` の 2 つの 32 ビット値から、`m2` の 2 つの 32 ビット値を引きます。

`__m64 _m_psubsb(__m64 m1, __m64 m2)`

飽和演算を使用して、`m1` の 8 つの符号付き 8 ビット値から、`m2` の 8 つの符号付き 8 ビット値を引きます。

`__m64 _m_psubsw(__m64 m1, __m64 m2)`

飽和演算を使用して、`m1` の 4 つの符号付き 16 ビット値から、`m2` の 4 つの符号付き 16 ビット値を引きます。

__m64 _m_psubusb(__m64 m1, __m64 m2)

飽和演算を使用して、m1 の 8 つの符号なし 8 ビット値から、m2 の 8 つの符号なし 8 ビット値を引きます。

__m64 _m_psubusw(__m64 m1, __m64 m2)

飽和演算を使用して、m1 の 4 つの符号なし 16 ビット値から、m2 の 4 つの符号なし 8 ビット値を引きます。

__m64 _m_pmaddwd(__m64 m1, __m64 m2)

m1 の 4 つの 16 ビット値に m2 の 4 つの 16 ビット値を掛けて、4 つの 32 ビット即値を求め、それらを 2 つずつ合計して 2 つの 32 ビットの結果を求めます。

__m64 _m_pmulhw(__m64 m1, __m64 m2)

m1 の 4 つの符号付き 16 ビット値に m2 の 4 つの符号付き 16 ビット値を掛けて、4 つの結果の上位 16 ビットを求めます。

__m64 _m_pmullw(__m64 m1, __m64 m2)

m1 の 4 つの 16 ビット値に m2 の 4 つの 16 ビット値を掛けて、4 つの結果の下位 16 ビットを求めます。

MMX® テクノロジのシフト組込み関数

MMX® テクノロジの組込み関数のプロトタイプは、ヘッダファイル mmintrin.h 内にあります。

組込み関数の名前	別名	シフトの方向	シフトの種類	対応する命令
_m_psllw	_mm_sll_pi16	左	論理	PSLLW
_m_psllwi	_mm_slli_pi16	左	論理	PSLLWI
_m_pslld	_mm_sll_pi32	左	論理	PSLLD
_m_pslldi	_mm_slli_pi32	左	論理	PSLLDI
_m_psllq	_mm_sll_si64	左	論理	PSLLQ
_m_psllqi	_mm_slli_si64	左	論理	PSLLQI
_m_psraw	_mm_sra_pi16	右	算術	PSRAW
_m_psrawi	_mm_srai_pi16	右	算術	PSRAWI

_m_psrad	_mm_sra_pi32	右	算術	PSRAD
_m_psradi	_mm_srai_pi32	右	算術	PSRADI
_m_psrlw	_mm_srl_pi16	右	論理	PSRLW
_m_psrlwi	_mm_srli_pi16	右	論理	PSRLWI
_m_psrld	_mm_srl_pi32	右	論理	PSRLD
_m_psrldi	_mm_srli_pi32	右	論理	PSRLDI
_m_psrlq	_mm_srl_si64	右	論理	PSRLQ
_m_psrlqi	_mm_srli_si64	右	論理	PSRLQI

```
__m64 _m_psllw(__m64 m, __m64 count)
```

m の 4 つの 16 ビット値を、count で指定した値だけ左にシフトし、下位ビットを 0 で埋めます。

```
__m64 _m_psllwi(__m64 m, int count)
```

m の 4 つの 16 ビット値を、count で指定した値だけ左にシフトし、下位ビットを 0 で埋めます。パフォーマンス上の理由で、count は定数にしてください。

```
__m64 _m_pslld(__m64 m, __m64 count)
```

m の 2 つの 32 ビット値を、count で指定した値だけ左にシフトし、下位ビットを 0 で埋めます。

```
__m64 _m_pslldi(__m64 m, int count)
```

m の 2 つの 32 ビット値を、count で指定した値だけ左にシフトし、下位ビットを 0 で埋めます。パフォーマンス上の理由で、count は定数にしてください。

```
__m64 _m_psllq(__m64 m, __m64 count)
```

m の 64 ビット値を、count で指定した値だけ左にシフトし、下位ビットを 0 で埋めます。

```
__m64 _m_psllqi(__m64 m, int count)
```

m の 64 ビット値を、count で指定した値だけ左にシフトし、下位ビットを 0 で埋めます。パフォーマンス上の理由で、count は定数にしてください。

```
__m64 _m_psraw(__m64 m, __m64 count)
```

m の 4 つの 16 ビット値を、count で指定した値だけ右にシフトし、上位ビットを符号ビットで埋めます。

```
__m64 _m_psrawi(__m64 m, int count)
```

m の 4 つの 16 ビット値を、count で指定した値だけ右にシフトし、上位ビットを符号ビットで埋めます。パフォーマンス上の理由で、count は定数にしてください。

```
__m64 _m_psrad(__m64 m, __m64 count)
```

m の 2 つの 32 ビット値を、count で指定した値だけ右にシフトし、上位ビットを符号ビットで埋めます。

```
__m64 _m_psradi(__m64 m, int count)
```

m の 2 つの 32 ビット値を、count で指定した値だけ右にシフトし、上位ビットを符号ビットで埋めます。パフォーマンス上の理由で、count は定数にしてください。

```
__m64 _m_psrlw(__m64 m, __m64 count)
```

m の 4 つの 16 ビット値を、count で指定した値だけ右にシフトし、上位ビットを 0 で埋めます。

```
__m64 _m_psrlwi(__m64 m, int count)
```

m の 4 つの 16 ビット値を、count で指定した値だけ右にシフトし、上位ビットを 0 で埋めます。パフォーマンス上の理由で、count は定数にしてください。

```
__m64 _m_psrld(__m64 m, __m64 count)
```

m の 2 つの 32 ビット値を、count で指定した値だけ右にシフトし、上位ビットを 0 で埋めます。

```
__m64 _m_psrldi(__m64 m, int count)
```

m の 2 つの 32 ビット値を、count で指定した値だけ右にシフトし、上位ビットを 0 で埋めます。パフォーマンス上の理由で、count は定数にしてください。

```
__m64 _m_psrlq(__m64 m, __m64 count)
```

m の 64 ビット値を、count で指定した値だけ右にシフトし、上位ビットを 0 で埋めます。

```
__m64 _m_psrlqi(__m64 m, int count)
```

m の 64 ビット値を、count で指定した値だけ右にシフトし、上位ビットを 0 で埋めます。パフォーマンス上の理由で、count は定数にしてください。

MMX® テクノロジの論理演算組込み関数

MMX® テクノロジの組込み関数のプロトタイプは、ヘッダファイル mmintrin.h 内にあります。

組込み関数の名前	別名	操作	対応する命令
_m_pand	_mm_and_si64	ビット単位の AND (論理積)	PAND
_m_pandn	_mm_andnot_si64	NOT (否定)	PANDN
_m_por	_mm_or_si64	ビット単位の OR (論理和)	POR
_m_pxor	_mm_xor_si64	ビット単位の XOR (排他的論理和)	PXOR

```
__m64 _m_pand(__m64 m1, __m64 m2)
```

m1 の 64 ビット値と m2 の 64 ビット値について、ビット単位の AND (論理積) 演算を実行します。

```
__m64 _m_pandn(__m64 m1, __m64 m2)
```

m1 の 64 ビット値の NOT (否定) 演算を実行し、その結果と m2 の 64 ビット値について、ビット単位の AND (論理積) 演算を実行します。

```
__m64 _m_por(__m64 m1, __m64 m2)
```

m1 の 64 ビット値と m2 の 64 ビット値について、ビット単位の OR (論理和) 演算を実行します。

```
__m64 _m_pxor(__m64 m1, __m64 m2)
```

m1 の 64 ビット値と m2 の 64 ビット値について、ビット単位の XOR (排他的論理和) 演算を実行します。

MMX® テクノロジの比較組込み関数

MMX® テクノロジの組込み関数のプロトタイプは、ヘッダファイル mmintrin.h 内にあります。

組込み関数の名前	別名	比較条件	要素の数	要素のサイズ (ビット)	対応する命令
_m_pcmpeqb	_mm_cmpeq_pi8	等しい	8	8	PCMPEQB
_m_pcmpeqw	_mm_cmpeq_pi16	等しい	4	16	PCMPEQW
_m_pcmpeqd	_mm_cmpeq_pi32	等しい	2	32	PCMPEQD
_m_pcmpgtb	_mm_cmpgt_pi8	より大きい	8	8	PCMPGTB
_m_pcmpgtw	_mm_cmpgt_pi16	より大きい	4	16	PCMPGTW
_m_pcmpgtd	_mm_cmpgt_pi32	より大きい	2	32	PCMPGTD

```
__m64 _m_pcmpeqb(__m64 m1, __m64 m2)
```

m1 の各 8 ビット値が、それに対応する m2 の 8 ビット値に等しい場合は、それに対応する結果の 8 ビット値をすべて 1 に設定します。それ以外の場合は、すべて 0 に設定します。

```
__m64 _m_pcmpeqw(__m64 m1, __m64 m2)
```

m1 の各 16 ビット値が、それに対応する m2 の 16 ビット値に等しい場合は、それに対応する結果の 16 ビット値をすべて 1 に設定します。それ以外の場合は、すべて 0 に設定します。

```
__m64 _m_pcmpeqd(__m64 m1, __m64 m2)
```

m1 の各 32 ビット値が、それに対応する m2 の 32 ビット値に等しい場合は、それに対応する結果の 32 ビット値をすべて 1 に設定します。それ以外の場合は、すべて 0 に設定します。

```
__m64 _m_pcmpgtb(__m64 m1, __m64 m2)
```

m1 の各 8 ビット値が、それに対応する m2 の 8 ビット値より大きい場合は、それに対応する結果の 8 ビット値をすべて 1 に設定します。それ以外の場合は、すべて 0 に設定します。

```
__m64 _m_pcmpgtw(__m64 m1, __m64 m2)
```

m1 の各 16 ビット値が、それに対応する m2 の 16 ビット値より大きい場合は、それに対応する結果の 16 ビット値をすべて 1 に設定します。それ以外の場合は、すべて 0 に設定します。

```
__m64 _m_pcmpgtd(__m64 m1, __m64 m2)
```

m1 の各 32 ビット値が、それに対応する m2 の 32 ビット値より大きい場合は、それに対応する結果の 32 ビット値をすべて 1 に設定します。それ以外の場合は、すべて 0 に設定します。

MMX® テクノロジの設定組込み関数

MMX® テクノロジの組込み関数のプロトタイプは、ヘッダファイル mmintrin.h 内にあります。

組込み関数の名前	操作	要素の数	要素のサイズ (ビット)	符号	逆順
_mm_setzero_si64	0 に設定	1	64	符号なし	符号なし
_mm_set_pi32	整数値の設定	2	32	符号なし	符号なし
_mm_set_pi16	整数値の設定	4	16	符号なし	符号なし
_mm_set_pi8	整数値の設定	8	8	符号なし	符号なし
_mm_set1_pi32	整数値の設定	2	32	符号あり	符号なし
_mm_set1_pi16	整数値の設定	4	16	符号あり	符号なし
_mm_set1_pi8	整数値の設定	8	8	符号あり	符号なし
_mm_setr_pi32	整数値の設定	2	32	符号なし	符号あり
_mm_setr_pi16	整数値の設定	4	16	符号なし	符号あり
_mm_setr_pi8	整数値の設定	8	8	符号なし	符号あり

注

次の説明では、MMX テクノロジ・レジスタのビット 0 が最下位ビット、ビット 63 が最上位ビットになります。

```
__m64 _mm_setzero_si64()
```

PXOR
64 ビット値を 0 に設定します。
r := 0x0

```
__m64 _mm_set_pi32(int i1, int i0)
```

(複合) 2 つの符号付き 32 ビット整数値を設定します。
```
r0 := i0
r1 := i1
```

```
__m64 _mm_set_pi16(short s3, short s2, short s1, short s0)
```

(複合) 4 つの符号付き 16 ビット整数値を設定します。
```
r0 := w0
r1 := w1
r2 := w2
r3 := w3
```

```
__m64 _mm_set_pi8(char b7, char b6, char b5, char b4, char
b3, char b2, char b1, char b0)
```

(複合) 8 つの符号付き 8 ビット整数値を設定します。
```
r0 := b0
r1 := b1
...
r7 := b7
```

```
__m64 _mm_set1_pi32(int i)
```

2 つの符号付き 32 ビット整数値を `i` に設定します。
```
r0 := i
r1 := i
```

```
__m64 _mm_set1_pi16(short s)
```

(複合) 4 つの符号付き 16 ビット整数値を `w` に設定します。
```
r0 := w
r1 := w
r2 := w
r3 := w
```

```
__m64 _mm_set1_pi8(char b)
```

(複合) 8 つの符号付き 8 ビット整数値を `b` に設定します。
```
r0 := b
r1 := b
...
r7 := b
```

```
__m64 _mm_setr_pi32(int i1, int i0)
```

(複合) 2 つの符号付き 32 ビット整数値を逆順で設定します。
```
r0 := i0
r1 := i1
```

```
__m64 _mm_setr_pi16(short s3, short s2, short s1, short s0)
```

(複合) 4 つの符号付き 16 ビット整数値を逆順で設定します。

```
r0 := w0
r1 := w1
r2 := w2
r3 := w3
```

```
__m64 _mm_setr_pi8(char b7, char b6, char b5, char b4, char
b3, char b2, char b1, char b0)
```

(複合) 8 つの符号付き 8 ビット整数値を逆順で設定します。

```
r0 := b0
r1 := b1
...
r7 := b7
```

Itanium® アーキテクチャ上での MMX® テクノロジ組込み関数の使用

MMX® テクノロジの組込み関数を使用して、Itanium® ベースのシステム上で MMX テクノロジ命令セットを利用できます。IA-32 アーキテクチャのソースコードとの互換性を保つために、これらの組込み関数の名前と機能は、IA-32 ベースの MMX テクノロジ組込み関数セットと同等になっています。

一部の組込み関数には、2 つ以上の名前があります。1 つの組込み関数に 2 つの名前がある場合は、いずれの名前も同じ命令を生成しますが、最初の名前の方が新しい命名規則に適合しているため、最初の名前を使用することをお勧めします。

MMX テクノロジの組込み関数のプロトタイプは、ヘッダファイル `mmintrin.h` 内にあります。

データ型

MMX テクノロジの組込み関数は、C データ型 `__m64` を使用します。このデータ型は、8 つの 8 ビット値、4 つの 16 ビット値、2 つの 32 ビット値、または 1 つの 64 ビット値を保持できます。

`__m64` データ型は、基本的な ANSI C データ型ではありません。このため、次のような使用上の制限があります:

- この新しいデータ型は、代入文の左辺で、戻り値またはパラメータとして使用してください。他の算術式 (″+″、″−″など) にこのデータ型を使用することはできません。

- この新しいデータ型は、バイト要素/構造にアクセスするための共用体などの集合体のオブジェクトとして使用してください。`__m64` オブジェクトのアドレスを指定できます。
- 新しいデータ型は、本書で説明する組込み関数でのみ使用してください。

ハードウェア命令の詳細については、『Intel® Architecture MMX Technology Programmer's Reference Manual』(英語) を参照してください。データ型の詳細については、『インテル® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、中巻』を参照してください。

ストリーミング SIMD 拡張命令

概要: ストリーミング SIMD 拡張命令

このセクションでは、インテル® C++ コンパイラ内でストリーミング SIMD 拡張命令をサポートする、C++ 言語レベルの機能について説明します。ここでは、ストリーミング SIMD 拡張命令の組込み関数の次のような機能について説明します:

- 浮動小数点演算組込み関数
- 算術演算組込み関数
- 論理演算組込み関数
- 比較組込み関数
- 変換組込み関数
- ロード操作
- 設定操作
- ストア操作
- キャッシュ制御
- 整数演算組込み関数
- メモリ操作と初期化操作の組込み関数
- その他の組込み関数
- Itanium® アーキテクチャ上でのストリーミング SIMD 拡張命令の使用

ストリーミング SIMD 拡張命令の組込み関数のプロトタイプは、ヘッダファイル `xmmintrin.h` 内にあります。

ストリーミング SIMD 拡張命令を使用する浮動小数点演算組込み関数

ストリーミング SIMD 拡張命令の組込み関数を使用してプログラムを作成する際は、ストリーミング SIMD 拡張命令によって提供されるハードウェア機能をよく理解している必要があります。特に、次の 4 つの重要な点に注意してください:

- _mm_loadr_pd や _mm_cmpgt_sd などの一部の組込み関数は、命令セットによって直接にはサポートされていません。これらの組込み関数は、プログラミング上の便宜のために用意したもので、実際には 2 つ以上のマシン言語命令で構成されています。
- __m128 オブジェクトとしてロードまたはストアされる浮動小数点データは、通常は 16 バイトにアライメントが合っていなければなりません。
- 一部の組込み関数は、命令の性質上、引数を即値で、すなわち定数整数 (リテラル) で指定する必要があります。
- 2 つの NaN (Not a Number) 引数を操作する算術演算の結果は未定義です。したがって、NaN 引数を使用する FP (浮動小数点) 演算は、対応するアセンブリ言語命令の予想される動作とは一致しません。

ストリーミング SIMD 拡張命令の算術演算

ストリーミング SIMD 拡張命令の組込み関数のプロトタイプは、ヘッダファイル xmmintrin.h 内にあります。

組込み関数	命令	操作	R0	R1	R2	R3
_mm_add_ss	ADDSS	加算	a0 [op] b0	a1	a2	a3
_mm_add_ps	ADDPS	加算	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
_mm_sub_ss	SUBSS	減算	a0 [op] b0	a1	a2	a3
_mm_sub_ps	SUBPS	減算	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
_mm_mul_ss	MULSS	乗算	a0 [op] b0	a1	a2	a3
_mm_mul_ps	MULPS	乗算	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
_mm_div_ss	DIVSS	除算	a0 [op] b0	a1	a2	a3
_mm_div_ps	DIVPS	除算	a0 [op] b0	a1 [op] b1	a2 [op] b2	a3 [op] b3
_mm_sqrt_ss	SQRTSS	平方根	[op] a0	a1	a2	a3
_mm_sqrt_ps	SQRTPS	平方根	[op] a0	[op] b1	[op] b2	[op] b3
_mm_rcp_ss	RCPSS	逆数	[op] a0	a1	a2	a3
_mm_rcp_ps	RCPPS	逆数	[op] a0	[op] b1	[op] b2	[op] b3
_mm_rsqrt_ss	RSQRTSS	平方根の逆数	[op] a0	a1	a2	a3
_mm_rsqrt_ps	RSQRTPS	平方根の逆数	[op] a0	[op] b1	[op] b2	[op] b3
_mm_min_ss	MINSS	最小値	[op](a0,b0)	a1	a2	a3

		の計算				
_mm_min_ps	MINPS	最小値の計算	[op](a0,b0)	[op] (a1, b1)	[op] (a2, b2)	[op] (a3, b3)
_mm_max_ss	MAXSS	最大値の計算	[op](a0,b0)	a1	a2	a3
_mm_max_ps	MAXPS	最大値の計算	[op](a0,b0)	[op] (a1, b1)	[op] (a2, b2)	[op] (a3, b3)

```
__m128 _mm_add_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値を加算します。上位 3 つの単精度浮動小数点値は、a からそのまま渡されます。

```
r0 := a0 + b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_add_ps(__m128 a, __m128 b)
```

a と b の 4 つの単精度浮動小数点値を加算します。

```
r0 := a0 + b0
r1 := a1 + b1
r2 := a2 + b2
r3 := a3 + b3
```

```
__m128 _mm_sub_ss(__m128 a, __m128 b)
```

a の最下位の単精度浮動小数点値から、b の最下位の単精度浮動小数点値を引きます。上位 3 つの単精度浮動小数点値は、a からそのまま渡されます。

```
r0 := a0 - b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_sub_ps(__m128 a, __m128 b)
```

a の 4 つの単精度浮動小数点値から、b の 4 つの単精度浮動小数点値を引きます。

```
r0 := a0 - b0
r1 := a1 - b1
r2 := a2 - b2
r3 := a3 - b3
```

```
__m128 _mm_mul_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値を乗算します。上位 3 つの単精度浮動小数点値は、a からそのまま渡されます。

```
r0 := a0 * b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_mul_ps(__m128 a, __m128 b)
```

a と b の 4 つの単精度浮動小数点値を乗算します。

```
r0 := a0 * b0
r1 := a1 * b1
r2 := a2 * b2
r3 := a3 * b3
```

```
__m128 _mm_div_ss(__m128 a, __m128 b )
```

a の最下位の単精度浮動小数点値を、b の最下位の単精度浮動小数点値で割ります。上位 3 つの単精度浮動小数点値は、a からそのまま渡されます。

```
r0 := a0 / b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_div_ps(__m128 a, __m128 b)
```

a の 4 つの単精度浮動小数点値を、b の 4 つの単精度浮動小数点値で割ります。

```
r0 := a0 / b0
r1 := a1 / b1
r2 := a2 / b2
r3 := a3 / b3
```

```
__m128 _mm_sqrt_ss(__m128 a)
```

a の最下位の単精度浮動小数点値の平方根を計算します。上位 3 つの単精度浮動小数点値はそのまま渡されます。

```
r0 := sqrt(a0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_sqrt_ps(__m128 a)
```

a の 4 つの単精度浮動小数点値の平方根を計算します。

```
r0 := sqrt(a0)
r1 := sqrt(a1)
r2 := sqrt(a2)
r3 := sqrt(a3)
```

```
__m128 _mm_rcp_ss(__m128 a)
```

a の最下位の単精度浮動小数点値の逆数の近似値を計算します。上位 3 つの単精度浮動小数点値はそのまま渡されます。

```
r0 := recip(a0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_rcp_ps(__m128 a)
```

a の 4 つの単精度浮動小数点値の逆数の近似値を計算します。

```
r0 := recip(a0)
r1 := recip(a1)
r2 := recip(a2)
r3 := recip(a3)
```

```
__m128 _mm_rsqrt_ss(__m128 a)
```

a の最下位の単精度浮動小数点値の平方根の逆数の近似値を計算します。上位 3 つの単精度浮動小数点値はそのまま渡されます。

```
r0 := recip(sqrt(a0))
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_rsqrt_ps(__m128 a)
```

a の 4 つの単精度浮動小数点値の平方根の逆数の近似値を計算します。

```
r0 := recip(sqrt(a0))
r1 := recip(sqrt(a1))
r2 := recip(sqrt(a2))
r3 := recip(sqrt(a3))
```

```
__m128 _mm_min_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、小さい方の値を計算をします。上位 3 つの単精度浮動小数点値は、a からそのまま渡されます。

```
r0 := min(a0, b0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_min_ps(__m128 a, __m128 b)
```

a と b の 4 つの単精度浮動小数点値について、それぞれ小さい方の値を計算します。

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

```
__m128 _mm_max_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、大きい方の値を計算をします。上位 3 つの単精度浮動小数点値は、a からそのまま渡されます。

```
r0 := max(a0, b0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_max_ps(__m128 a, __m128 b)
```

a と b の 4 つの単精度浮動小数点値について、大きい方の値を計算します。

```
r0 := max(a0, b0)
r1 := max(a1, b1)
r2 := max(a2, b2)
r3 := max(a3, b3)
```

ストリーミング SIMD 拡張命令の論理演算

ストリーミング SIMD 拡張命令の組込み関数のプロトタイプは、ヘッダファイル xmmintrin.h 内にあります。

組込み関数の名前	操作	対応する命令
_mm_and_ps	ビット単位の AND (論理積)	ANDPS
_mm_andnot_ps	NOT (否定)	ANDNPS
_mm_or_ps	ビット単位の OR (論理和)	ORPS
_mm_xor_ps	ビット単位の XOR (排他的論理和)	XORPS

```
__m128 _mm_and_ps(__m128 a, __m128 b)
```

a の 4 つの単精度浮動小数点値と b の 4 つの単精度浮動小数点値について、ビット単位の AND (論理積) を計算します。

```
r0 := a0 & b0
r1 := a1 & b1
r2 := a2 & b2
r3 := a3 & b3
```

```
__m128 _mm_andnot_ps(__m128 a, __m128 b)
```

a の 4 つの単精度浮動小数点値の NOT (否定) 演算を実行し、その結果と b の 4 つの単精度浮動小数点値について、ビット単位の AND (論理積) を計算します。

```
r0 := ~a0 & b0
r1 := ~a1 & b1
r2 := ~a2 & b2
r3 := ~a3 & b3
```

```
__m128 _mm_or_ps(__m128 a, __m128 b)
```

a の 4 つの単精度浮動小数点値と b の 4 つの単精度浮動小数点値について、ビット単位の OR (論理和) を計算します。

```
r0 := a0 | b0
r1 := a1 | b1
```

```
      r2 := a2 | b2
      r3 := a3 | b3

__m128 _mm_xor_ps(__m128 a, __m128 b)
```

a の 4 つの単精度浮動小数点値と b の 4 つの単精度浮動小数点値について、ビット単位の XOR (排他的論理和) を計算します。

```
r0 := a0 ^ b0
r1 := a1 ^ b1
r2 := a2 ^ b2
r3 := a3 ^ b3
```

ストリーミング SIMD 拡張命令の比較操作

比較組込み関数は、a と b の比較を実行します。パックド形式では、a と b の 4 つの単精度浮動小数点値を比較して、128 ビットマスクを返します。スカラ形式では、a と b の最下位の単精度浮動小数点値を比較して、32 ビットマスクを返します。上位 3 つの単精度浮動小数点値は、a からそのまま渡されます。マスクは、各要素について、比較の結果が真の場合は 0xffffffff に設定し、偽の場合は 0x0 に設定されます。

ストリーミング SIMD 拡張命令の組込み関数のプロトタイプは、ヘッダファイル xmmintrin.h 内にあります。

組込み関数の名前	比較条件	対応する命令
_mm_cmpeq_ss	等しい	CMPEQSS
_mm_cmpeq_ps	等しい	CMPEQPS
_mm_cmplt_ss	より小さい	CMPLTSS
_mm_cmplt_ps	より小さい	CMPLTPS
_mm_cmple_ss	以下	CMPLESS
_mm_cmple_ps	以下	CMPLEPS
_mm_cmpgt_ss	より大きい	CMPLTSS
_mm_cmpgt_ps	より大きい	CMPLTPS
_mm_cmpge_ss	以上	CMPLESS
_mm_cmpge_ps	以上	CMPLEPS
_mm_cmpneq_ss	等しくない	CMPNEQSS
_mm_cmpneq_ps	等しくない	CMPNEQPS
_mm_cmpnlt_ss	より小さくない	CMPNLTSS

_mm_cmpnlt_ps	より小さくない	CMPNLTPS
_mm_cmpnle_ss	以下でない	CMPNLESS
_mm_cmpnle_ps	以下でない	CMPNLEPS
_mm_cmpngt_ss	より大きくない	CMPNLTSS
_mm_cmpngt_ps	より大きくない	CMPNLTPS
_mm_cmpnge_ss	以上でない	CMPNLESS
_mm_cmpnge_ps	以上でない	CMPNLEPS
_mm_cmpord_ss	順序化可能	CMPORDSS
_mm_cmpord_ps	順序化可能	CMPORDPS
_mm_cmpunord_ss	順序化不可能	CMPUNORDSS
_mm_cmpunord_ps	順序化不可能	CMPUNORDPS
_mm_comieq_ss	等しい	COMISS
_mm_comilt_ps	より小さい	COMISS
_mm_comile_ss	以下	COMISS
_mm_comigt_ss	より大きい	COMISS
_mm_comige_ss	以上	COMISS
_mm_comineq_ss	等しくない	COMISS
_mm_ucomieq_ss	等しい	UCOMISS
_mm_ucomilt_ss	より小さい	UCOMISS
_mm_ucomile_ss	以下	UCOMISS
_mm_ucomigt_ss	より大きい	UCOMISS
_mm_ucomige_ss	以上	UCOMISS
_mm_ucomineq_ss	等しくない	UCOMISS

```
__m128 _mm_cmpeq_ss(__m128 a, __m128 b)
```

等しいかどうか比較します。

```
r0 := (a0 == b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpeq_ps(__m128 a, __m128 b)
```

等しいかどうか比較します。

```
r0 := (a0 == b0) ? 0xffffffff : 0x0
r1 := (a1 == b1) ? 0xffffffff : 0x0
```

```
      r2 := (a2 == b2) ? 0xffffffff : 0x0
      r3 := (a3 == b3) ? 0xffffffff : 0x0

__m128 _mm_cmplt_ss(__m128 a, __m128 b)
```

より小さいどうか比較します。

```
      r0 := (a0 < b0) ? 0xffffffff : 0x0
      r1 := a1 ; r2 := a2 ; r3 := a3

__m128 _mm_cmplt_ps(__m128 a, __m128 b)
```

より小さいどうか比較します。

```
      r0 := (a0 < b0) ? 0xffffffff : 0x0
      r1 := (a1 < b1) ? 0xffffffff : 0x0
      r2 := (a2 < b2) ? 0xffffffff : 0x0
      r3 := (a3 < b3) ? 0xffffffff : 0x0

__m128 _mm_cmple_ss(__m128 a, __m128 b)
```

以下かどうか比較します。

```
      r0 := (a0 <= b0) ? 0xffffffff : 0x0
      r1 := a1 ; r2 := a2 ; r3 := a3

__m128 _mm_cmple_ps(__m128 a, __m128 b)
```

以下かどうか比較します。

```
      r0 := (a0 <= b0) ? 0xffffffff : 0x0
      r1 := (a1 <= b1) ? 0xffffffff : 0x0
      r2 := (a2 <= b2) ? 0xffffffff : 0x0
      r3 := (a3 <= b3) ? 0xffffffff : 0x0

__m128 _mm_cmpgt_ss(__m128 a, __m128 b)
```

より大きいかどうか比較します。

```
      r0 := (a0 > b0) ? 0xffffffff : 0x0
      r1 := a1 ; r2 := a2 ; r3 := a3

__m128 _mm_cmpgt_ps(__m128 a, __m128 b)
```

より大きいかどうか比較します。

```
      r0 := (a0 > b0) ? 0xffffffff : 0x0
      r1 := (a1 > b1) ? 0xffffffff : 0x0
      r2 := (a2 > b2) ? 0xffffffff : 0x0
      r3 := (a3 > b3) ? 0xffffffff : 0x0

__m128 _mm_cmpge_ss(__m128 a, __m128 b)
```

以上かどうか比較します。

```
      r0 := (a0 >= b0) ? 0xffffffff : 0x0
      r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpge_ps(__m128 a, __m128 b)
```

以上かどうか比較します。

```
r0 := (a0 >= b0) ? 0xffffffff : 0x0
r1 := (a1 >= b1) ? 0xffffffff : 0x0
r2 := (a2 >= b2) ? 0xffffffff : 0x0
r3 := (a3 >= b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpneq_ss(__m128 a, __m128 b)
```

等しくないかどうか比較します。

```
r0 := (a0 != b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpneq_ps(__m128 a, __m128 b)
```

等しくないかどうか比較します。

```
r0 := (a0 != b0) ? 0xffffffff : 0x0
r1 := (a1 != b1) ? 0xffffffff : 0x0
r2 := (a2 != b2) ? 0xffffffff : 0x0
r3 := (a3 != b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)
```

より小さくないかどうか比較します。

```
r0 := !(a0 < b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)
```

より小さくないかどうか比較します。

```
r0 := !(a0 < b0) ? 0xffffffff : 0x0
r1 := !(a1 < b1) ? 0xffffffff : 0x0
r2 := !(a2 < b2) ? 0xffffffff : 0x0
r3 := !(a3 < b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpnle_ss(__m128 a, __m128 b)
```

以下でないかどうか比較します。

```
r0 := !(a0 <= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpnle_ps(__m128 a, __m128 b)
```

以下でないかどうか比較します。

```
r0 := !(a0 <= b0) ? 0xffffffff : 0x0
r1 := !(a1 <= b1) ? 0xffffffff : 0x0
r2 := !(a2 <= b2) ? 0xffffffff : 0x0
r3 := !(a3 <= b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpngt_ss(__m128 a, __m128 b)
```

より大きくないかどうか比較します。

```
r0 := !(a0 > b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpngt_ps(__m128 a, __m128 b)
```

より大きくないかどうか比較します。

```
r0 := !(a0 > b0) ? 0xffffffff : 0x0
r1 := !(a1 > b1) ? 0xffffffff : 0x0
r2 := !(a2 > b2) ? 0xffffffff : 0x0
r3 := !(a3 > b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpnge_ss(__m128 a, __m128 b)
```

以上でないかどうか比較します。

```
r0 := !(a0 >= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpnge_ps(__m128 a, __m128 b)
```

以上でないかどうか比較します。

```
r0 := !(a0 >= b0) ? 0xffffffff : 0x0
r1 := !(a1 >= b1) ? 0xffffffff : 0x0
r2 := !(a2 >= b2) ? 0xffffffff : 0x0
r3 := !(a3 >= b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpord_ss(__m128 a, __m128 b)
```

順序化可能かどうか判定します。

```
r0 := (a0 ord? b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpord_ps(__m128 a, __m128 b)
```

順序化可能かどうか判定します。

```
r0 := (a0 ord? b0) ? 0xffffffff : 0x0
r1 := (a1 ord? b1) ? 0xffffffff : 0x0
r2 := (a2 ord? b2) ? 0xffffffff : 0x0
r3 := (a3 ord? b3) ? 0xffffffff : 0x0
```

```
__m128 _mm_cmpunord_ss(__m128 a, __m128 b)
```

順序化不可能かどうか判定します。

```
r0 := (a0 unord? b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cmpunord_ps(__m128 a, __m128 b)
```

順序化不可能かどうか判定します。

```
r0 := (a0 unord? b0) ? 0xffffffff : 0x0
r1 := (a1 unord? b1) ? 0xffffffff : 0x0
r2 := (a2 unord? b2) ? 0xffffffff : 0x0
r3 := (a3 unord? b3) ? 0xffffffff : 0x0
```

```
int _mm_comieq_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、a と b が等しいかどうか比較します。a と b が等しい場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 == b0) ? 0x1 : 0x0
```

```
int _mm_comilt_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、a が b より小さいかどうか比較します。a が b より小さい場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 < b0) ? 0x1 : 0x0
```

```
int _mm_comile_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、a が b 以下かどうか比較します。a が b 以下の場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 <= b0) ? 0x1 : 0x0
```

```
int _mm_comigt_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、a が b より大きいかどうか比較します。a が b より大きい場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 > b0) ? 0x1 : 0x0
```

```
int _mm_comige_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、a が b 以上かどうか比較します。a が b 以上の場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 >= b0) ? 0x1 : 0x0
```

```
int _mm_comineq_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、a と b が等しくないかどうか比較します。a と b が等しくない場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 != b0) ? 0x1 : 0x0
```

```
int _mm_ucomieq_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、a と b が等しいかどうか比較します。a と b が等しい場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 == b0) ? 0x1 : 0x0
```

```
int _mm_ucomilt_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、a が b より小さいかどうか比較します。a が b より小さい場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 < b0) ? 0x1 : 0x0
```

```
int _mm_ucomile_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、a が b 以下かどうか比較します。a が b 以下の場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 <= b0) ? 0x1 : 0x0
```

```
int _mm_ucomigt_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、a が b より大きいかどうか比較します。a が b 以上の場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 > b0) ? 0x1 : 0x0
```

```
int _mm_ucomige_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、a が b 以上かどうか比較します。a が b 以上の場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 >= b0) ? 0x1 : 0x0
```

```
int _mm_ucomineq_ss(__m128 a, __m128 b)
```

a と b の最下位の単精度浮動小数点値について、a と b が等しくないかどうか比較します。a と b が等しくない場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 != b0) ? 0x1 : 0x0
```

ストリーミング SIMD 拡張命令の変換操作

次の表に、変換操作のリストを示します。表の後に、最新のニーモニック命名規則に従って、各組込み関数の説明を示します。これらの組込み関数を以前に使用していた場合に備えて、別名を用意しています。

ストリーミング SIMD 拡張命令の組込み関数のプロトタイプは、ヘッダファイル xmmintrin.h 内にあります。

組込み関数の名前	別名	対応する命令
_mm_cvt_ss2si	_mm_cvtss_si32	CVTSS2SI
_mm_cvt_ps2pi	_mm_cvtps_pi32	CVTPS2PI
_mm_cvtt_ss2si	_mm_cvttss_si32	CVTTSS2SI
_mm_cvtt_ps2pi	_mm_cvttps_pi32	CVTTPS2PI
_mm_cvt_si2ss	_mm_cvtsi32_ss	CVTSI2SS
_mm_cvt_pi2ps	_mm_cvtpi32_ps	CVTTPS2PI
_mm_cvtpi16_ps		複合
_mm_cvtpu16_ps		複合
_mm_cvtpi8_ps		複合
_mm_cvtpu8_ps		複合
_mm_cvtpi32x2_ps		複合
_mm_cvtps_pi16		複合
_mm_cvtps_pi8		複合

```
int _mm_cvt_ss2si(__m128 a)
```

現在の丸めモードに従って、a の最下位の単精度浮動小数点値を 32 ビット整数に変換します。

```
r := (int)a0
```

```
__m64 _mm_cvt_ps2pi(__m128 a)
```

現在の丸めモードに従って、a の下位 2 つの単精度浮動小数点値を 2 つの 32 ビット整数に変換し、パックド形式で返します。

```
r0 := (int)a0
r1 := (int)a1
```

```
int _mm_cvtt_ss2si(__m128 a)
```

切り捨てを使用して、a の最下位の単精度浮動小数点値を 32 ビット整数に変換します。

```
r := (int)a0
```

```
__m64 _mm_cvtt_ps2pi(__m128 a)
```

切り捨てを使用して、a の下位 2 つの単精度浮動小数点値を 2 つの 32 ビット整数に変換し、パックド形式で返します。

```
r0 := (int)a0
r1 := (int)a1
```

```
__m128 _mm_cvt_si2ss(__m128, int)
```

32 ビット整数 b を単精度浮動小数点値に変換します。上位 3 つの単精度浮動小数点値は、a からそのまま渡されます。

```
r0 := (float)b
r1 := a1 ; r2 := a2 ; r3 := a3
```

```
__m128 _mm_cvt_pi2ps(__m128, __m64)
```

b のパックド形式の 2 つの 32 ビット整数値を、2 つの単精度浮動小数点値に変換します。上位 2 つの単精度浮動小数点値は、a からそのまま渡されます。

```
r0 := (float)b0
r1 := (float)b1
r2 := a2
r3 := a3
```

```
__inline __m128 _mm_cvtpi16_ps(__m64 a)
```

a の 4 つの符号付き 16 ビット整数値を、4 つの単精度浮動小数点値に変換します。

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

```
__inline __m128 _mm_cvtpu16_ps(__m64 a)
```

a の 4 つの符号なし 16 ビット整数値を、4 つの単精度浮動小数点値に変換します。

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

```
__inline __m128 _mm_cvtpi8_ps(__m64 a)
```

a の下位 4 つの符号付き 8 ビット整数値を、4 つの単精度浮動小数点値に変換します。

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

```
__inline __m128 _mm_cvtpu8_ps(__m64 a)
```

a の下位 4 つの符号なし 8 ビット整数値を、4 つの単精度浮動小数点値に変換します。

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

```
__inline __m128 _mm_cvtpi32x2_ps(__m64 a, __m64 b)
```

a の 2 つの符号付き 32 ビット整数値と b の 2 つの符号付き 32 ビット整数値を、4 つの単精度浮動小数点値に変換します。

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)b0
r3 := (float)b1
```

```
__inline __m64 _mm_cvtps_pi16(__m128 a)
```

a の 4 つの単精度浮動小数点値を、4 つの符号付き 16 ビット整数値に変換します。

```
r0 := (short)a0
r1 := (short)a1
r2 := (short)a2
r3 := (short)a3
```

```
__inline __m64 _mm_cvtps_pi8(__m128 a)
```

a の 4 つの単精度浮動小数点値を、結果の下位 4 つの符号付き 8 ビット整数値に変換します。

```
r0 := (char)a0
r1 := (char)a1
r2 := (char)a2
r3 := (char)a3
```

ストリーミング SIMD 拡張命令のロード操作

「メモリ操作と初期化操作のまとめ」にある一覧表を参照してください。

ストリーミング SIMD 拡張命令の組込み関数のプロトタイプは、ヘッダファイル xmmintrin.h 内にあります。

```
__m128 _mm_load_ss(float * p )
```

単精度浮動小数点値を最下位ワードにロードし、上位 3 ワードをクリアします。

```
r0 := *p
r1 := 0.0 ; r2 := 0.0 ; r3 := 0.0
```

```
__m128 _mm_load_ps1(float * p )
```

1 つの単精度浮動小数点値をロードして、その値を 4 ワードすべてにコピーします。

```
r0 := *p
r1 := *p
r2 := *p
r3 := *p
```

```
__m128 _mm_load_ps(float * p )
```

4 つの単精度浮動小数点値をロードします。アドレスは 16 バイトにアライメントが合っていなければなりません。

```
r0 := p[0]
r1 := p[1]
r2 := p[2]
r3 := p[3]
```

```
__m128 _mm_loadu_ps(float * p)
```

4 つの単精度浮動小数点値をロードします。アドレスは 16 バイトにアライメントが合っていなくてもかまいません。

```
r0 := p[0]
r1 := p[1]
r2 := p[2]
r3 := p[3]
```

```
__m128 _mm_loadr_ps(float * p)
```

4 つの単精度浮動小数点値を逆順でロードします。アドレスは 16 バイトにアライメントが合っていなければなりません。

```
r0 := p[3]
r1 := p[2]
```

```
r2 := p[1]
r3 := p[0]
```

ストリーミング SIMD 拡張命令の設定操作

「メモリ操作と初期化操作のまとめ」にある一覧表を参照してください。

ストリーミング SIMD 拡張命令の組込み関数のプロトタイプは、ヘッダファイル xmmintrin.h 内にあります。

```
__m128 _mm_set_ss(float w )
```

単精度浮動小数点値の最下位ワードを w に設定し、上位 3 ワードをクリアします。

```
r0 := w
r1 := r2 := r3 := 0.0
```

```
__m128 _mm_set_ps1(float w )
```

4 つの単精度浮動小数点値を w に設定します。

```
r0 := r1 := r2 := r3 := w
```

```
__m128 _mm_set_ps(float z, float y, float x, float w )
```

4 つの単精度浮動小数点値を、4 つの入力値に設定します。

```
r0 := w
r1 := x
r2 := y
r3 := z
```

```
__m128 _mm_setr_ps(float z, float y, float x, float w )
```

4 つの単精度浮動小数点値を、逆順で 4 つの入力値に設定します。

```
r0 := z
r1 := y
r2 := x
r3 := w
```

```
__m128 _mm_setzero_ps(void)
```

4 つの単精度浮動小数点値をクリアします。

```
r0 := r1 := r2 := r3 := 0.0
```

ストリーミング SIMD 拡張命令のストア操作

「メモリ操作と初期化操作のまとめ」にある一覧表を参照してください。

ストリーミング SIMD 拡張命令の組込み関数のプロトタイプは、ヘッダファイル xmmintrin.h 内にあります。

```
void _mm_store_ss(float * p, __m128 a)
```

最下位の単精度浮動小数点値をストアします。

```
*p := a0
```

```
void _mm_store_ps1(float * p, __m128 a )
```

最下位の単精度浮動小数点値を 4 ワードにストアします。

```
p[0] := a0
p[1] := a0
p[2] := a0
p[3] := a0
```

```
void _mm_store_ps(float *p, __m128 a)
```

4 つの単精度浮動小数点値をストアします。アドレスは 16 バイトにアライメントが合っていなければなりません。

```
p[0] := a0
p[1] := a1
p[2] := a2
p[3] := a3
```

```
void _mm_storeu_ps(float *p, __m128 a)
```

4 つの単精度浮動小数点値をストアします。アドレスは 16 バイトにアライメントが合っていなくてもかまいません。

```
p[0] := a0
p[1] := a1
p[2] := a2
p[3] := a3
```

```
void _mm_storer_ps(float * p, __m128 a )
```

4 つの単精度浮動小数点値を逆順でストアします。アドレスは 16 バイトにアライメントが合っていなければなりません。

```
p[0] := a3
p[1] := a2
p[2] := a1
p[3] := a0
```

```
__m128 _mm_move_ss( __m128 a, __m128 b)
```

最下位ワードを、b の単精度浮動小数点値に設定します。上位 3 つの単精度浮動小数点値は a からそのまま渡されます。

```
r0 := b0
r1 := a1
r2 := a2
r3 := a3
```

ストリーミング SIMD 拡張命令によるキャッシュ制御

ストリーミング SIMD 拡張命令の組込み関数のプロトタイプは、ヘッダファイル xmmintrin.h 内にあります。

```
void _mm_pause(void)
```

プロセッサに固有の時間の間、次の命令の実行を遅らせます。この命令を実行しても、アーキテクチャ上の状態は変化しません。この組込み関数を使用すると、パフォーマンスが大きく向上します。この組込み関数については、以降で詳しく説明します。

PAUSE 組込み関数

PAUSE 組込み関数は、ダイナミック・エグゼキューション (特に、アウトオブオーダー実行) をサポートするプロセッサ上で、spin-wait ループに使用します。spin-wait ループ内で PAUSE を使用すると、ロックの解放を検出するコードの処理速度が向上します。動的スケジューリングに PAUSE 命令を使用すると、スピンループの終了時のペナルティが軽減されます。

PAUSE 命令を使用したループの例

```
spin loop:pause
cmp eax, A
jne spin_loop
```

上の例では、メモリ・ロケーション A がレジスタ eax の値と一致するまで、プログラムはスピンします。次のコード・シーケンスは、test-and-test-and-set 操作を示しています。この例では、ロックの取得に失敗した場合にのみ、スピンが発生します。

```
get lock: mov eax, 1
xchg eax, A ; Try to get lock
cmp eax, 0 ; Test if successful
jne spin_loop
```

クリティカル・セクション

```
// critical_section code
```

```
mov A, 0 ; Release lock
jmp continue
spin loop: pause;
// spin-loop hint
cmp 0, A ;
// check lock availability
jne spin loop
jmp get lock
// continue: other code
```

この例では、ロックの取得に成功すると予測して、最初の条件分岐は分岐せずに、そのままクリティカル・セクションの処理に移ります。すべての spin-wait ループに、PAUSE 命令を使用することを強くお勧めします。PAUSE は、既存のすべての IA-32 プロセッサで使用可能なため、プロセッサのタイプをテストする (CPUID テスト) 必要はありません。すべての従来のプロセッサは、PAUSE を NOP として実行しますが、PAUSE をヒントとして使用するプロセッサでは、パフォーマンスが大きく向上する可能性があります。

ストリーミング SIMD 拡張命令を使用する整数演算組込み関数

次の表に、整数演算組込み関数のリストを示します。表の後に、最新のニーモニック命名規則に従って、各組込み関数の説明を示します。

ストリーミング SIMD 拡張命令の組込み関数のプロトタイプは、ヘッダファイル xmmintrin.h 内にあります。

組込み関数の名前	別名	操作	対応する命令
_m_pextrw	_mm_extract_pi16	4 ワードのうち 1 つを抽出する	PEXTRW
_m_pinsrw	_mm_insert_pi16	1 ワードを挿入する	PINSRW
_m_pmaxsw	_mm_max_pi16	最大値を計算する	PMAXSW
_m_pmaxub	_mm_max_pu8	最大値を計算する (符号なし)	PMAXUB
_m_pminsw	_mm_min_pi16	最小値を計算する	PMINSW
_m_pminub	_mm_min_pu8	最小値を計算する (符号なし)	PMINUB
_m_pmovmskb	_mm_movemask_pi8	8 ビットマスクを作成する	PMOVMSKB
_m_pmulhuw	_mm_mulhi_pu16	乗算 (上位ビットを返す)	PMULHUW
_m_pshufw	_mm_shuffle_pi16	4 ワードを組み合わせて返す	PSHUFW
_m_maskmovq	_mm_maskmove_si64	条件付きストア	MASKMOVQ
_m_pavgb	_mm_avg_pu8	丸め平均を計算する	PAVGB
_m_pavgw	_mm_avg_pu16	丸め平均を計算する	PAVGW

_m_psadbw	_mm_sad_pu8	差の絶対値の合計を計算する	PSADBW

ここで説明する組込み関数を使用する場合は、mmx レジスタのマルチメディア・ステートを空にする必要があります。詳細については、「EMMS 命令: 必要な理由」の項目を参照してください。

```
int _m_pextrw(__m64 a, int n)
```

a の 4 ワードのうち 1 つを抽出します。セレクタ n は即値でなければなりません。

```
r := (n==0) ? a0 : ( (n==1) ? a1 : ( (n==2) ?
a2 : a3 ) )
```

```
__m64 _m_pinsrw(__m64 a, int d, int n)
```

a の 4 ワードのうち 1 つに、ワード d を挿入します。セレクタ n は即値でなければなりません。

```
r0 := (n==0) ? d : a0;
r1 := (n==1) ? d : a1;
r2 := (n==2) ? d : a2;
r3 := (n==3) ? d : a3;
```

```
__m64 _m_pmaxsw(__m64 a, __m64 b)
```

a と b のワードについて、要素ごとに最大値を計算します。

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

```
__m64 _m_pmaxub(__m64 a, __m64 b)
```

a と b の符号なしバイトについて、要素ごとに最大値を計算します。

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

```
__m64 _m_pminsw(__m64 a, __m64 b)
```

a と b のワードについて、要素ごとに最小値を計算します。

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

```
__m64 _m_pminub(__m64 a, __m64 b)
```

a と b の符号なしバイトについて、要素ごとに最小値を計算します。

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

```
int _m_pmovmskb(__m64 a)
```

a の各バイトの最上位ビットから、8 ビットマスクを作成します。

```
r := sign(a7)<<7 | sign(a6)<<6 |... | sign(a0)
```

```
__m64 _m_pmulhuw(__m64 a, __m64 b)
```

a と b の対応する符号なしワードを乗算し、得られた 32 ビットの中間結果の上位 16 ビットを返します。

```
r0 := hiword(a0 * b0)
r1 := hiword(a1 * b1)
r2 := hiword(a2 * b2)
r3 := hiword(a3 * b3)
```

```
__m64 _m_pshufw(__m64 a, int n)
```

a の 4 ワードを組み合わせて返します。セレクタ n は即値でなければなりません。

```
r0 := word (n&0x3) of a
r1 := word ((n>>2)&0x3) of a
r2 := word ((n>>4)&0x3) of a
r3 := word ((n>>6)&0x3) of a
```

```
void _m_maskmovq(__m64 d, __m64 n, char *p)
```

d のバイト要素を、条件付きでアドレス p にストアします。セレクタ n の各バイトの最上位ビットによって、それに対応する d の各バイトがストアされるかどうかが決まります。

```
if (sign(n0)) p[0] := d0
if (sign(n1)) p[1] := d1
...
if (sign(n7)) p[7] := d7
```

```
__m64 _m_pavgb(__m64 a, __m64 b)
```

a と b の対応する符号なしバイトについて、(丸め) 平均を計算します。

```
t = (unsigned short)a0 + (unsigned short)b0
r0 = (t >> 1) | (t & 0x01)
...
t = (unsigned short)a7 + (unsigned short)b7
r7 = (unsigned char)((t >> 1) | (t & 0x01))
```

```
__m64 _m_pavgw(__m64 a, __m64 b)
```

a と b の対応する符号なしワードについて、(丸め) 平均を計算します。

```
t = (unsigned int)a0 + (unsigned int)b0
r0 = (t >> 1) | (t & 0x01)
...
t = (unsigned word)a7 + (unsigned word)b7
r7 = (unsigned short)((t >> 1) | (t & 0x01))
```

```
__m64 _m_psadbw(__m64 a, __m64 b)
```

a と b の対応する符号なしバイトの差の絶対値の合計を計算し、最下位ワードの値を返します。上位 3 ワードはクリアされます。

```
r0 = abs(a0-b0) +... + abs(a7-b7)
r1 = r2 = r3 = 0
```

ストリーミング SIMD 拡張命令を使用するメモリ操作と初期化操作

このセクションでは、ロード操作、設定操作、およびストア操作を行う組込み関数について説明します。ロード組込み関数と設定組込み関数はよく似ており、いずれも __m128 型のデータを初期化します。しかし、設定組込み関数は、データを定数で初期化するための関数で、float 引数を使用します。ロード組込み関数は、メモリからデータをロードする命令を模倣するための関数で、浮動小数点引数を使用します。ストア組込み関数は、初期化したデータを、指定したアドレスに割り当てます。

次の表に、各種の組込み関数のリストを示します。以降の項には、各組込み関数の構文と簡単な説明を示します。

ストリーミング SIMD 拡張命令の組込み関数のプロトタイプは、ヘッダファイル xmmintrin.h 内にあります。

組込み関数の名前	別名	操作	対応する命令
_mm_load_ss		最下位の値をロードして、上位 3 つの値をクリアする	MOVSS
_mm_load_ps1	_mm_load1_ps	1 つの値を 4 ワードすべてにロードする	MOVSS + Shuffling
_mm_load_ps		4 つの値をロードする (アドレスのアライメントが合っていなければなりません)	MOVAPS
_mm_loadu_ps		4 つの値をロードする (アドレスのアライメントが合っている必要はありません)	MOVUPS
_mm_loadr_ps		4 つの値を逆順でロードする	MOVAPS + Shuffling

_mm_set_ss		最下位の値を設定し、上位 3 つの値をクリアする	複合
_mm_set_ps1	_mm_set1_ps	4 ワードすべてを同じ値に設定する	複合
_mm_set_ps		4 つの値を設定する (アドレスのアライメントが合っていなければなりません)	複合
_mm_setr_ps		4 つの値を逆順で設定する	複合
_mm_setzero_ps		4 つの値をすべてクリアする	複合
_mm_store_ss		最下位の値をストアする	MOVSS
_mm_store_ps1	_mm_store1_ps	最下位の値を 4 ワードすべてにストアする (アドレスは、16 バイトにアライメントが合っていなければなりません)	Shuffling + MOVSS
_mm_store_ps		4 つの値をストアする (アドレスのアライメントが合っていなければなりません)	MOVAPS
_mm_storeu_ps		4 つの値をストアする (アドレスのアライメントが合っている必要はありません)	MOVUPS
_mm_storer_ps		4 つの値を逆順でストアする	MOVAPS + Shuffling
_mm_move_ss		最下位ワードを設定し、上位 3 つの値はそのまま渡す	MOVSS
_mm_getcsr		レジスタの内容を返す	STMXCSR
_mm_setcsr		コントロール・レジスタ	LDMXCSR
_mm_prefetch			
_mm_stream_pi			
_mm_stream_ps			
_mm_sfence			
_mm_cvtss_f32			

```
__m128 _mm_load_ss(float const*a)
```

単精度浮動小数点値を最下位ワードにロードし、上位 3 ワードをクリアします。

```
r0 := *a
r1 := 0.0 ; r2 := 0.0 ; r3 := 0.0
```

```
__m128 _mm_load_ps1(float const*a)
```

1 つの単精度浮動小数点値をロードして、その値を 4 ワードすべてにコピーします。

```
r0 := *a
r1 := *a
r2 := *a
r3 := *a
```

```
__m128 _mm_load_ps(float const*a)
```

4 つの単精度浮動小数点値をロードします。アドレスは 16 バイトにアライメントが合っていなければなりません。

```
r0 := a[0]
r1 := a[1]
r2 := a[2]
r3 := a[3]
```

```
__m128 _mm_loadu_ps(float const*a)
```

4 つの単精度浮動小数点値をロードします。アドレスは 16 バイトにアライメントが合っている必要はありません。

```
r0 := a[0]
r1 := a[1]
r2 := a[2]
r3 := a[3]
```

```
__m128 _mm_loadr_ps(float const*a)
```

4 つの単精度浮動小数点値を逆順でロードします。アドレスは 16 バイトにアライメントが合っていなければなりません。

```
r0 := a[3]
r1 := a[2]
r2 := a[1]
r3 := a[0]
```

```
__m128 _mm_set_ss(float a)
```

単精度浮動小数点値の最下位ワードを a に設定し、上位 3 ワードをクリアします。

```
r0 := c
r1 := r2 := r3 := 0.0
```

```
__m128 _mm_set_ps1(float a)
```

4 つの単精度浮動小数点値を a に設定します。

```
r0 := r1 := r2 := r3 := a
```

```
__m128 _mm_set_ps(float a, float b, float c, float d)
```

4 つの単精度浮動小数点値を、4 つの入力値に設定します。

```
r0 := a
r1 := b
r2 := c
r3 := d
```

```
__m128 _mm_setr_ps(float a, float b, float c, float d)
```

4 つの単精度浮動小数点値を、逆順で 4 つの入力値に設定します。

```
r0 := d
r1 := c
r2 := b
r3 := a
```

```
__m128 _mm_setzero_ps(void)
```

4 つの単精度浮動小数点値をクリアします。

```
r0 := r1 := r2 := r3 := 0.0
```

```
void _mm_store_ss(float *v, __m128 a)
```

最下位の単精度浮動小数点値をストアします。

```
*v := a0
```

```
void _mm_store_ps1(float *v, __m128 a)
```

最下位の単精度浮動小数点値を 4 ワードにストアします。

```
v[0] := a0
v[1] := a0
v[2] := a0
v[3] := a0
```

```
void _mm_store_ps(float *v, __m128 a)
```

4 つの単精度浮動小数点値をストアします。アドレスは 16 バイトにアライメントが合っていなければなりません。

```
v[0] := a0
v[1] := a1
v[2] := a2
v[3] := a3
```

```
void _mm_storeu_ps(float *v, __m128 a)
```

4 つの単精度浮動小数点値をストアします。アドレスは 16 バイトにアライメントが合っている必要はありません。

```
v[0] := a0
v[1] := a1
v[2] := a2
v[3] := a3
```

```
void _mm_storer_ps(float *v, __m128 a)
```

4 つの単精度浮動小数点値を逆順でストアします。アドレスは 16 バイトにアライメントが合っていなければなりません。

```
v[0] := a3
v[1] := a2
v[2] := a1
v[3] := a0
```

```
__m128 _mm_move_ss(__m128 a, __m128 b)
```

最下位ワードを、b の単精度浮動小数点値に設定します。上位 3 つの単精度浮動小数点値は、a からそのまま渡されます。

```
r0 := b0
r1 := a1
r2 := a2
r3 := a3
```

```
unsigned int _mm_getcsr(void)
```

コントロール・レジスタの内容を返します。

```
void _mm_setcsr(unsigned int i)
```

コントロール・レジスタを指定された値に設定します。

```
void _mm_prefetch(char const*a, int sel)
```

(PREFETCH を使用) 1 キャッシュ・ライン分のデータを、アドレス a からプロセッサに "近い" 位置にロードします。sel の値は、プリフェッチ操作のタイプを指定します。この値には、プリフェッチ命令のタイプに応じて、定数 _MM_HINT_T0、_MM_HINT_T1、_MM_HINT_T2、または _MM_HINT_NTA を指定してください。

```
void _mm_stream_pi(__m64 *p, __m64 a)
```

(MOVNTQ を使用) a のデータを、キャッシュを介さずに、アドレス p にストアします。この組込み関数を使用する前に、mmx レジスタのマルチメディア・ステートを空にする必要があります。詳細については、「EMMS 命令: 必要な理由」を参照してください。

```
void _mm_stream_ps(float *p, __m128 a)
```

(MOVNTPS を参照) a のデータを、キャッシュを介さずに、アドレス p にストアします。アドレスは 16 バイトにアライメントが合っていなければなりません。

```
void _mm_sfence(void)
```

(SFENCE を使用) すべての先行するストアが、後に続くストアより前に、グローバルにアクセス可能になるのを保証します。

```
float _mm_cvtss_f32(__m128 a)
```

この組込み関数は __m128 の最初のベクトル要素から単精度浮動小数点値を抽出します。使用されるコンテキストで可能な最も効率的な方法で行われます。この組込み関数は特定の SSE 命令には対応付けられません。

ストリーミング SIMD 拡張命令を使用するその他の組込み関数

ストリーミング SIMD 拡張命令の組込み関数のプロトタイプは、ヘッダファイル xmmintrin.h 内にあります。

組込み関数の名前	操作	対応する命令
_mm_shuffle_ps	シャッフル	SHUFPS
_mm_unpackhi_ps	上位の値のアンパック	UNPCKHPS
_mm_unpacklo_ps	下位の値のアンパック	UNPCKLPS
_mm_loadh_pi	上位の値のロード	MOVHPS reg, mem
_mm_storeh_pi	上位の値のストア	MOVHPS mem, reg
_mm_movehl_ps	上位から下位への移動	MOVHLPS
_mm_movelh_ps	下位から上位への移動	MOVLHPS
_mm_loadl_pi	下位の値のロード	MOVLPS reg, mem
_mm_storel_pi	下位の値のストア	MOVLPS mem, reg
_mm_movemask_ps	4 ビットマスクの作成	MOVMSKPS

```
__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)
```

マスク imm8 に基づいて、a と b から 4 つの単精度浮動小数点値を選択します。マスクは即値でなければなりません。シャッフルのセマンティクスについては、「ストリーミング SIMD 拡張命令を使用してシャッフルを行うマクロ関数」を参照してください。

```
__m128 _mm_unpackhi_ps(__m128 a, __m128 b)
```

a と b から上位 2 つの単精度浮動小数点値を選択し、インターリーブ (交互に配置) します。

```
r0 := a2
r1 := b2
r2 := a3
r3 := b3
```

```
__m128 _mm_unpacklo_ps(__m128 a, __m128 b)
```

a と b から下位 2 つの単精度浮動小数点値を選択し、インターリーブします。

```
r0 := a0
r1 := b0
r2 := a1
r3 := b1
```

```
__m128 _mm_loadh_pi(__m128, __m64 const *p)
```

アドレス p からロードされた 64 ビットのデータで、上位 2 つの単精度浮動小数点値を設定します。

```
r0 := a0
r1 := a1
r2 := *p0
r3 := *p1
```

```
void _mm_storeh_pi(__m64 *p, __m128 a)
```

a の上位 2 つの単精度浮動小数点値を、アドレス p にストアします。

```
*p0 := a2
*p1 := a3
```

```
__m128 _mm_movehl_ps(__m128 a, __m128 b)
```

b の上位 2 つの単精度浮動小数点値を、結果の下位 2 つの単精度浮動小数点値に移動します。a の上位 2 つの単精度浮動小数点値は、そのまま結果に渡されます。

```
r3 := a3
r2 := a2
r1 := b3
r0 := b2
```

```
__m128 _mm_movelh_ps(__m128 a, __m128 b)
```

b の下位 2 つの単精度浮動小数点値を、結果の上位 2 つの単精度浮動小数点値に移動します。a の下位 2 つの単精度浮動小数点値は、そのまま結果に渡されます。

```
r3 := b1
r2 := b0
```

```
r1 := a1
r0 := a0
```

```
__m128 _mm_loadl_pi(__m128 a, __m64 const *p)
```

アドレス p からロードされた 64 ビットのデータで、下位 2 つの単精度浮動小数点値を設定します。上位 2 つの値は、a からそのまま渡されます。

```
r0 := *p0
r1 := *p1
r2 := a2
r3 := a3
```

```
void _mm_storel_pi(__m64 *p, __m128 a)
```

a の下位 2 つの単精度浮動小数点値を、アドレス p にストアします。

```
*p0 := a0
*p1 := a1
```

```
int _mm_movemask_ps(__m128 a)
```

4 つの単精度浮動小数点値の最上位ビットを使用して、4 ビットマスクを作成します。

```
r := sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 |
sign(a0)
```

Itanium® アーキテクチャ上でのストリーミング SIMD 拡張命令の使用

ストリーミング SIMD 拡張命令の組込み関数によって、Itanium® アーキテクチャ上でストリーミング SIMD 拡張命令を利用できます。IA-32 アーキテクチャとのソースコードの互換性を保つために、これらの組込み関数の名前と機能は、IA-32 ベースのストリーミング SIMD 拡張命令の組込み関数セットと同等になっています。

これらの組込み関数を使用してプログラムを作成するには、ストリーミング SIMD 拡張命令によって提供されるハードウェア機能をよく理解している必要があります。特に、次の点に注意してください:

- 一部の組込み関数は、以前に定義した IA-32 組込み関数との互換性のためにのみ用意されています。Itanium ベースのシステム上でこれらの組込み関数を使用すると、通常はパフォーマンスが低下します。下記の「互換性とパフォーマンス」の項を参照してください。
- __m128 オブジェクトとしてロードまたはストアされる浮動小数点 (FP) データは、16 バイトにアライメントが合っていなければなりません。

- 一部の組込み関数は、命令の性質上、引数を即値で、すなわち定数整数 (リテラル) で指定する必要があります。

データ型

ストリーミング SIMD 拡張命令の組込み関数は、新しいデータ型の `__m128` を使用します。このデータ型は、4 つの単精度浮動小数点値で構成される 128 ビットデータを表します。これは 128 ビットの IA-32 ストリーミング SIMD 拡張命令レジスタに対応します。

コンパイラは、`__m128` 型のローカルデータのアライメントを、スタック上の 16 バイト境界に合わせます。これらのデータ型のグローバル・データも、16 バイトにアライメントを合わせます。`integer` 型、`float` 型、または `double` 型の配列のアライメントを合わせるには、`declspec` ディレクティブを使用できます。

Itanium 命令は、パックドデータの操作でもスカラデータの操作でも、同じ方法でストリーミング SIMD 拡張命令レジスタを操作します。したがって、スカラデータを表す `__m32` データ型はありません。スカラ操作には、`__m128` オブジェクトと "スカラ" 形式の組込み関数を使用します。コンパイラとプロセッサは、32 ビットのメモリ参照によって、これらの操作を実行します。ただし、パフォーマンス上の理由で、できるだけスカラ形式の操作をパックド形式の操作で置き換えることをお勧めします。

`__m128` オブジェクトのアドレスを指定できます。

詳細については、『IA-32 インテル® アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、中巻：命令セット・リファレンス』インテル社、資料番号 245471J を参照してください。

Itanium ベース・システム上での実装手法

ストリーミング SIMD 拡張命令の組込み関数は、`__m128` データ型について定義しています。このデータ型は、4 つの単精度浮動小数点値で構成される 128 ビットデータです。Itanium ベースのシステム用の SIMD 命令は、2 つの単精度浮動小数点値を保持する 64 ビット浮動小数点レジスタのデータを操作します。したがって、それぞれの `__m128` オペランドは、実際には浮動小数点レジスタのペアになります。このため、各組込み関数は、浮動小数点レジスタ・オペランドのペアを操作する、少なくとも 2 つの Itanium 命令に対応します。

互換性とパフォーマンス

Itanium ベースのシステム用のストリーミング SIMD 拡張命令の組込み関数はほとんど、パフォーマンスの向上のためではなく、既存の IA-32 組込み関数との互換性を保つために用意されたものです。IA-32 システム上でパフォーマンスを向上させる組

込み関数を使用しても、Itanium ベースのシステム上ではパフォーマンスが向上しない場合もあります。この理由の 1 つとして、一部の組込み関数は、IA-32 命令セットには厳密に対応付けられますが、Itanium 命令セットには対応付けられていない場合があります。したがって、Itanium ベースのシステム上でのパフォーマンスの向上のために用意している組込み関数と、単に既存の IA-32 コードとの互換性を保つために用意している組込み関数は、区別して使用する必要があります。

次の組込み関数を使用すると、パフォーマンスが低下する可能性があります。これらの組込み関数は、既存のコードを移植する場合や、重要でないコード・セクションにのみ使用してください。

- ストリーミング SIMD 拡張命令のスカラ組込み関数 (名前の最後に _ss が付くもの) - できるだけパックド (_ps) 版を使用してください。
- ストリーミング SIMD 拡張命令の比較組込み関数 comi および ucomi - これらの組込み関数は、IA-32 命令 COMISS および UCOMISS にのみ対応します。これらの組込み関数を実行するには、一連の Itanium 命令を実行する必要があります。
- 変換操作には、通常は複数の命令が必要です。_mm_cvtpi16_ps、_mm_cvtpu16_ps、_mm_cvtpi8_ps、_mm_cvtpu8_ps、_mm_cvtpi32x2_ps、_mm_cvtps_pi16、_mm_cvtps_pi8 は、特にパフォーマンス上のコストがかかります。
- ストリーミング SIMD 拡張命令ユーティリティ組込み関数 _mm_movemask_ps

精度が多少低下してもかまわない場合は、真の div 組込み関数や sqrt 組込み関数の代わりに、逆数の近似値を計算する SIMD の組込み関数 (rcp) や逆数の平方根の近似値を計算する SIMD の組込み関数 (rsqrt) を使用すれば、処理速度が大幅に向上します。

ストリーミング SIMD 拡張命令を使用してシャッフルを行うマクロ関数

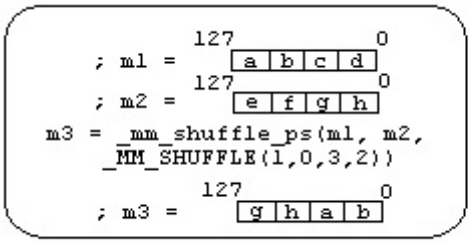
ストリーミング SIMD 拡張命令は、シャッフル操作を記述する定数を生成するマクロ関数を用意しています。このマクロは、4 つの小さな整数 (0～3 の範囲) を組み合わせて、SHUFPS 命令が使用する 8 ビット即値を生成します。次の例を参照してください。

シャッフル関数のマクロ

```
_MM_SHUFFLE(z,y,x,w)
/* expands to the following value */
(z<<6) | (y<<4) | (x<<2)| w
```

4 つの整数は、第 1 入力オペランドと第 2 入力オペランドからそれぞれどの 2 ワードを取り出して結果のワードに入れるかを選択するセレクタとして機能します。

シャッフル関数のマクロの元のワードと結果のワード



コントロール・レジスタを読み書きするマクロ関数

次のマクロ関数を使用して、コントロール・レジスタの各ビットを読み書きできます。詳細については、「設定操作」を参照してください。Itanium® ベースのシステムでは、これらのマクロではアクセスできない FPSR ビットもあります。詳細については、「Itanium® 命令のネイティブ組込み関数」の `getfpsr()` および `setfpsr()` 組込み関数を参照してください。

例外状態マクロ	マクロ引数
`_MM_SET_EXCEPTION_STATE(x)`	`_MM_EXCEPT_INVALID`
`_MM_GET_EXCEPTION_STATE()`	`_MM_EXCEPT_DIV_ZERO`
	`_MM_EXCEPT_DENORM`
マクロの定義 コントロール・レジスタの最下位から 6 番目のビットを読み書きします。	`_MM_EXCEPT_OVERFLOW`
	`_MM_EXCEPT_UNDERFLOW`
	`_MM_EXCEPT_INEXACT`

次の例では、ゼロ除算例外が発生したかどうかをテストします。

_MM_EXCEPT_DIV_ZERO マクロによる例外状態の確認

```
if (_MM_GET_EXCEPTION_STATE(x) & _MM_EXCEPT_DIV_ZERO) {
    /* Exception has occurred */
}
```

例外マスクマクロ	マクロ引数
`_MM_SET_EXCEPTION_MASK(x)`	`_MM_MASK_INVALID`

_MM_GET_EXCEPTION_MASK ()	_MM_MASK_DIV_ZERO
	_MM_MASK_DENORM
マクロの定義 コントロール・レジスタのビット 7～ビット 12 を読み書きします。 **注**: 6 つの例外マスクビットのすべてが、常に影響を受けます。明示的にセットされないビットはクリアされます。	_MM_MASK_OVERFLOW
	_MM_MASK_UNDERFLOW
	_MM_MASK_INEXACT

次の例では、オーバーフロー例外とアンダーフロー例外をマスクし、その他のすべての例外をマスク解除します。

_MM_MASK_OVERFLOW と _MM_MASK_UNDERFLOW による例外マスクの変更

```
_MM_SET_EXCEPTION_MASK(MM_MASK_OVERFLOW | _MM_MASK_UNDERFLOW)
```

丸めモード	マクロ引数
_MM_SET_ROUNDING_MODE(x)	_MM_ROUND_NEAREST
_MM_GET_ROUNDING_MODE()	_MM_ROUND_DOWN
マクロの定義 コントロール・レジスタのビット 13 とビット 14 を読み書きします。	_MM_ROUND_UP
	_MM_ROUND_TOWARD_ZERO

次の例では、丸めモードがゼロ方向への丸めになっているかどうかをテストします。

_MM_ROUND_TOWARD_ZERO による丸めモードの確認

```
if (_MM_GET_ROUNDING_MODE() == _MM_ROUND_TOWARD_ZERO) {
/* Rounding mode is round toward zero */
}
```

ゼロ・フラッシュ・モード	マクロ引数
_MM_SET_FLUSH_ZERO_MODE(x)	_MM_FLUSH_ZERO_ON
_MM_GET_FLUSH_ZERO_MODE()	_MM_FLUSH_ZERO_OFF
マクロの定義 コントロール・レジスタのビット 15 を読み書きします。	

次の例では、ゼロ・フラッシュ・モードを無効にします。

_MM_FLUSH_ZERO_OFF によるゼロ・フラッシュ・モードの変更
`_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_OFF)`

行列の転置を行うマクロ関数

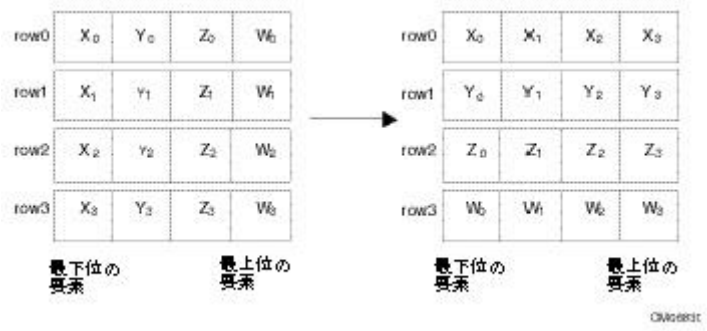
ストリーミング SIMD 拡張命令は、単精度浮動小数点値の 4×4 行列を転置する、次のマクロ関数も用意しています。

```
_MM_TRANSPOSE4_PS(row0, row1, row2, row3)
```

引数 `row0`、`row1`、`row2`、`row3` は `__m128` 値であり、引数内の各要素が、4×4 行列の行に対応します。転置した行列は、引数 `row0`、`row1`、`row2`、`row3` の形式で返されます。`row0` には元の行列の列 0 が格納され、`row1` には元の行列の列 1 が格納されます (以下同様)。

次の図「`_MM_TRANSPOSE4_PS` マクロによる行列の転置」に、このマクロの転置機能を示します。

_MM_TRANSPOSE4_PS マクロによる行列の転置



ストリーミング SIMD 拡張命令 2

概要: ストリーミング SIMD 拡張命令 2 の組込み関数

このセクションでは、インテル® C++ コンパイラ内でインテル® Pentium® 4 プロセッサのストリーミング SIMD 拡張命令 2 をサポートする、C++ 言語レベルの機能について説明します。これらの機能は、次の 2 種類に分類されます:

- 浮動小数点組込み関数 -- 倍精度浮動小数点データ型 (__m128d) に対する、算術演算、論理演算、比較、変換、メモリ操作、初期化を行う組込み関数について説明します。
- 整数組込み関数 -- 拡張精度整数データ型 (__m128i) に対する、算術演算、論理演算、比較、変換、メモリ操作、初期化を行う組込み関数について説明します。

注

Pentium 4 プロセッサのストリーミング SIMD 拡張命令 2 の組込み関数は、IA-32 プラットフォーム用にのみ定義されており、Itanium® ベースのプラットフォーム用には定義されていません。Pentium 4 プロセッサのストリーミング SIMD 拡張命令 2 は、128 ビットデータ (2 つの 64 ビット倍精度浮動小数点値) を操作します。Pentium 4 プロセッサのストリーミング SIMD 拡張命令 2 は、Itanium ベースのシステム上では使用できません。

詳細については、『Pentium® 4 processor Streaming SIMD Extensions 2 External Architecture Specification (EAS)』(英語) などの Pentium 4 プロセッサのマニュアル (developer.intel.com からダウンロード可能) を参照してください。ストリーミング SIMD 拡張命令 2 の組込み関数を使用してプログラムを作成する際は、ストリーミング SIMD 拡張命令 2 によって提供されるハードウェア機能についてよく理解している必要があります。特に、次の 3 つの点に注意してください:

- _mm_loadr_pd や _mm_cmpgt_sd などの一部の組込み関数は、命令セットによって直接にはサポートされていません。これらの組込み関数は、プログラミング上の便宜のために用意されたものであり、実行時にはコストがかかります。
- __m128d オブジェクトとしてロードまたはストアされるデータは、通常は 16 バイトにアライメントが合っていなければなりません。
- 一部の組込み関数は、命令の性質上、引数を即値で、すなわち定数整数 (リテラル) で指定する必要があります。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

ストリーミング SIMD 拡張命令 2 の浮動小数点算術演算

次の表に、ストリーミング SIMD 拡張命令 2 の算術演算組込み関数のリストを示します。表の後に、各組込み関数の説明を示します。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

組込み関数の名前	対応する命令		R0 の値	R1 の値
_mm_add_sd	ADDSD	加算	a0 [op] b0	a1
_mm_add_pd	ADDPD	加算	a0 [op] b0	a1 [op] b1
_mm_sub_sd	SUBSD	減算	a0 [op] b0	a1
_mm_sub_pd	SUBPD	減算	a0 [op] b0	a1 [op] b1
_mm_mul_sd	MULSD	乗算	a0 [op] b0	a1
_mm_mul_pd	MULPD	乗算	a0 [op] b0	a1 [op] b1
_mm_div_sd	DIVSD	除算	a0 [op] b0	a1
_mm_div_pd	DIVPD	除算	a0 [op] b0	a1 [op] b1
_mm_sqrt_sd	SQRTSD	平方根の計算	a0 [op] b0	a1
_mm_sqrt_pd	SQRTPD	平方根の計算	a0 [op] b0	a1 [op] b1
_mm_min_sd	MINSD	最小値の計算	a0 [op] b0	a1
_mm_min_pd	MINPD	最小値の計算	a0 [op] b0	a1 [op] b1
_mm_max_sd	MAXSD	最大値の計算	a0 [op] b0	a1
_mm_max_pd	MAXPD	最大値の計算	a0 [op] b0	a1 [op] b1

```
__m128d _mm_add_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値を加算します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := a0 + b0
r1 := a1
```

```
__m128d _mm_add_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値を加算します。

```
r0 := a0 + b0
r1 := a1 + b1
```

```
__m128d _mm_sub_sd(__m128d a, __m128d b)
```

a の下位の倍精度浮動小数点値から、b の下位の倍精度浮動小数点値を引きます。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := a0 - b0
r1 := a1
```

```
__m128d _mm_sub_pd(__m128d a, __m128d b)
```

a の 2 つの倍精度浮動小数点値から、b の 2 つの倍精度浮動小数点値を引きます。

```
r0 := a0 - b0
r1 := a1 - b1
```

```
__m128d _mm_mul_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値を乗算します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := a0 * b0
r1 := a1
```

```
__m128d _mm_mul_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値を乗算します。

```
r0 := a0 * b0
r1 := a1 * b1
```

```
__m128d _mm_div_sd(__m128d a, __m128d b)
```

a の下位の倍精度浮動小数点値を、b の下位の倍精度浮動小数点値で割ります。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := a0 / b0
r1 := a1
```

```
__m128d _mm_div_pd(__m128d a, __m128d b)
```

a の 2 つの倍精度浮動小数点値を、b の 2 つの倍精度浮動小数点値で割ります。

```
r0 := a0 / b0
r1 := a1 / b1
```

```
__m128d _mm_sqrt_sd(__m128d a, __m128d b)
```

b の下位の倍精度浮動小数点値の平方根を計算します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := sqrt(b0)
r1 := a1
```

```
__m128d _mm_sqrt_pd(__m128d a)
```

a の 2 つの倍精度浮動小数点値の平方根を計算します。

```
r0 := sqrt(a0)
r1 := sqrt(a1)
```

```
__m128d _mm_min_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、小さい方の値を計算します。上位の倍精度浮動小数点値は、a からそのまま渡されます。
```
r0 := min (a0, b0)
r1 := a1
```

```
__m128d _mm_min_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値について、それぞれ小さい方の値を計算します。
```
r0 := min(a0, b0)
r1 := min(a1, b1)
```

```
__m128d _mm_max_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、大きい方の値を計算します。上位の倍精度浮動小数点値は、a からそのまま渡されます。
```
r0 := max (a0, b0)
r1 := a1
```

```
__m128d _mm_max_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値について、それぞれ大きい方の値を計算します。
```
r0 := max(a0, b0)
r1 := max(a1, b1)
```

ストリーミング SIMD 拡張命令 2 の論理演算

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

```
__m128d _mm_and_pd(__m128d a, __m128d b)
```

(ANDPD を使用) a と b の 2 つの倍精度浮動小数点値について、ビット単位の AND (論理積) を計算します。
```
r0 := a0 & b0
r1 := a1 & b1
```

```
__m128d _mm_andnot_pd(__m128d a, __m128d b)
```

(ANDNPD を使用) a の 128 ビット値のビット単位の NOT (否定) を実行し、その結果と b の 128 ビット値について、ビット単位の AND (論理積) を計算します。
```
r0 := (~a0) & b0
r1 := (~a1) & b1
```

```
__m128d _mm_or_pd(__m128d a, __m128d b)
```

(ORPD を使用) a と b の 2 つの倍精度浮動小数点値について、ビット単位の OR (論理和) を計算します。

```
r0 := a0 | b0
r1 := a1 | b1
```

```
__m128d _mm_xor_pd(__m128d a, __m128d b)
```

(XORPD を使用) a と b の 2 つの倍精度浮動小数点値について、ビット単位の XOR (排他的論理和) を計算します。

```
r0 := a0 ^ b0
r1 := a1 ^ b1
```

ストリーミング SIMD 拡張命令 2 の比較操作

比較組込み関数は、a と b の比較を実行します。パックド形式の場合は、a と b の 2 つの倍精度浮動小数点値を比較して、128 ビットマスクを返します。スカラ形式の場合は、a と b の下位の倍精度浮動小数点値を比較して、64 ビットマスクを返します。上位の倍精度浮動小数点値は、a からそのまま渡されます。マスクは、各要素について、比較の結果が真の場合は 0xffffffffffffffff に設定し、偽の場合は 0x0 に設定します。命令名の後の r は、SIMD 命令の実行時にオペランドが逆順にされることを示します。次の表に、ストリーミング SIMD 拡張命令 2 の比較組込み関数のリストを示します。表の後に、各組込み関数の説明を示します。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

組込み関数の名前	対応する命令	比較の条件
_mm_cmpeq_pd	CMPEQPD	等しい
_mm_cmplt_pd	CMPLTPD	より小さい
_mm_cmple_pd	CMPLEPD	以下
_mm_cmpgt_pd	CMPLTPDr	より大きい
_mm_cmpge_pd	CMPLEPDr	以上
_mm_cmpord_pd	CMPORDPD	順序化可能
_mm_cmpunord_pd	CMPUNORDPD	順序化不可能
_mm_cmpneq_pd	CMPNEQPD	等しくない
_mm_cmpnlt_pd	CMPNLTPD	より小さくない
_mm_cmpnle_pd	CMPNLEPD	以下でない

_mm_cmpngt_pd	CMPNLTPDr	より大きくない
_mm_cmpnge_pd	CMPLEPDr	以上でない
_mm_cmpeq_sd	CMPEQSD	等しい
_mm_cmplt_sd	CMPLTSD	より小さい
_mm_cmple_sd	CMPLESD	以下
_mm_cmpgt_sd	CMPLTSDr	より大きい
_mm_cmpge_sd	CMPLESDr	以上
_mm_cmpord_sd	CMPORDSD	順序化可能
_mm_cmpunord_sd	CMPUNORDSD	順序化不可能
_mm_cmpneq_sd	CMPNEQSD	等しくない
_mm_cmpnlt_sd	CMPNLTSD	より小さくない
_mm_cmpnle_sd	CMPNLESD	以下でない
_mm_cmpngt_sd	CMPNLTSDr	より大きくない
_mm_cmpnge_sd	CMPNLESDR	以上でない
_mm_comieq_sd	COMISD	等しい
_mm_comilt_sd	COMISD	より小さい
_mm_comile_sd	COMISD	以下
_mm_comigt_sd	COMISD	より大きい
_mm_comige_sd	COMISD	以上
_mm_comineq_sd	COMISD	等しくない
_mm_ucomieq_sd	UCOMISD	等しい
_mm_ucomilt_sd	UCOMISD	より小さい
_mm_ucomile_sd	UCOMISD	以下
_mm_ucomigt_sd	UCOMISD	より大きい
_mm_ucomige_sd	UCOMISD	以上
_mm_ucomineq_sd	UCOMISD	等しくない

```
__m128d _mm_cmpeq_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値が等しいかどうか比較します。

```
r0 := (a0 == b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 == b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmplt_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値について、a が b より小さいかどうか比較します。

```
r0 := (a0 < b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 < b1) ? 0xffffffffffffffff : 0x0
```

```
___m128d _mm_cmple_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値について、a が b 以下かどうか比較します。

```
r0 := (a0 <= b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 <= b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpgt_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値について、a が b より大きいかどうか比較します。

```
r0 := (a0 > b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 > b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpge_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値について、a が b 以上かどうか比較します。

```
r0 := (a0 >= b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 >= b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpord_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値が順序化可能かどうか判定します。

```
r0 := (a0 ord b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 ord b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpunord_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値が順序化不可能かどうか判定します。

```
r0 := (a0 unord b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 unord b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpneq_pd ( __m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値が等しくないかどうか比較します。

```
r0 := (a0 != b0) ? 0xffffffffffffffff : 0x0
r1 := (a1 != b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpnlt_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値について、a が b より小さくないかどうか比較します。

```
r0 := !(a0 < b0) ? 0xffffffffffffffff : 0x0
r1 := !(a1 < b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpnle_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値について、a が b 以下でないかどうか比較します。

```
r0 := !(a0 <= b0) ? 0xffffffffffffffff : 0x0
r1 := !(a1 <= b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpngt_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値について、a が b より大きくないかどうか比較します。

```
r0 := !(a0 > b0) ? 0xffffffffffffffff : 0x0
r1 := !(a1 > b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpnge_pd(__m128d a, __m128d b)
```

a と b の 2 つの倍精度浮動小数点値について、a が b 以上でないかどうか比較します。

```
r0 := !(a0 >= b0) ? 0xffffffffffffffff : 0x0
r1 := !(a1 >= b1) ? 0xffffffffffffffff : 0x0
```

```
__m128d _mm_cmpeq_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値が等しいかどうか比較します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := (a0 == b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmplt_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b より小さいかどうか比較します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := (a0 < b0) ? 0xffffffffffffffff : 0x0
r1 := i1
```

```
__m128d _mm_cmple_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b 以下かどうか比較します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := (a0 <= b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpgt_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b より大きいかどうか比較します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := (a0 > b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpge_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b 以上かどうか比較します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := (a0 >= b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpord_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値が順序化可能かどうか判定します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := (a0 ord b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpunord_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値が順序化不可能かどうか判定します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := (a0 unord b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpneq_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値が等しくないかどうか比較します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := (a0 != b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpnlt_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b より小さくないかどうか比較します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := !(a0 < b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpnle_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b 以下でないかどうか比較します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := !(a0 <= b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpngt_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b より大きくないかどうか比較します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := !(a0 > b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
__m128d _mm_cmpnge_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b 以上でないかどうか比較します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := !(a0 >= b0) ? 0xffffffffffffffff : 0x0
r1 := a1
```

```
int _mm_comieq_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a と b が等しいかどうか比較します。a と b が等しい場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 == b0) ? 0x1 : 0x0
```

```
int _mm_comilt_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b より小さいかどうか比較します。a が b より小さい場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 < b0) ? 0x1 : 0x0
```

```
int _mm_comile_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b 以下かどうか比較します。a が b 以下の場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 <= b0) ? 0x1 : 0x0
```

```
int _mm_comigt_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b より大きいかどうか比較します。a が b より大きい場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 > b0) ? 0x1 : 0x0
```

```
int _mm_comige_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b 以上かどうか比較します。a が b 以上の場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 >= b0) ? 0x1 : 0x0
```

```
int _mm_comineq_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a と b が等しくないかどうか比較します。a と b が等しくない場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 != b0) ? 0x1 : 0x0
```

```
int _mm_ucomieq_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a と b が等しいかどうか比較します。a と b が等しい場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 == b0) ? 0x1 : 0x0
```

```
int _mm_ucomilt_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b より小さいかどうか比較します。a が b より小さい場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 < b0) ? 0x1 : 0x0
```

```
int _mm_ucomile_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b 以下かどうか比較します。a が b 以下の場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 <= b0) ? 0x1 : 0x0
```

```
int _mm_ucomigt_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b より大きいかどうか比較します。a が b より大きい場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 > b0) ? 0x1 : 0x0
```

```
int _mm_ucomige_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a が b 以上かどうか比較します。a が b 以上の場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 >= b0) ? 0x1 : 0x0
```

```
int _mm_ucomineq_sd(__m128d a, __m128d b)
```

a と b の下位の倍精度浮動小数点値について、a と b が等しくないかどうか比較します。a と b が等しくない場合は、1 を返します。それ以外の場合は、0 を返します。

```
r := (a0 != b0) ? 0x1 : 0x0
```

ストリーミング SIMD 拡張命令 2 の変換操作

変換組込み関数は、データ型を他のデータ型に変換します。_mm_cvtpd_ps などの変換を実行すると、データの精度が低下します。このような場合に使用する丸めモードは、MXCSR レジスタの値によって決まります。デフォルトの丸めモードは、最近値への丸めです。ただし、C および C++ 言語は、型変換の実行時に切り捨てモードを使用します。_mm_cvttpd_epi32 および _mm_cvttsd_si32 組込み関数は、MXCSR レジスタで指定した丸めモードに関係なく、切り捨てモードを使用します。

次の表に、ストリーミング SIMD 拡張命令 2 の変換操作組込み関数のリストを示します。表の後に、各組込み関数の説明を示します。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

組込み関数の名前	対応する命令	戻り値の種類	パラメータ
_mm_cvtpd_ps	CVTPD2PS	__m128	(__m128d a)
_mm_cvtps_pd	CVTPS2PD	__m128d	(__m128 a)
_mm_cvtepi32_pd	CVTDQ2PD	__m128d	(__m128i a)
_mm_cvtpd_epi32	CVTPD2DQ	__m128i	(__m128d a)
_mm_cvtsd_si32	CVTSD2SI	int	(__m128d a)
_mm_cvtsd_ss	CVTSD2SS	__m128	(__m128 a, __m128d b)
_mm_cvtsi32_sd	CVTSI2SD	__m128d	(__m128d a, int b)
_mm_cvtss_sd	CVTSS2SD	__m128d	(__m128d a, __m128 b)
_mm_cvttpd_epi32	CVTTPD2DQ	__m128i	(__m128d a)
_mm_cvttsd_si32	CVTTSD2SI	int	(__m128d a)

_mm_cvtpd_pi32	CVTPD2PI	__m64	(__m128d a)
_mm_cvttpd_pi32	CVTTPD2PI	__m64	(__m128d a)
_mm_cvtpi32_pd	CVTPI2PD	__m128d	(__m64 a)
_mm_cvtsd_f64	なし	double	(__m128d a)

```
__m128 _mm_cvtpd_ps(__m128d a)
```

a の 2 つの倍精度浮動小数点値を単精度浮動小数点値に変換します。

```
r0 := (float) a0
r1 := (float) a1
r2 := 0.0 ; r3 := 0.0
```

```
__m128d _mm_cvtps_pd(__m128 a)
```

a の下位 2 つの単精度浮動小数点値を倍精度浮動小数点値に変換します。

```
r0 := (double) a0
r1 := (double) a1
```

```
__m128d _mm_cvtepi32_pd(__m128i a)
```

a の下位 2 つの符号付き 32 ビット整数値を倍精度浮動小数点値に変換します。

```
r0 := (double) a0
r1 := (double) a1
```

```
__m128i _mm_cvtpd_epi32(__m128d a)
```

a の 2 つの倍精度浮動小数点値を符号付き 32 ビット整数値に変換します。

```
r0 := (int) a0
r1 := (int) a1
r2 := 0x0 ; r3 := 0x0
```

```
int _mm_cvtsd_si32(__m128d a)
```

a の下位の倍精度浮動小数点値を符号付き 32 ビット整数値に変換します。

```
r := (int) a0
```

```
__m128 _mm_cvtsd_ss(__m128 a, __m128d b)
```

b の下位の倍精度浮動小数点値を単精度浮動小数点値に変換します。a の上位の単精度浮動小数点値はそのまま渡されます。

```
r0 := (float) b0
r1 := a1; r2 := a2 ; r3 := a3
```

```
__m128d _mm_cvtsi32_sd(__m128d a, int b)
```

b の符号付き整数値を倍精度浮動小数点値に変換します。a の上位の倍精度浮動小数点値はそのまま渡されます。

```
r0 := (double) b
r1 := a1
```

```
__m128d _mm_cvtss_sd(__m128d a, __m128 b)
```

b の最下位の単精度浮動小数点値を倍精度浮動小数点値に変換します。a の上位の倍精度浮動小数点値はそのまま渡されます。

```
r0 := (double) b0
r1 := a1
```

```
__m128i _mm_cvttpd_epi32(__m128d a)
```

切り捨てを使用して、a の 2 つの倍精度浮動小数点値を符号付き 32 ビット整数値に変換します。

```
r0 := (int) a0
r1 := (int) a1
r2 := 0x0 ; r3 := 0x0
```

```
int _mm_cvttsd_si32(__m128d a)
```

切り捨てを使用して、a の下位の倍精度浮動小数点値を符号付き 32 ビット整数に変換します。

```
r := (int) a0
```

```
__m64 _mm_cvtpd_pi32(__m128d a)
```

a の 2 つの倍精度浮動小数点値を符号付き 32 ビット整数値に変換します。

```
r0 := (int) a0
r1 := (int) a1
```

```
__m64 _mm_cvttpd_pi32(__m128d a)
```

切り捨てを使用して、a の 2 つの倍精度浮動小数点値を符号付き 32 ビット整数値に変換します。

```
r0 := (int) a0
r1 := (int) a1
```

```
__m128d _mm_cvtpi32_pd(__m64 a)
```

a の 2 つの符号付き 32 ビット整数値を倍精度浮動小数点値に変換します。

```
r0 := (double) a0
r1 := (double) a1
```

```
_mm_cvtsd_f64(__m128d a)
```

この組込み関数は __m128d の最初のベクトル要素から倍精度浮動小数点値を抽出します。使用されるコンテキストで可能な最も効率的な方法で行われます。この組込み関数は特定の SSE2 命令には対応付けられません。

ストリーミング SIMD 拡張命令 2 の浮動小数点メモリ操作と初期化操作

このセクションでは、ロード操作、設定操作、およびストア操作を行う組込み関数について説明します。ロード組込み関数と設定組込み関数はよく似ており、いずれも __m128d 型のデータを初期化します。しかし、設定組込み関数は、データを定数で初期化するための関数で、double 引数を使用します。ロード組込み関数は、メモリからデータをロードする命令を模倣するための関数で、double ポインタ引数を使用します。ストア組込み関数は、初期化したデータを、指定したアドレスに割り当てます。

注

移動操作の組込み関数はありません。1 つのレジスタから別のレジストへデータを移動するには、A = B のように単に代入します。A と B は移動操作のソースで対象レジスタです。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

ストリーミング SIMD 拡張命令 2 のロード操作

次のロード操作組込み関数とそれに対応する命令は、ストリーミング SIMD 拡張命令 2 をサポートするプロセッサ上で有効です。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

```
__m128d _mm_load_pd(double const*dp)
```

(MOVAPD を使用) 2 つの倍精度浮動小数点値をロードします。アドレス dp は、16 バイトにアライメントが合っていなければなりません。

```
r0 := p[0]
r1 := p[1]
```

```
__m128d _mm_load1_pd(double const*dp)
```

(MOVSD とシャッフリングを使用) 1 つの倍精度浮動小数点値をロードして、その値を両方の要素にコピーします。アドレス dp は、16 バイトにアライメントが合っていなくてもかまいません。

```
r0 := *p
r1 := *p
```

```
__m128d _mm_loadr_pd(double const*dp)
```

(MOVAPD とシャッフリングを使用) 2 つの倍精度浮動小数点値を逆順でロードします。アドレス dp は、16 バイトにアライメントが合っていなければなりません。

```
r0 := p[1]
r1 := p[0]
```

```
__m128d _mm_loadu_pd(double const*dp)
```

(MOVUPD を使用) 2 つの倍精度浮動小数点値をロードします。アドレス dp は、16 バイトにアライメントが合っていなくてもかまいません。

```
r0 := p[0]
r1 := p[1]
```

```
__m128d _mm_load_sd(double const*dp)
```

(MOVSD を使用) 1 つの倍精度浮動小数点値をロードします。上位の倍精度浮動小数点値は 0 に設定されます。アドレス dp は、16 バイトにアライメントが合っていなくてもかまいません。

```
r0 := *p
r1 := 0.0
```

```
__m128d _mm_loadh_pd(__m128d a, double const*dp)
```

(MOVHPD を使用) 結果の上位の倍精度浮動小数点値として、1 つの倍精度浮動小数点値をロードします。下位の倍精度浮動小数点値は、a からそのまま渡されます。アドレス dp は、16 バイトにアライメントが合っていなくてもかまいません。

```
r0 := a0
r1 := *p
```

```
__m128d _mm_loadl_pd(__m128d a, double const*dp)
```

(MOVLPD を使用) 結果の下位の倍精度浮動小数点値として、1 つの倍精度浮動小数点値をロードします。上位の倍精度浮動小数点値は、a からそのまま渡されます。アドレス dp は、16 バイトにアライメントが合っていなくてもかまいません。

```
r0 := *p
r1 := a1
```

ストリーミング SIMD 拡張命令 2 の設定操作

次の設定操作組込み関数とそれに対応する命令は、ストリーミング SIMD 拡張命令 2 をサポートするプロセッサ上で有効です。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

```
__m128d _mm_set_sd(double w)
```

(複合) 下位の倍精度浮動小数点値を w に設定し、上位の倍精度浮動小数点値を 0 に設定します。

```
r0 := w
r1 := 0.0
```

```
__m128d _mm_set1_pd(double w)
```

(複合) 2 つの倍精度浮動小数点値を w に設定します。

```
r0 := w
r1 := w
```

```
__m128d _mm_set_pd(double w, double x)
```

(複合) 下位の倍精度浮動小数点値を x に設定し、上位の倍精度浮動小数点値を w に設定します。

```
r0 := x
r1 := w
```

```
__m128d _mm_setr_pd(double w, double x)
```

(複合) 下位の倍精度浮動小数点値を w に設定し、上位の倍精度浮動小数点値を x に設定します。

```
r0 := w
r1 := x
```

```
__m128d _mm_setzero_pd(void)
```

(XORPD を使用) 2 つの倍精度浮動小数点値を 0 に設定します。

```
r0 := 0.0
r1 := 0.0
```

```
__m128d _mm_move_sd( __m128d a, __m128d b)
```

(MOVSD を使用) 下位の倍精度浮動小数点値を、b の下位の倍精度浮動小数点値に設定します。上位の倍精度浮動小数点値は、a からそのまま渡されます。

```
r0 := b0
r1 := a1
```

ストリーミング SIMD 拡張命令 2 のストア操作

次のストア操作組込み関数とそれに対応する命令は、ストリーミング SIMD 拡張命令 2 をサポートするプロセッサ上で有効です。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

```
void _mm_store_sd(double *dp, __m128d a)
```

(MOVSD を使用) a の下位の倍精度浮動小数点値をストアします。アドレス dp は、16 バイトにアライメントが合っていなくてもかまいません。

```
*dp := a0
```

```
void _mm_store1_pd(double *dp, __m128d a)
```

(MOVAPD とシャッフリングを使用) a の下位の倍精度浮動小数点値を 2 回ストアします。アドレス dp は、16 バイトにアライメントが合っていなければなりません。

```
dp[0] := a0
dp[1] := a0
```

```
void _mm_store_pd(double *dp, __m128d a)
```

(MOVAPD を使用) 2 つの倍精度浮動小数点値をストアします。アドレス dp は、16 バイトにアライメントが合っていなければなりません。

```
dp[0] := a0
dp[1] := a1
```

```
void _mm_storeu_pd(double *dp, __m128d a)
```

(MOVUPD を使用) 2 つの倍精度浮動小数点値をストアします。アドレス dp は、16 バイトにアライメントが合っていなくてもかまいません。

```
dp[0] := a0
dp[1] := a1
```

```
void _mm_storer_pd(double *dp, __m128d a)
```

(MOVAPD とシャッフリングを使用) 2 つの倍精度浮動小数点値を逆順でストアします。アドレス dp は、16 バイトにアライメントが合っていなければなりません。

```
dp[0] := a1
dp[1] := a0
```

```
void _mm_storeh_pd(double *dp, __m128d a)
```

(MOVHPD を使用) a の上位の倍精度浮動小数点値をストアします。

```
*dp := a1
```

```
void _mm_storel_pd(double *dp, __m128d a)
```

(MOVLPD を使用) a の下位の倍精度浮動小数点値をストアします。

```
*dp := a0
```

ストリーミング SIMD 拡張命令 2 のその他の操作

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

```
__m128d _mm_unpackhi_pd(__m128d a, __m128d b)
```

(UNPCKHPD を使用) a と b の上位の倍精度浮動小数点値をインターリーブ (交互に配置) します。

```
r0 := a1
r1 := b1
```

```
__m128d _mm_unpacklo_pd(__m128d a, __m128d b)
```

(UNPCKLPD を使用) a と b の下位の倍精度浮動小数点値をインターリーブします。

```
r0 := a0
r1 := b0
```

```
int _mm_movemask_pd(__m128d a)
```

(MOVMSKPD を使用) a の 2 つの倍精度浮動小数点値の符号ビットから、2 ビットマスクを作成します。

```
r := sign(a1) << 1 | sign(a0)
```

```
__m128d _mm_shuffle_pd(__m128d a, __m128d b, int i)
```

(SHUFPD を使用) マスク i に基づいて、a と b から 2 つの倍精度浮動小数点値を選択します。マスクは即値でなければなりません。シャッフルのセマンティクスについては、「シャッフルを行うマクロ関数」を参照してください。

ストリーミング SIMD 拡張命令 2 の整数算術演算

次の表に、ストリーミング SIMD 拡張命令 2 の整数算術演算組込み関数のリストを示します。表の後に、各組込み関数の説明を示します。ストリーミング SIMD 拡張命令 2 のパックド算術演算組込み関数については、「浮動小数点算術演算」を参照してください。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル `emmintrin.h` 内にあります。

組込み関数	命令	操作
_mm_add_epi8	PADDB	加算
_mm_add_epi16	PADDW	加算
_mm_add_epi32	PADDD	加算
_mm_add_si64	PADDQ	加算
_mm_add_epi64	PADDQ	加算
_mm_adds_epi8	PADDSB	加算
_mm_adds_epi16	PADDSW	加算
_mm_adds_epu8	PADDUSB	加算
_mm_adds_epu16	PADDUSW	加算
_mm_avg_epu8	PAVGB	平均値の計算
_mm_avg_epu16	PAVGW	平均値の計算
_mm_madd_epi16	PMADDWD	乗算/加算
_mm_max_epi16	PMAXSW	最大値の計算
_mm_max_epu8	PMAXUB	最大値の計算
_mm_min_epi16	PMINSW	最小値の計算
_mm_min_epu8	PMINUB	最小値の計算
_mm_mulhi_epi16	PMULHW	乗算
_mm_mulhi_epu16	PMULHUW	乗算
_mm_mullo_epi16	PMULLW	乗算
_mm_mul_su32	PMULUDQ	乗算
_mm_mul_epu32	PMULUDQ	乗算
_mm_sad_epu8	PSADBW	差の計算/加算
_mm_sub_epi8	PSUBB	減算
_mm_sub_epi16	PSUBW	減算

_mm_sub_epi32	PSUBD	減算
_mm_sub_si64	PSUBQ	減算
_mm_sub_epi64	PSUBQ	減算
_mm_subs_epi8	PSUBSB	減算
_mm_subs_epi16	PSUBSW	減算
_mm_subs_epu8	PSUBUSB	減算
_mm_subs_epu16	PSUBUSW	減算

```
__mm128i _mm_add_epi8(__m128i a, __m128i b)
```

a の 16 の符号付きまたは符号なし 8 ビット整数を、b の 16 の符号付きまたは符号なし 8 ビット整数に加算します。

```
r0 := a0 + b0
r1 := a1 + b1
...
r15 := a15 + b15
```

```
__mm128i _mm_add_epi16(__m128i a, __m128i b)
```

a の 8 つの符号付きまたは符号なし 16 ビット整数を、b の 8 つの符号付きまたは符号なし 16 ビット整数に加算します。

```
r0 := a0 + b0
r1 := a1 + b1
...
r7 := a7 + b7
```

```
__m128i _mm_add_epi32(__m128i a, __m128i b)
```

a の 4 つの符号付きまたは符号なし 32 ビット整数を、b の 4 つの符号付きまたは符号なし 32 ビット整数に加算します。

```
r0 := a0 + b0
r1 := a1 + b1
r2 := a2 + b2
r3 := a3 + b3
```

```
__m64 _mm_add_si64(__m64 a, __m64 b)
```

a の符号付きまたは符号なし 64 ビット整数を、b の符号付きまたは符号なし 64 ビット整数に加算します。

```
r := a + b
```

```
__m128i _mm_add_epi64(__m128i a, __m128i b)
```

a の 2 つの符号付きまたは符号なし 64 ビット整数を、b の 2 つの符号付きまたは符号なし 64 ビット整数に加算します。

```
r0 := a0 + b0
r1 := a1 + b1
```

```
__m128i _mm_adds_epi8(__m128i a, __m128i b)
```

飽和演算を使用して、a の 16 の符号付き 8 ビット整数を、b の 16 の符号付き 8 ビット整数に加算します。

```
r0 := SignedSaturate(a0 + b0)
r1 := SignedSaturate(a1 + b1)
...
r15 := SignedSaturate(a15 + b15)
```

```
__m128i _mm_adds_epi16(__m128i a, __m128i b)
```

飽和演算を使用して、a の 8 つの符号付き 16 ビット整数を、b の 8 つの符号付き 16 ビット整数に加算します。

```
r0 := SignedSaturate(a0 + b0)
r1 := SignedSaturate(a1 + b1)
...
r7 := SignedSaturate(a7 + b7)
```

```
__m128i _mm_adds_epu8(__m128i a, __m128i b)
```

飽和演算を使用して、a の 16 の符号なし 8 ビット整数を、b の 16 の符号なし 8 ビット整数に加算します。

```
r0 := UnsignedSaturate(a0 + b0)
r1 := UnsignedSaturate(a1 + b1)
...
r15 := UnsignedSaturate(a15 + b15)
```

```
__m128i _mm_adds_epu16(__m128i a, __m128i b)
```

飽和演算を使用して、a の 8 つの符号なし 16 ビット整数を、b の 8 つの符号なし 16 ビット整数に加算します。

```
r0 := UnsignedSaturate(a0 + b0)
r1 := UnsignedSaturate(a1 + b1)
...
r15 := UnsignedSaturate(a7 + b7)
```

```
__m128i _mm_avg_epu8(__m128i a, __m128i b)
```

a の 16 の符号なし 8 ビット整数と b の 16 の符号なし 8 ビット整数について、対応する値の平均値を計算し、その結果を丸めます。

```
r0 := (a0 + b0) / 2
r1 := (a1 + b1) / 2
...
r15 := (a15 + b15) / 2
```

```
__m128i _mm_avg_epu16(__m128i a, __m128i b)
```

a の 8 つの符号なし 16 ビット整数と b の 8 つの符号なし 16 ビット整数について、対応する値の平均値を計算し、その結果を丸めます。

```
r0 := (a0 + b0) / 2
r1 := (a1 + b1) / 2
...
r7 := (a7 + b7) / 2
```

```
__m128i _mm_madd_epi16(__m128i a, __m128i b)
```

a の 8 つの符号付き 16 ビット整数に、b の 8 つの符号付き 16 ビット整数を掛けます。得られた符号付き 32 ビット整数を 2 つずつ加算して、4 つの符号付き 32 ビット整数としてパックします。

```
r0 := (a0 * b0) + (a1 * b1)
r1 := (a2 * b2) + (a3 * b3)
r2 := (a4 * b4) + (a5 * b5)
r3 := (a6 * b6) + (a7 * b7)
```

```
__m128i _mm_max_epi16(__m128i a, __m128i b)
```

a の 8 つの符号付き 16 ビット整数と b の 8 つの符号付き 16 ビット整数について、それぞれの値のペアの最大値を計算します。

```
r0 := max(a0, b0)
r1 := max(a1, b1)
...
r7 := max(a7, b7)
```

```
__m128i _mm_max_epu8(__m128i a, __m128i b)
```

a の 16 の符号なし 8 ビット整数と b の 16 の符号なし 8 ビット整数について、それぞれの値のペアの最大値を計算します。

```
r0 := max(a0, b0)
r1 := max(a1, b1)
...
r15 := max(a15, b15)
```

```
__m128i _mm_min_epi16(__m128i a, __m128i b)
```

a の 8 つの符号付き 16 ビット整数と b の 8 つの符号付き 16 ビット整数について、それぞれの値のペアの最小値を計算します。

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

```
__m128i _mm_min_epu8(__m128i a, __m128i b)
```

a の 16 の符号なし 8 ビット整数と b の 16 の符号なし 8 ビット整数について、それぞれの値のペアの最小値を計算します。

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r15 := min(a15, b15)
```

```
__m128i _mm_mulhi_epi16(__m128i a, __m128i b)
```

a の 8 つの符号付き 16 ビット整数に、b の 8 つの符号付き 16 ビット整数を掛けます。得られた 8 つの符号付き 32 ビット整数の上位 16 ビットをパックします。

```
r0 := (a0 * b0)[31:16]
r1 := (a1 * b1)[31:16]
...
r7 := (a7 * b7)[31:16]
```

```
__m128i _mm_mulhi_epu16(__m128i a, __m128i b)
```

a の 8 つの符号なし 16 ビット整数に、b の 8 つの符号なし 16 ビット整数を掛けます。得られた 8 つの符号なし 32 ビット整数の上位 16 ビットをパックします。

```
r0 := (a0 * b0)[31:16]
r1 := (a1 * b1)[31:16]
...
r7 := (a7 * b7)[31:16]
```

```
__m128i_mm_mullo_epi16(__m128i a, __m128i b)
```

a の 8 つの符号付きまたは符号なし 16 ビット整数に、b の 8 つの符号付きまたは符号なし 16 ビット整数を掛けます。得られた 8 つの符号付きまたは符号なし 32 ビット整数の下位 16 ビットをパックします。

```
r0 := (a0 * b0)[15:0]
r1 := (a1 * b1)[15:0]
...
r7 := (a7 * b7)[15:0]
```

```
__m64 _mm_mul_su32(__m64 a, __m64 b)
```

a の下位の 32 ビット整数に、b の下位の 32 ビット整数を掛けて、64 ビット整数の結果を返します。

```
r := a0 * b0
```

```
__m128i _mm_mul_epu32(__m128i a, __m128i b)
```

a の 2 つの符号なし 32 ビット整数に、b の 2 つの符号なし 32 ビット整数を掛けます。得られた 2 つの符号なし 64 ビット整数をパックしま

す。
```
r0 := a0 * b0
r1 := a2 * b2
```

```
__m128i _mm_sad_epu8(__m128i a, __m128i b)
```

a の 16 の符号なし 8 ビット整数と b の 16 の符号なし 8 ビット整数について、それぞれの差の絶対値を計算します。上位の 8 つの差と下位の 8 つの差をそれぞれに合計して、得られた 2 つの符号なし 16 ビット整数を、結果の上位および下位の 64 ビット要素の中にパックします。
```
r0 := abs(a0 - b0) + abs(a1 - b1) +...+ abs(a7 -
b7)
r1 := 0x0 ; r2 := 0x0 ; r3 := 0x0
r4 := abs(a8 - b8) + abs(a9 - b9) +...+ abs(a15 -
b15)
r5 := 0x0 ; r6 := 0x0 ; r7 := 0x0
```

```
__m128i _mm_sub_epi8(__m128i a, __m128i b)
```

a の 16 の符号付きまたは符号なし 8 ビット整数から、b の 16 の符号付きまたは符号なし 8 ビット整数を引きます。
```
r0 := a0 - b0
r1 := a1 - b1
...
r15 := a15 - b15
```

```
__m128i_mm_sub_epi16(__m128i a, __m128i b)
```

a の 8 つの符号付きまたは符号なし 16 ビット整数から、b の 8 つの符号付きまたは符号なし 16 ビット整数を引きます。
```
r0 := a0 - b0
r1 := a1 - b1
...
r7 := a7 - b7
```

```
__m128i _mm_sub_epi32(__m128i a, __m128i b)
```

a の 4 つの符号付きまたは符号なし 32 ビット整数から、b の 4 つの符号付きまたは符号なし 32 ビット整数を引きます。
```
r0 := a0 - b0
r1 := a1 - b1
r2 := a2 - b2
r3 := a3 - b3
```

```
__m64 _mm_sub_si64 (__m64 a, __m64 b)
```

a の符号付きまたは符号なし 64 ビット整数から、b の符号付きまたは符号なし 64 ビット整数を引きます。
```
r := a - b
```

```
__m128i _mm_sub_epi64(__m128i a, __m128i b)
```

a の 2 つの符号付きまたは符号なし 64 ビット整数から、b の 2 つの符号付きまたは符号なし 64 ビット整数を引きます。

```
r0 := a0 - b0
r1 := a1 - b1
```

```
__m128i _mm_subs_epi8(__m128i a, __m128i b)
```

飽和演算を使用して、a の 16 の符号付き 8 ビット整数から、b の 16 の符号付き 8 ビット整数を引きます。

```
r0 := SignedSaturate(a0 - b0)
r1 := SignedSaturate(a1 - b1)
...
r15 := SignedSaturate(a15 - b15)
```

```
__m128i _mm_subs_epi16(__m128i a, __m128i b)
```

飽和演算を使用して、a の 8 つの符号付き 16 ビット整数から、b の 8 つの符号付き 16 ビット整数を引きます。

```
r0 := SignedSaturate(a0 - b0)
r1 := SignedSaturate(a1 - b1)
...
r7 := SignedSaturate(a7 - b7)
```

```
__m128i _mm_subs_epu8(__m128i a, __m128i b)
```

飽和演算を使用して、a の 16 の符号なし 8 ビット整数から、b の 16 の符号なし 8 ビット整数を引きます。

```
r0 := UnsignedSaturate(a0 - b0)
r1 := UnsignedSaturate(a1 - b1)
...
r15 := UnsignedSaturate(a15 - b15)
```

```
__m128i _mm_subs_epu16(__m128i a, __m128i b)
```

飽和演算を使用して、a の 8 つの符号なし 16 ビット整数から、b の 8 つの符号なし 16 ビット整数を引きます。

```
r0 := UnsignedSaturate(a0 - b0)
r1 := UnsignedSaturate(a1 - b1)
...
r7 := UnsignedSaturate(a7 - b7)
```

ストリーミング SIMD 拡張命令 2 の整数論理演算

次の 4 つの論理演算組込み関数とそれに対応する命令は、ストリーミング SIMD 拡張命令 2 をサポートするプロセッサ上で有効です。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

```
__m128i _mm_and_si128(__m128i a, __m128i b)
```

(PAND を使用) a の 128 ビット値と b の 128 ビット値について、ビット単位の AND (論理積) を計算します。

```
r := a & b
```

```
__m128i _mm_andnot_si128(__m128i a, __m128i b)
```

(PANDN を使用) a の 128 ビット値のビット単位の NOT (否定) を実行し、その結果と b の 128 ビット値について、ビット単位の AND (論理積) を計算します。

```
r := (~a) & b
```

```
__m128i _mm_or_si128(__m128i a, __m128i b)
```

(POR を使用) a の 128 ビット値と b の 128 ビット値について、ビット単位の OR (論理和) を計算します。

```
r := a | b
```

```
__m128i _mm_xor_si128(__m128i a, __m128i b)
```

(PXOR を使用) a の 128 ビット値と b の 128 ビット値について、ビット単位の XOR (排他的論理和)を計算します。

```
r := a ^ b
```

ストリーミング SIMD 拡張命令 2 の整数シフト操作

ストリーミング SIMD 拡張命令 2 に対応したシフト演算組込み関数とその説明を下表に示します。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

組込み関数	シフトの方向	シフトの種類	対応する命令
_mm_slli_si128	左	論理	PSLLDQ
_mm_slli_epi16	左	論理	PSLLW
_mm_sll_epi16	左	論理	PSLLW
_mm_slli_epi32	左	論理	PSLLD
_mm_sll_epi32	左	論理	PSLLD

_mm_slli_epi64	左	論理	PSLLQ
_mm_sll_epi64	左	論理	PSLLQ
_mm_srai_epi16	右	算術	PSRAW
_mm_sra_epi16	右	算術	PSRAW
_mm_srai_epi32	右	算術	PSRAD
_mm_sra_epi32	右	算術	PSRAD
_mm_srli_si128	右	論理	PSRLDQ
_mm_srli_epi16	右	論理	PSRLW
_mm_srl_epi16	右	論理	PSRLW
_mm_srli_epi32	右	論理	PSRLD
_mm_srl_epi32	右	論理	PSRLD
_mm_srli_epi64	右	論理	PSRLQ
_mm_srl_epi64	右	論理	PSRLQ

```
__m128i _mm_slli_si128(__m128i a, int imm)
```

a の 128 ビット値を imm バイトだけ左にシフトし、下位ビットを 0 で埋めます。imm は即値でなければなりません。

```
r := a << (imm * 8)
```

```
__m128i _mm_slli_epi16(__m128i a, int count)
```

a に含まれている 8 つの符号付きまたは符号なし 16 ビット整数を左に count ビットだけシフトし、ゼロをシフトインします。

```
r0 := a0 << count
r1 := a1 << count
...
r7 := a7 << count
```

```
__m128i _mm_sll_epi16(__m128i a, __m128i count)
```

a に含まれている 8 つの符号付きまたは符号なし 16 ビット整数を左に count ビットだけシフトし、ゼロをシフトインします。

```
r0 := a0 << count
r1 := a1 << count
...
r7 := a7 << count
```

```
__m128i _mm_slli_epi32(__m128i a, int count)
```

a に含まれている 4 つの符号付きまたは符号なし 32 ビット整数を左に count ビットだけシフトし、ゼロをシフトインします。

```
r0 := a0 << count
r1 := a1 << count
r2 := a2 << count
r3 := a3 << count
```

```
__m128i _mm_sll_epi32(__m128i a, __m128i count)
```

a に含まれている 4 つの符号付きまたは符号なし 32 ビット整数を左に count ビットだけシフトし、ゼロをシフトインします。

```
r0 := a0 << count
r1 := a1 << count
r2 := a2 << count
r3 := a3 << count
```

```
__m128i _mm_slli_epi64(__m128i a, int count)
```

a に含まれている 2 つの符号付きまたは符号なし 64 ビット整数を左に count ビットだけシフトし、ゼロをシフトインします。

```
r0 := a0 << count
r1 := a1 << count
```

```
__m128i _mm_sll_epi64(__m128i a, __m128i count)
```

a に含まれている 2 つの符号付きまたは符号なし 64 ビット整数を左に count ビットだけシフトし、ゼロをシフトインします。

```
r0 := a0 << count
r1 := a1 << count
```

```
__m128i _mm_srai_epi16(__m128i a, int count)
```

a に含まれている 8 つの符号付き 16 ビット整数を右に count ビットだけシフトし、その符号ビットをシフトインします。

```
r0 := a0 >> count
r1 := a1 >> count
...
r7 := a7 >> count
```

```
__m128i _mm_sra_epi16(__m128i a, __m128i count)
```

a に含まれている 8 つの符号付き 16 ビット整数を右に count ビットだけシフトし、その符号ビットをシフトインします。

```
r0 := a0 >> count
r1 := a1 >> count
...
r7 := a7 >> count
```

```
__m128i _mm_srai_epi32(__m128i a, int count)
```

a に含まれている 4 つの符号付き 32 ビット整数を右に count ビットだけシフトし、その符号ビットをシフトインします。

```
r0 := a0 >> count
r1 := a1 >> count
r2 := a2 >> count
r3 := a3 >> count
```

```
__m128i _mm_sra_epi32(__m128i a, __m128i count)
```

a に含まれている 4 つの符号付き 32 ビット整数を右に count ビットだけシフトし、その符号ビットをシフトインします。

```
r0 := a0 >> count
r1 := a1 >> count
r2 := a2 >> count
r3 := i3 >> count
```

```
__m128i _mm_srli_si128(__m128i a, int imm)
```

a に含まれている 128 ビット値を右に imm バイトだけシフトし、ゼロをシフトインします。imm は即値でなければなりません。

```
r := srl(a, imm*8)
```

```
__m128i _mm_srli_epi16(__m128i a, int count)
```

a に含まれている 8 つの符号付きまたは符号なし 16 ビット整数を右に count ビットだけシフトし、ゼロをシフトインします。

```
r0 := srl(a0, count)
r1 := srl(a1, count)
...
r7 := srl(a7, count)
```

```
__m128i _mm_srl_epi16(__m128i a, __m128i count)
```

a に含まれている 8 つの符号付きまたは符号なし 16 ビット整数を右に count ビットだけシフトし、ゼロをシフトインします。

```
r0 := srl(a0, count)
r1 := srl(a1, count)
...
r7 := srl(a7, count)
```

```
__m128i _mm_srli_epi32(__m128i a, int count)
```

a に含まれている 4 つの符号付きまたは符号なし 32 ビット整数を右に count ビットだけシフトし、ゼロをシフトインします。

```
r0 := srl(a0, count)
r1 := srl(a1, count)
r2 := srl(a2, count)
r3 := srl(a3, count)
```

```
__m128i _mm_srl_epi32(__m128i a, __m128i count)
```

a に含まれている 4 つの符号付きまたは符号なし 32 ビット整数を右に count ビットだけシフトし、ゼロをシフトインします。

```
r0 := srl(a0, count)
r1 := srl(a1, count)
r2 := srl(a2, count)
r3 := srl(a3, count)
```

```
__m128i _mm_srli_epi64(__m128i a, int count)
```

a に含まれている 2 つの符号付きまたは符号なし 64 ビット整数を右に count ビットだけシフトし、ゼロをシフトインします。

```
r0 := srl(a0, count)
r1 := srl(a1, count)
```

```
__m128i _mm_srl_epi64(__m128i a, __m128i count)
```

a に含まれている 2 つの符号付きまたは符号なし 64 ビット整数を右に count ビットだけシフトし、ゼロをシフトインします。

```
r0 := srl(a0, count)
r1 := srl(a1, count)
```

ストリーミング SIMD 拡張命令 2 の整数比較操作

ストリーミング SIMD 拡張命令 2 に対応した比較演算組込み関数とその説明を下表に示します。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

組込み関数名	命令	比較条件	要素数	要素のサイズ
_mm_cmpeq_epi8	PCMPEQB	等しい	16	8
_mm_cmpeq_epi16	PCMPEQW	等しい	8	16
_mm_cmpeq_epi32	PCMPEQD	等しい	4	32
_mm_cmpgt_epi8	PCMPGTB	より大きい	16	8
_mm_cmpgt_epi16	PCMPGTW	より大きい	8	16
_mm_cmpgt_epi32	PCMPGTD	より大きい	4	32
_mm_cmplt_epi8	PCMPGTBr	より小さい	16	8
_mm_cmplt_epi16	PCMPGTWr	より小さい	8	16
_mm_cmplt_epi32	PCMPGTDr	より小さい	4	32

```
__m128i _mm_cmpeq_epi8(__m128i a, __m128i b)
```

a の 16 の符号付きまたは符号なし 8 ビット整数と、b の 16 の符号付きまたは符号なし 8 ビット整数が等しいかどうか比較します。

```
r0 := (a0 == b0) ? 0xff : 0x0
r1 := (a1 == b1) ? 0xff : 0x0
...
r15 := (a15 == b15) ? 0xff : 0x0
```

```
__m128i _mm_cmpeq_epi16(__m128i a, __m128i b)
```

a の 8 つの符号付きまたは符号なし 16 ビット整数と、b の 8 つの符号付きまたは符号なし 16 ビット整数が等しいかどうか比較します。

```
r0 := (a0 == b0) ? 0xffff : 0x0
r1 := (a1 == b1) ? 0xffff : 0x0
...
r7 := (a7 == b7) ? 0xffff : 0x0
```

```
__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)
```

a の 4 つの符号付きまたは符号なし 32 ビット整数と、b の 4 つの符号付きまたは符号なし 32 ビット整数が等しいかどうか比較します。

```
r0 := (a0 == b0) ? 0xffffffff : 0x0
r1 := (a1 == b1) ? 0xffffffff : 0x0
r2 := (a2 == b2) ? 0xffffffff : 0x0
r3 := (a3 == b3) ? 0xffffffff : 0x0
```

```
__m128i _mm_cmpgt_epi8(__m128i a, __m128i b)
```

a の 16 の符号付き 8 ビット整数が、b の 16 の符号付き 8 ビット整数より大きいかどうか比較します。

```
r0 := (a0 > b0) ? 0xff : 0x0
r1 := (a1 > b1) ? 0xff : 0x0
...
r15 := (a15 > b15) ? 0xff : 0x0
```

```
__m128i _mm_cmpgt_epi16(__m128i a, __m128i b)
```

a の 8 つの符号付き 16 ビット整数が、b の 8 つの符号付き 16 ビット整数より大きいかどうか比較します。

```
r0 := (a0 > b0) ? 0xffff : 0x0
r1 := (a1 > b1) ? 0xffff : 0x0
...
r7 := (a7 > b7) ? 0xffff : 0x0
```

```
__m128i _mm_cmpgt_epi32(__m128i a, __m128i b)
```

a の 4 つの符号付き 32 ビット整数が、b の 4 つの符号付き 32 ビット整数より大きいかどうか比較します。

```
r0 := (a0 > b0) ? 0xffff : 0x0
r1 := (a1 > b1) ? 0xffff : 0x0
r2 := (a2 > b2) ? 0xffff : 0x0
r3 := (a3 > b3) ? 0xffff : 0x0
```

```
__m128i _mm_cmplt_epi8( __m128i a, __m128i b)
```

a の 16 の符号付き 8 ビット整数が、b の 16 の符号付き 8 ビット整数より小さいかどうか比較します。

```
r0 := (a0 < b0) ? 0xff : 0x0
r1 := (a1 < b1) ? 0xff : 0x0
...
r15 := (a15 < b15) ? 0xff : 0x0
```

```
__m128i _mm_cmplt_epi16( __m128i a, __m128i b)
```

a の 8 つの符号付き 16 ビット整数が、b の 8 つの符号付き 16 ビット整数より小さいかどうか比較します。

```
r0 := (a0 < b0) ? 0xffff : 0x0
r1 := (a1 < b1) ? 0xffff : 0x0
...
r7 := (a7 < b7) ? 0xffff : 0x0
```

```
__m128i _mm_cmplt_epi32( __m128i a, __m128i b)
```

a の 4 つの符号付き 32 ビット整数が、b の 4 つの符号付き 32 ビット整数より小さいかどうか比較します。

```
r0 := (a0 < b0) ? 0xffff : 0x0
r1 := (a1 < b1) ? 0xffff : 0x0
r2 := (a2 < b2) ? 0xffff : 0x0
r3 := (a3 < b3) ? 0xffff : 0x0
```

ストリーミング SIMD 拡張命令 2 の変換操作

次の 2 つの変換組込み関数とそれに対応する命令は、ストリーミング SIMD 拡張命令 2 をサポートするプロセッサ上で有効です。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

```
__m128i _mm_cvtsi32_si128(int a)
```

(MOVD を使用) 32 ビット整数 a を __m128i オブジェクトの最下位 32 ビットに移動し、a の符号ビットを __m128i オブジェクトの上位 96

ビットにコピーします。
```
r0 := a
r1 := 0x0 ; r2 := 0x0 ; r3 := 0x0
```

```
int _mm_cvtsi128_si32(__m128i a)
```

(MOVD を使用) a の最下位 32 ビットを、32 ビット整数に移動します。
```
r := a0
```

```
__m128 _mm_cvtepi32_ps(__m128i a)
```

a の 4 つの符号付き 32 ビット整数値を単精度浮動小数点値に変換します。
```
r0 := (float) a0
r1 := (float) a1
r2 := (float) a2
r3 := (float) a3
```

```
__m128i _mm_cvtps_epi32(__m128 a)
```

a の 4 つの単精度浮動小数点値を符号付き 32 ビット整数値に変換します。
```
r0 := (int) a0
r1 := (int) a1
r2 := (int) a2
r3 := (int) a3
```

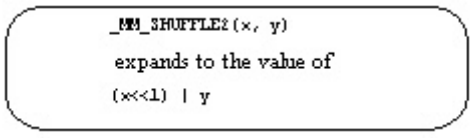
```
__m128i _mm_cvttps_epi32(__m128 a)
```

切り捨てを使用して、a の 4 つの単精度浮動小数点値を符号付き 32 ビット整数値に変換します。
```
r0 := (int) a0
r1 := (int) a1
r2 := (int) a2
r3 := (int) a3
```

シャッフルを行うマクロ関数

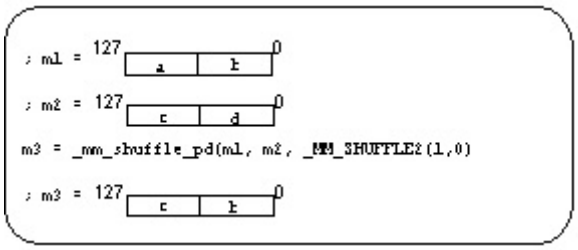
ストリーミング SIMD 拡張命令 2 には、シャッフル操作を記述する定数を生成するマクロ関数を用意しています。このマクロは、2 つの小さな整数 (0～1 の範囲) を組み合わせて、SHUFPD 命令が使用する 2 ビット即値を生成します。次の例を参照してください。

シャッフル関数のマクロ



2 つの整数は、第 1 入力オペランドと第 2 入力オペランドからそれぞれどの 2 ワードを取り出して結果のワードに入れるかを選択するセレクタとして機能します。

シャッフル関数のマクロの元のワードと結果のワード



ストリーミング SIMD 拡張命令 2 のキャッシュ操作

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

```
void _mm_stream_pd(double *p, __m128d a)
```

(MOVNTPD を使用) a のデータを、キャッシュを介さずに、アドレス p にストアします。アドレス p は、16 バイトにアライメントが合っていなければなりません。アドレス p を含むキャッシュ・ラインが既にキャッシュ内にある場合、キャッシュは更新されます。

```
p[0] := a0
p[1] := a1
```

```
void _mm_stream_si128(__m128i *p, __m128i a)
```

a のデータを、キャッシュを介さずに、アドレス p にストアします。アドレス p を含むキャッシュ・ラインが既にキャッシュ内にある場合、キャッシュは更新されます。アドレス p は、16 バイトにアライメントが合っていなければなりません。

```
*p := a
```

```
void _mm_stream_si32(int *p, int a)
```

a のデータを、キャッシュを介さずに、アドレス p にストアします。アドレス p を含むキャッシュ・ラインが既にキャッシュ内にある場合、キャッシュは更新されます。
*p := a

void _mm_clflush(void const*p)

コヒーレンシ・ドメイン内のすべてのキャッシュから、p を含むキャッシュ・ラインをフラッシュし、無効化します。

void _mm_lfence(void)

プログラムの順序でロードフェンス命令に先行するすべてのロード命令が、フェンスに続くロード命令より前に、グローバルにアクセス可能になるのを保証します。

void _mm_mfence(void)

プログラムの順序でメモリフェンス命令に先行するすべてのメモリアクセス命令が、フェンスに続くメモリアクセス命令より前に、グローバルにアクセス可能になるのを保証します。

void _mm_pause(void)

プロセッサに固有の時間の間、次の命令の実行を遅らせます。この命令を実行しても、アーキテクチャ上の状態は変化しません。この組込み関数を使用すると、パフォーマンスが大きく向上します。この組込み関数については、以降で詳しく説明します。

PAUSE 組込み関数

PAUSE 組込み関数は、ダイナミック・エグゼキューション (特に、アウトオブオーダー実行) をサポートするプロセッサ上で、spin-wait ループに使用します。spin-wait ループ内で PAUSE を使用すると、ロックの解放を検出するコードの処理速度が向上します。動的スケジューリングに PAUSE 命令を使用すると、スピンループの終了時のペナルティが軽減されます。

PAUSE 命令を使用したループの例

```
spin_loop:pause
cmp eax, A
jne spin_loop
```

上の例では、メモリ・ロケーション A がレジスタ eax の値と一致するまで、プログラムはスピンします。次のコード・シーケンスは、test-and-test-and-set 操作を示しています。この例では、ロックの取得に失敗した場合にのみ、スピンが発生します。

```
get lock: mov eax, 1
xchg eax, A ; Try to get lock
cmp eax, 0 ; Test if successful
jne spin loop
critical section code
mov A, 0 ; Release lock
jmp continue
spin loop: pause ; Spin-loop hint
cmp 0, A ; Check lock availability
jne spin loop
jmp get lock
continue:
```

この例では、ロックの取得に成功すると予測して、最初の条件分岐は分岐せずに、そのままクリティカル・セクションの処理に移ります。すべての spin-wait ループに、PAUSE 命令を使用することを強くお勧めします。PAUSE は、既存のすべての IA-32 プロセッサで使用可能なため、プロセッサのタイプをテストする (CPUID テスト) 必要はありません。すべての従来のプロセッサは、PAUSE を NOP として実行しますが、PAUSE をヒントとして使用するプロセッサでは、パフォーマンスが大きく向上する可能性があります。

ストリーミング SIMD 拡張命令 2 のその他の操作

次の表に、ストリーミング SIMD 拡張命令 2 のその他の組込み関数のリストを示します。表の後に、各組込み関数の説明を示します。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

組込み関数	対応する命令	操作
_mm_packs_epi16	PACKSSWB	パックド飽和
_mm_packs_epi32	PACKSSDW	パックド飽和
_mm_packus_epi16	PACKUSWB	パックド飽和
_mm_extract_epi16	PEXTRW	抽出
_mm_insert_epi16	PINSRW	挿入
_mm_movemask_epi8	PMOVMSKB	マスクの作成
_mm_shuffle_epi32	PSHUFD	シャッフル
_mm_shufflehi_epi16	PSHUFHW	シャッフル

_mm_shufflelo_epi16	PSHUFLW	シャッフル
_mm_unpackhi_epi8	PUNPCKHBW	インターリーブ
_mm_unpackhi_epi16	PUNPCKHWD	インターリーブ
_mm_unpackhi_epi32	PUNPCKHDQ	インターリーブ
_mm_unpackhi_epi64	PUNPCKHQDQ	インターリーブ
_mm_unpacklo_epi8	PUNPCKLBW	インターリーブ
_mm_unpacklo_epi16	PUNPCKLWD	インターリーブ
_mm_unpacklo_epi32	PUNPCKLDQ	インターリーブ
_mm_unpacklo_epi64	PUNPCKLQDQ	インターリーブ
_mm_movepi64_pi64	MOVDQ2Q	移動
_m128i_mm_movpi64_epi64	MOVQ2DQ	移動
_mm_move_epi64	MOVQ	移動

```
__m128i _mm_packs_epi16(__m128i a, __m128i b)
```

a と b の 16 の符号付き 16 ビット整数を 8 ビット整数にパックして、飽和処理します。

```
r0 := SignedSaturate(a0)
r1 := SignedSaturate(a1)
...
r7 := SignedSaturate(a7)
r8 := SignedSaturate(b0)
r9 := SignedSaturate(b1)
...
r15 := SignedSaturate(b7)
```

```
__m128i _mm_packs_epi32(__m128i a, __m128i b)
```

a と b の 8 つの符号付き 32 ビット整数を符号付き 16 ビット整数にパックして、飽和処理します。

```
r0 := SignedSaturate(a0)
r1 := SignedSaturate(a1)
r2 := SignedSaturate(a2)
r3 := SignedSaturate(a3)
r4 := SignedSaturate(b0)
r5 := SignedSaturate(b1)
r6 := SignedSaturate(b2)
r7 := SignedSaturate(b3)
```

```
__m128i _mm_packus_epi16(__m128i a, __m128i b)
```

a と b の 16 の符号付き 16 ビット整数を符号なし 8 ビット整数にパックして、飽和処理します。

```
r0 := UnsignedSaturate(a0)
r1 := UnsignedSaturate(a1)
...
r7 := UnsignedSaturate(a7)
r8 := UnsignedSaturate(b0)
r9 := UnsignedSaturate(b1)
...
r15 := UnsignedSaturate(b7)
```

```
int _mm_extract_epi16(__m128i a, int imm)
```

選択された符号付きまたは符号なし 16 ビット整数を a から抽出して、0 で拡張します。セレクタ imm は、即値でなければなりません。

```
r := (imm == 0) ? a0 :
( (imm == 1) ? a1 :
...
(imm == 7) ? a7 )
```

```
__m128i _mm_insert_epi16(__m128i a, int b, int imm)
```

b の最下位 16 ビットを、a の選択された 16 ビット整数に挿入します。セレクタ imm は、即値でなければなりません。

```
r0 := (imm == 0) ? b : a0;
r1 := (imm == 1) ? b : a1;
...
r7 := (imm == 7) ? b : a7;
```

```
int _mm_movemask_epi8(__m128i a)
```

a の 16 の符号付きまたは符号なし 8 ビット整数の最上位ビットを使用して 16 ビットマスクを作成し、上位ビットを 0 で拡張します。

```
r := a15[7] << 15 |
a14[7] << 14 |
...
a1[7] << 1 |
a0[7]
```

```
__m128i _mm_shuffle_epi32(__m128i a, int imm)
```

imm の指定に従って、a の 4 つの符号付きまたは符号なし 32 ビット整数をシャッフルします。シャッフル値 imm は、即値でなければなりません。シャッフルのセマンティクスについては、このセクションの最後の「シャッフルを行うマクロ関数」を参照してください。

```
__m128i _mm_shufflehi_epi16(__m128i a, int imm)
```

imm の指定に従って、a の上位 4 つの符号付きまたは符号なし 16 ビット整数をシャッフルします。シャッフル値 imm は、即値でなければ

なりません。シャッフルのセマンティクスについては、このセクションの最後の「シャッフルを行うマクロ関数」を参照してください。

```
__m128i _mm_shufflelo_epi16(__m128i a, int imm)
```

imm の指定に従って、a の下位 4 つの符号付きまたは符号なし 16 ビット整数をシャッフルします。シャッフル値 imm は、即値でなければなりません。シャッフルのセマンティクスについては、このセクションの最後の「シャッフルを行うマクロ関数」を参照してください。

```
__m128i _mm_unpackhi_epi8(__m128i a, __m128i b)
```

a の上位 8 つの符号付きまたは符号なし 8 ビット整数と、b の上位 8 つの符号付きまたは符号なし 8 ビット整数をインターリーブ (交互に配置) します。

```
r0 := a8 ; r1 := b8
r2 := a9 ; r3 := b9
...
r14 := a15 ; r15 := b15
```

```
__m128i _mm_unpackhi_epi16(__m128i a, __m128i b)
```

a の上位 4 つの符号付きまたは符号なし 16 ビット整数と、b の上位 4 つの符号付きまたは符号なし 16 ビット整数をインターリーブします。

```
r0 := a4 ; r1 := b4
r2 := a5 ; r3 := b5
r4 := a6 ; r5 := b6
r6 := a7 ; r7 := b7
```

```
__m128i _mm_unpackhi_epi32(__m128i a, __m128i b)
```

a の上位 2 つの符号付きまたは符号なし 32 ビット整数と、b の上位 2 つの符号付きまたは符号なし 32 ビット整数をインターリーブします。

```
r0 := a2 ; r1 := b2
r2 := a3 ; r3 := b3
```

```
__m128i _mm_unpackhi_epi64(__m128i a, __m128i b)
```

a の上位の符号付きまたは符号なし 64 ビット整数と、b の上位の符号付きまたは符号なし 64 ビット整数をインターリーブします。

```
r0 := a1 ; r1 := b1
```

```
__m128i _mm_unpacklo_epi8(__m128i a, __m128i b)
```

a の下位 8 つの符号付きまたは符号なし 8 ビット整数と、b の下位 8 つの符号付きまたは符号なし 8 ビット整数をインターリーブします。

```
r0 := a0 ; r1 := b0
```

```
r2 := a1 ; r3 := b1
...
r14 := a7 ; r15 := b7
```

```
__m128i _mm_unpacklo_epi16(__m128i a, __m128i b)
```

a の下位 4 つの符号付きまたは符号なし 16 ビット整数と、b の下位 4 つの符号付きまたは符号なし 16 ビット整数をインターリーブします。

```
r0 := a0 ; r1 := b0
r2 := a1 ; r3 := b1
r4 := a2 ; r5 := b2
r6 := a3 ; r7 := b3
```

```
__m128i _mm_unpacklo_epi32(__m128i a, __m128i b)
```

a の下位 2 つの符号付きまたは符号なし 32 ビット整数と、b の下位 2 つの符号付きまたは符号なし 32 ビット整数をインターリーブします。

```
r0 := a0 ; r1 := b0
r2 := a1 ; r3 := b1
```

```
__m128i _mm_unpacklo_epi64(__m128i a, __m128i b)
```

a の下位の符号付きまたは符号なし 64 ビット整数と、b の下位の符号付きまたは符号なし 64 ビット整数をインターリーブします。

```
r0 := a0 ; r1 := b0
```

```
__m64 _mm_movepi64_pi64(__m128i a)
```

a の下位 64 ビットを、__m64 型として返します。

```
r0 := a0 ;
```

```
__128i _mm_movpi64_pi64(__m64 a)
```

a の 64 ビットを結果の下位 64 ビットに移動し、上位ビットを 0 に設定します。

```
r0 := a0 ; r1 := 0X0 ;
```

```
__128i _mm_move_epi64(__128i a)
```

a の下位 64 ビットを結果の下位 64 ビットに移動し、上位ビットを 0 に設定します。

```
r0 := a0 ; r1 := 0X0 ;
```

ストリーミング SIMD 拡張命令 2 の整数メモリ操作と初期化操作

整数のロード、設定、およびストア組込み関数とそれに対応する命令によって、ストリーミング SIMD 拡張命令 2 のメモリ操作と初期化操作を実行できます。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

- ロード操作
- 設定操作
- ストア操作

ストリーミング SIMD 拡張命令 2 の整数ロード演算

次のロード操作組込み関数とそれに対応する命令は、ストリーミング SIMD 拡張命令 2 をサポートするプロセッサ上で有効です。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

```
__m128i _mm_load_si128(__m128i const*p)
```

(MOVDQA を使用) 128 ビット値をロードします。アドレス p は、16 バイトにアライメントが合っていなければなりません。

```
r := *p
```

```
__m128i _mm_loadu_si128(__m128i const*p)
```

(MOVDQU を使用) 128 ビット値をロードします。アドレス p は、16 バイトにアライメントが合っていなくてもかまいません。

```
r := *p
```

```
__m128i _mm_loadl_epi64(__m128i const*p)
```

(MOVQ を使用) p で指定された値の下位 64 ビットを結果の下位 64 ビットにロードし、結果の上位 64 ビットは 0 に設定します。

```
r0:= *p[63:0]
r1:=0x0
```

ストリーミング SIMD 拡張命令 2 の整数設定操作

次の設定操作組込み関数とそれに対応する命令は、ストリーミング SIMD 拡張命令 2 をサポートするプロセッサ上で有効です。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

```
__m128i _mm_set_epi64(__m64 q1, __m64 q0)
```

2 つの 64 ビット整数値を設定します。

```
r0 := q0
r1 := q1
```

```
__m128i _mm_set_epi32(int i3, int i2, int i1, int i0)
```

4 つの符号付き 32 ビット整数値を設定します。

```
r0 := i0
r1 := i1
r2 := i2
r3 := i3
```

```
__m128i _mm_set_epi16(short w7, short w6, short w5, short
w4, short w3, short w2, short w1, short w0)
```

8 つの符号付き 16 ビット整数値を設定します。

```
r0 := w0
r1 := w1
...
r7 := w7
```

```
__m128i _mm_set_epi8(char b15, char b14, char b13, char b12,
char b11, char b10, char b9, char b8, char b7, char b6,
char b5, char b4, char b3, char b2, char b1, char b0)
```

16 の符号付き 8 ビット整数値を設定します。

```
r0 := b0
r1 := b1
...
r15 := b15
```

```
__m128i _mm_set1_epi64(__m64 q)
```

2 つの 64 ビット整数値を q に設定します。

```
r0 := q
r1 := q
```

```
__m128i _mm_set1_epi32(int i)
```

4 つの符号付き 32 ビット整数値を i に設定します。

```
r0 := i
r1 := i
r2 := i
r3 := i
```

```
__m128i _mm_set1_epi16(short w)
```

8 つの符号付き 16 ビット整数値を w に設定します。

```
r0 := w
r1 := w
...
r7 := w
```

```
__m128i _mm_set1_epi8(char b)
```

16 の符号付き 8 ビット整数値を b に設定します。

```
r0 := b
r1 := b
...
r15 := b
```

```
__m128i _mm_setr_epi64(__m64 q0, __m64 q1)
```

2 つの 64 ビット整数値を逆順で設定します。

```
r0 := q0
r1 := q1
```

```
__m128i _mm_setr_epi32(int i0, int i1, int i2, int i3)
```

4 つの符号付き 32 ビット整数値を逆順で設定します。

```
r0 := i0
r1 := i1
r2 := i2
r3 := i3
```

```
__m128i _mm_setr_epi16(short w0, short w1, short w2, short
w3, short w4, short w5, short w6, short w7)
```

8 つの符号付き 16 ビット整数値を逆順で設定します。

```
r0 := w0
r1 := w1
...
r7 := w7
```

```
__m128i _mm_setr_epi8(char b15, char b14, char b13, char
b12, char b11, char b10, char b9, char b8, char b7, char b6,
char b5, char b4, char b3, char b2, char b1, char b0)
```

16 の符号付き 8 ビット整数値を逆順で設定します。

```
r0 := b0
r1 := b1
...
r15 := b15
```

```
__m128i _mm_setzero_si128()
```

128 ビット値を 0 に設定します。
```
r := 0x0
```

ストリーミング SIMD 拡張命令 2 の整数ストア操作

次のストア操作組込み関数とそれに対応する命令は、ストリーミング SIMD 拡張命令 2 をサポートするプロセッサ上で有効です。

ストリーミング SIMD 拡張命令 2 の組込み関数のプロトタイプは、ヘッダファイル emmintrin.h 内にあります。

```
void _mm_store_si128(__m128i *p, __m128i b)
```

(MOVDQA を使用) 128 ビット値をストアします。アドレス p は、16 バイトにアライメントが合っていなければなりません。
```
*p := a
```

```
void _mm_storeu_si128(__m128i *p, __m128i b)
```

(MOVDQU を使用) 128 ビット値をストアします。アドレス p は、16 バイトにアライメントが合っていなくてもかまいません。
```
*p := a
```

```
void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p)
```

(MASKMOVDQU を使用) d のバイト要素を、条件付きでアドレス p にストアします。セレクタ n の各バイトの最上位ビットによって、それに対応する d の各バイトがストアされるかどうかが決まります。アドレス p は、16 バイトにアライメントが合っていなくてもかまいません。
```
if (n0[7]) p[0] := d0
if (n1[7]) p[1] := d1
...
if (n15[7]) p[15] := d15
```

```
void _mm_storel_epi64(__m128i *p, __m128i q)
```

(MOVQ を使用) p で指定された値の下位 64 ビットをストアします。
```
*p[63:0]:=a0
```

新しい IA-32 組込み関数

概要: 新しい IA-32 組込み関数

このセクションで説明するインテル® C++ 組込み関数は、HT テクノロジ インテル® Pentium® 4 プロセッサ (SSE3 対応) 用に設計されたものです。これらの組込み関数は他の IA-32 プロセッサ上では正しく機能しません。

- マクロ関数
- 浮動小数点ベクトル組込み関数
- 整数ベクトル組込み関数
- その他の組込み関数

マクロ関数

次のマクロ組込み関数は、HT テクノロジ インテル® Pentium® 4 プロセッサ (SSE3 対応) 用に設計されたものです。

`_MM_SET_DENORMALS_ZERO_MODE(x)`

> マクロ引数: `__MM_DENORMALS_ZERO_ON`、`_MM_DENORMALS_ZERO_OFF` のいずれか。
> 制御レジスタの適切なビットを設定して、"DAZ (`denormals are zeros`)" モードをオンまたはオフにします。

`_MM_GET_DENORMALS_ZERO_MODE()`

> 引数なし。制御レジスタの DAZ (denormals are zeros) モードビットの現在の値を返します。

浮動小数点ベクトル組込み関数

次の浮動小数点ベクトル組込み関数は、HT テクノロジ インテル® Pentium® 4 プロセッサ (SSE3 対応) 用に設計されたものです。

単精度浮動小数点ベクトル組込み関数

```
extern __m128 _mm_addsub_ps(__m128 a, __m128 b);
```

奇数ベクトル要素を加え、偶数ベクトル要素を引きます。

```
r0 := a0 - b0;
r1 := a1 + b1;
r2 := a2 - b2;
r3 := a3 + b3;
```

```
extern __m128 _mm_hadd_ps(__m128 a, __m128 b);
```

隣接したベクトル要素を加えます。

```
r0 := a0 + a1;
r1 := a2 + a3;
r2 := b0 + b1;
r3 := b2 + b3;
```

```
extern __m128 _mm_hsub_ps(__m128 a, __m128 b);
```

隣接したベクトル要素を引きます。

```
r0 := a0 - a1;
r1 := a2 - a3;
r2 := b0 - b1;
r3 := b2 - b3;
```

```
extern __m128 _mm_movehdup_ps(__m128 a);
```

奇数ベクトル要素を偶数ベクトル要素へ複製します。

```
r0 := a1;
r1 := a1;
r2 := a3;
r3 := a3;
```

```
extern __m128 _mm_moveldup_ps(__m128 a);
```

偶数ベクトル要素を奇数ベクトル要素へ複製します。

```
r0 := a0;
r1 := a0;
r2 := a2;
r3 := a2;
```

倍精度浮動小数点ベクトル組込み関数

```
extern __m128d _mm_addsub_pd(__m128d a, __m128d b);
```

下位ベクトル要素を引き、上位ベクトル要素を加えます。

```
r0 := a0 - b0;
r1 := a1 + b1;
```

```
extern __m128d _mm_hadd_pd(__m128d a, __m128d b);
```

隣接したベクトル要素を加えます。
```
r0 := a0 + a1;
r1 := b0 + b1;
```

```
extern __m128d _mm_hsub_pd(__m128d a, __m128d b);
```

隣接したベクトル要素を引きます。
```
r0 := a0 - a1;
r1 := b0 - b1;
```

```
extern __m128d _mm_loaddup_pd(double const * dp);
```

double の値を上位と下位ベクトル要素へ複製します。
```
r0 := *dp;
r1 := *dp;
```

```
extern __m128d _mm_movedup_pd(__m128d a);
```

下位ベクトル要素を上位ベクトル要素へ複製します。
```
r0 := a0;
r1 := a0;
```

整数ベクトル組込み関数

次の整数ベクトル組込み関数は、HT テクノロジ インテル® Pentium® 4 プロセッサ (SSE3 対応) 用に設計されたものです。

```
extern __m128i _mm_lddqu_si128(__m128i const *p);
```

128 ビットの値をロードします (アライメントが合っている必要はありません)。この組込み関数は、ほとんどの場合 `movdqu` よりもパフォーマンスが向上します。しかし、書き込まれた直後のメモリ値を読み取る場合は、`movdqu` よりもパフォーマンスが低下します。
```
r := *p;
```

その他の組込み関数

次のその他の組込み関数は、HT テクノロジ インテル® Pentium® 4 プロセッサ (SSE3 対応) 用に設計されたものです。

```
extern void _mm_monitor(void const *p, unsigned extensions,
unsigned hints);
```

MONITOR 命令を生成します。論理アドレスを得るために p を使用してモニタ・ハードウェア用のアドレス範囲をセットアップし、レジスタ eax でモニタ命令に渡します。拡張パラメータは、ecx で渡されるモニタ・ハードウェアへの拡張オプションを含みます。ヒント・パラメータは、edx で渡されるモニタ・ハードウェアへのヒントを含みます。拡張でゼロではない値が指定されると、一般的な保護違反が発生します。

```
extern void _mm_mwait(unsigned extensions, unsigned hints);
```

MWAIT 命令を生成します。この命令は、プロセッサが、実行を停止して、イベントのクラスが発生するまでインプリメンテーション依存の最適化された状態に入ることができるようにするヒントです。将来のプロセッサ設計では、拡張およびヒント・パラメータはプロセッサに追加情報を伝えるために使用されます。拡張およびヒントのゼロではない値はすべて予約されています。拡張でゼロではない値が指定されると、一般的な保護違反が発生します。

Itanium® 命令の組込み関数

概要: Itanium® 命令の組込み関数

このセクションでは、Itanium® 命令のネイティブ組込み関数を一覧表示し説明します。これらの組込み関数は、IA-32 アーキテクチャ上では使用できません。Itanium 命令の組込み関数によって、プログラマは、C および C++ 言語の標準的な構文では生成できない Itanium 命令を利用できます。

Itanium アーキテクチャ用の組込み関数のプロトタイプは、ヘッダファイル ia64intrin.h 内にあります。

Itanium® 命令のネイティブ組込み関数

Itanium® アーキテクチャ用の組込み関数のプロトタイプは、ヘッダファイル ia64intrin.h 内にあります。

整数演算

組込み関数	対応する命令
__int64 _m64_dep_mr(__int64 r, __int64 s, const int pos, const int len)	dep (Deposit)

`__int64 _m64_dep_mi(const int v, __int64 s, const int p, const int len)`	`dep` (Deposit)
`__int64 _m64_dep_zr(__int64 s, const int pos, const int len)`	`dep.z` (Deposit)
`__int64 _m64_dep_zi(const int v, const int pos, const int len)`	`dep.z` (Deposit)
`__int64 _m64_extr(__int64 r, const int pos, const int len)`	`extr` (Extract)
`__int64 _m64_extru(__int64 r, const int pos, const int len)`	`extr.u` (Extract)
`__int64 _m64_xmal(__int64 a, __int64 b, __int64 c)`	`xma.l` (Fixed-point multiply add。128 ビットの下位 64 ビットを使用します。結果は符号付きです。)
`__int64 _m64_xmalu(__int64 a, __int64 b, __int64 c)`	`xma.lu` (Fixed-point multiply add。128 ビットの下位 64 ビットを使用します。結果は符号なしです。)
`__int64 _m64_xmah(__int64 a, __int64 b, __int64 c)`	`xma.h` (Fixed-point multiply add。128 ビットの上位 64 ビットを使用します。結果は符号付きです。)
`__int64 _m64_xmahu(__int64 a, __int64 b, __int64 c)`	`xma.hu` (Fixed-point multiply add。128 ビットの上位 64 ビットを使用します。結果は符号なしです。)
`__int64 _m64_popcnt(__int64 a)`	`popcnt` (Population count)
`__int64 _m64_shladd(__int64 a, const int count, __int64 b)`	`shladd` (Shift left and add)
`__int64 _m64_shrp(__int64 a, __int64 b, const int count)`	`shrp` (Shift right pair)

FSR 演算

組込み関数	説明
`void _fsetc(int amask, int omask)`	`FPSR.sf0` の制御ビットをセットします。`fsetc.sf0 r, r` 命令に対応付けます。これに対応する、制御ビットの読み取り命令はありません。`_mm_getfpsr()` を使用してください。
`void _fclrf(void)`	浮動小数点ステータス・フラグ (`FPSR.sf0` の 6 ビットフラグ) をクリアします。`fclrf.sf0` 命令に対応付けます。

`__int64 _m64_dep_mr(__int64 r, __int64 s, const int pos, const int len)`

右揃えした 64 ビット値 `r` を、`s` の値の中の任意のビット位置にデポジットし、その結果を返します。デポジットしたビット・フィールドは、ビッ

ト位置 pos を始点として、len で指定したビット数だけ左に (最上位ビットの方向に) 延長します。

```
__int64 _m64_dep_mi(const int v, __int64 s, const int p,
const int len)
```

符号で拡張した値 v (すべて 1 またはすべて 0)を、s の値の中の任意のビット位置にデポジットし、その結果を返します。デポジットしたビット・フィールドは、ビット位置 p を始点として、len で指定したビット数だけ左に (最上位ビットの方向に) 延長します。

```
__int64 _m64_dep_zr(__int64 s, const int pos, const int
len)
```

右揃えした 64 ビット値 s を、すべて 0 の 64 ビット・フィールド内の任意のビット位置にデポジットし、その結果を返します。デポジットしたビット・フィールドは、ビット位置 pos を始点として、len で指定したビット数だけ左に (最上位ビットの方向に) 延長します。

```
__int64 _m64_dep_zi(const int v, const int pos, const int
len)
```

符号で拡張した値 v (すべて 1 またはすべて 0)を、すべて 0 の 64 ビット・フィールド内の任意のビット位置にデポジットし、その結果を返します。デポジットしたビット・フィールドは、ビット位置 pos を始点として、len で指定したビット数だけ左に (最上位ビットの方向に) 延長します。

```
__int64 _m64_extr(__int64 r, const int pos, const int len)
```

64 ビット値 r から 1 つのフィールドを抽出し、右揃えにして符号で拡張した値を返します。抽出したフィールドは、pos の位置を始点として、len ビットだけ左に延長します。抽出したフィールドの最上位ビットの符号が使用されます。

```
__int64 _m64_extru(__int64 r, const int pos, const int len)
```

64 ビット値 r から 1 つのフィールドを抽出し、右揃えにして 0 で拡張した値を返します。抽出したフィールドは、pos の位置を始点として、len ビットだけ左に延長します。

```
__int64 _m64_xmal(__int64 a, __int64 b, __int64 c)
```

64 ビット値 a と b を符号付き整数と見なして乗算し、全 128 ビットの符号付きの結果を求めます。64 ビット値 c を 0 で拡張してこの積に加算し、得られた和の最下位 64 ビットを返します。

`__int64 _m64_xmalu(__int64 a, __int64 b, __int64 c)`

64 ビット値 `a` と `b` を符号付き整数と見なして乗算し、全 128 ビットの符号なしの結果を求めます。64 ビット値 `c` を 0 で拡張してこの積に加算し、得られた和の最下位 64 ビットを返します。

`__int64 _m64_xmah(__int64 a, __int64 b, __int64 c)`

64 ビット値 `a` と `b` を符号付き整数と見なして乗算し、全 128 ビットの符号付きの結果を求めます。64 ビット値 `c` を 0 で拡張してこの積に加算し、得られた和の最上位 64 ビットを返します。

`__int64 _m64_xmahu(__int64 a, __int64 b, __int64 c)`

64 ビット値 `a` と `b` を符号なし整数と見なして乗算し、全 128 ビットの符号なしの結果を求めます。64 ビット値 `c` を 0 で拡張してこの積に加算し、得られた和の最上位 64 ビットを返します。

`__int64 _m64_popcnt(__int64 a)`

64 ビット整数 `a` の中のビットのうち、値が 1 のビットをカウントし、得られたビット数を返します。

`__int64 _m64_shladd(__int64 a, const int count, __int64 b)`

`a` を `count` ビットだけ左にシフトして、`b` に加算します。結果を返します。

`__int64 _m64_shrp(__int64 a, __int64 b, const int count)`

`a` と `b` を連結して 128 ビット値を作成し、`count` ビットだけ右にシフトします。結果の最下位 64 ビットを返します。

ロックおよびアトミック操作に関連する組込み関数

これらの組込み関数のプロトタイプは、ヘッダファイル `ia64intrin.h` 内にあります。

組込み関数	説明
`unsigned __int64 _InterlockedExchange8(volatile unsigned char *Target, unsigned __int64 value)`	`xchg1` 命令に対応付けます。第 1 引数で指定されたアドレスに第 2 引数の最下位のバイトをアトミックに書きます。
`unsigned __int64`	第 1 引数で指定されたアドレスの

_InterlockedCompareExchange8_rel(volatile unsigned char *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	最下位のバイトをアトミックに比較/交換します。適切な設定の cmpxchg1.rel 命令に対応付けます。
unsigned __int64 _InterlockedCompareExchange8_acq(volatile unsigned char *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	上記と同じですが、acquire セマンティックを使用します。
unsigned __int64 _InterlockedExchange16(volatile unsigned short *Target, unsigned __int64 value)	xchg2 命令に対応付けます。第 1 引数で指定されたアドレスに第 2 引数の最下位のワードをアトミックに書きます。
unsigned __int64 _InterlockedCompareExchange16_rel(volatile unsigned short *Destination, unsigned int64 Exchange, unsigned __int64 Comparand)	第 1 引数で指定されたアドレスの最下位のワードをアトミックに比較/交換します。適切な設定の cmpxchg2.rel 命令に対応付けます。
unsigned __int64 _InterlockedCompareExchange16_acq(volatile unsigned short *Destination, unsigned int64 Exchange, unsigned __int64 Comparand)	上記と同じですが、acquire セマンティックを使用します。
int _InterlockedIncrement(volatile int *addend	引数で指定された値を 1 ずつアトミックに増分します。fetchadd4 命令に対応付けます。
int _InterlockedDecrement(volatile int *addend	引数で指定された値を 1 ずつアトミックに減分します。fetchadd4 命令に対応付けます。
int _InterlockedExchange(volatile int *Target, long value	交換操作をアトミックに実行します。xchg4 命令に対応付けます。
int _InterlockedCompareExchange(volatile int *Destination, int Exchange, int Comparand	比較/交換操作をアトミックに実行します。適切な設定の cmpxchg4 命令に対応付けます。
int _InterlockedExchangeAdd(volatile int *addend, int increment	比較/交換操作を使用して、加数に対して増分値をアトミックに加算します。cmpxchg4 命令を使用するループに対応付けられ、アトミックな性質が保証されます。
int InterlockedAdd(volatile int *addend, int increment)	上記と同じですが、元の値ではなく、新しい値を返します。
void * _InterlockedCompareExchangePointer(void * volatile *Destination, void *Exchange, void *Comparand)	exch8 命令を対応付けます。第 1 引数 (すべての引数はポインタ) で指定されたポインタ値をアトミックに比較/交換します。
unsigned __int64 _InterlockedExchangeU(volatile unsigned int *Target, unsigned __int64 value)	第 1 引数で指定された 32 ビットデータをアトミックに交換します。

	xchg4 命令に対応付けます。
unsigned __int64 _InterlockedCompareExchange_rel(volatile unsigned int *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	適切な設定の cmpxchg4.rel 命令に対応付けます。第 1 引数 (64 ビットポインタ) で指定された値をアトミックに比較/交換します。
unsigned __int64 _InterlockedCompareExchange_acq(volatile unsigned int *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	上記と同じですが、cmpxchg4.acq 命令を対応付けます。
void _ReleaseSpinLock(volatile int *x)	スピンロックを解放します。
__int64 _InterlockedIncrement64(volatile __int64 *addend)	引数で指定された値を 1 ずつ増分します。fetchadd 命令に対応付けます。
__int64 _InterlockedDecrement64(volatile __int64 *addend)	引数で指定された値を 1 ずつ減分します。fetchadd 命令に対応付けます。
__int64 _InterlockedExchange64(volatile __int64 *Target, __int64 value)	交換操作をアトミックに実行します。xchg 命令に対応付けます。
unsigned __int64 _InterlockedExchangeU64(volatile unsigned __int64 *Target, unsigned __int64 value)	InterlockedExchange64 (符号なしのデータ) と同じです。
unsigned __int64 _InterlockedCompareExchange64_rel(volatile unsigned __int64 *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	適切な設定の cmpxchg.rel 命令に対応付けます。第 1 引数 (64 ビットポインタ) で指定された値をアトミックに比較/交換します。
unsigned __int64 _InterlockedCompareExchange64_acq(volatile unsigned __int64 *Destination, unsigned __int64 Exchange, unsigned __int64 Comparand)	適切な設定の cmpxchg.acq 命令に対応付けます。第 1 引数 (64 ビットポインタ) で指定された値をアトミックに比較/交換します。
__int64 _InterlockedCompareExchange64(volatile __int64 *Destination, __int64 Exchange, __int64 Comparand)	符号付きデータについては上記と同じです。
__int64 _InterlockedExchangeAdd64(volatile __int64 *addend, __int64 increment)	比較/交換操作を使用して、加数に対して増分値をアトミックに加算します。cmpxchg 命令を使用するループに対応付けられ、アトミックな性質を保証します。
__int64 _InterlockedAdd64(volatile __int64 *addend, __int64 increment);	上記と同じです。元の値ではなく、新しい値を返します。下記の注を参照してください。

注

_InterlockedSub64 は _InterlockedAdd64 に基づいたマクロ定義として提供されています。

```
#define _InterlockedSub64(target, incr)
_InterlockedAdd64((target),(-(incr))).
```

cmpxchg を使用して、target に対して incr 値をアトミックに減算します。cmpxchg 命令を使用するループに対応付けられ、アトミックな性質を保証します。

ロードとストア

ロードとストアの組込み関数を使用して、特定のデータ・オブジェクトのメモリアクセスの順序を制限することができます。これを使用するのは、-serialize-volatile- オプションを使用して、ユーザのメモリアクセスの順序を厳密に抑止する場合です。

組込み関数	プロトタイプ	説明
__st1_rel	void __st1_rel(void *dst, const char value);	st1.rel 命令を生成します。
__st2_rel	void __st2_rel(void *dst, const short value);	st2.rel 命令を生成します。
__st4_rel	void __st4_rel(void *dst, const int value);	st4.rel 命令を生成します。
__st8_rel	void __st8_rel(void *dst, const __int64 value);	st8.rel 命令を生成します。
__ld1_acq	unsigned char __ld1_acq(void *src);	ld1.acq 命令を生成します。
__ld2_acq	unsigned short __ld2_acq(void *src);	ld2.acq 命令を生成します。
__ld4_acq	unsigned int __ld4_acq(void *src);	ld4.acq 命令を生成します。
__ld8_acq	unsigned __int64 __ld8_acq(void *src);	ld8.acq 命令を生成します。

オペレーティング・システムに関連する組込み関数

Itanium® アーキテクチャ用の組込み関数のプロトタイプは、ヘッダファイル ia64intrin.h 内にあります。

組込み関数	説明
unsigned int64 getReg(const	指定されたインデックスに基づいて、ハードウェア・レ

int whichReg)	ジスタから値を取得します。対応する mov = r 命令を生成します。以下のレジスタにアクセスできます: getReg() および setReg() のレジスタ名を参照してください。
void __setReg(const int whichReg, unsigned __int64 value)	指定されたインデックスに基づいて、ハードウェア・レジスタの値を設定します。対応する mov = r 命令を生成します。 getReg() および setReg() のレジスタ名を参照してください。
unsigned __int64 __getIndReg(const int whichIndReg, __int64 index)	インデックス付きレジスタの値を返します。インデックスは第 2 引数で、レジスタファイルは第 1 引数です。
void __setIndReg(const int whichIndReg, __int64 index, unsigned __int64 value)	インデックス付きレジスタで値をコピーします。インデックスは第 2 引数で、レジスタファイルは第 1 引数です。
void *__ptr64 _rdteb(void)	TEB アドレスを取得します。TEB アドレスは、r13 に保持され、move r=tp 命令に対応付けます。
void __isrlz(void)	シリアル化命令を実行します。srlz.i 命令に対応付けます。
void __dsrlz(void)	データをシリアル化します。srlz.d 命令に対応付けます。
unsigned __int64 __fetchadd4_acq(unsigned int *addend, const int increment)	fetchadd4.acq 命令を対応付けます。
unsigned __int64 __fetchadd4_rel(unsigned int *addend, const int increment)	fetchadd4.rel 命令を対応付けます。
unsigned __int64 __fetchadd8_acq(unsigned __int64 *addend, const int increment)	fetchadd8.acq 命令を対応付けます。
unsigned __int64 __fetchadd8_rel(unsigned __int64 *addend, const int increment)	fetchadd8.rel 命令を対応付けます。
void __fwb(void)	書き込みバッファをフラッシュします。fwb 命令に対応付けます。
void __ldfs(const int whichFloatReg, void *src)	ldfs 命令を対応付けます。指定したレジスタに単精度値をロードします。
void __ldfd(const int whichFloatReg, void *src)	ldfd 命令を対応付けます。指定したレジスタに倍精度値をロードします。
void __ldfe(const int whichFloatReg, void *src)	ldfe 命令を対応付けます。指定したレジスタに拡張精度値をロードします。
void __ldf8(const int whichFloatReg, void *src)	ldf8 命令を対応付けます。

void __ldf_fill(const int whichFloatReg, void *src)	ldf.fill 命令を対応付けます。
void __stfs(void *dst, const int whichFloatReg)	sfts 命令を対応付けます。
void __stfd(void *dst, const int whichFloatReg)	stfd 命令を対応付けます。
void __stfe(void *dst, const int whichFloatReg)	stfe 命令を対応付けます。
void __stf8(void *dst, const int whichFloatReg)	stf8 命令を対応付けます。
void __stf_spill(void *dst, const int whichFloatReg)	stf.spill 命令を対応付けます。
void __mf(void)	メモリフェンス命令を実行します。mf 命令に対応付けます。
void __mfa(void)	メモリフェンス (受け入れ形式) 命令を実行します。mf.a 命令に対応付けます。
void __synci(void)	メモリの同期化を有効にします。sync.i 命令に対応付けます。
void __thash(__int64)	変換ハッシュ・エントリ・アドレスを生成します。thash r = r 命令に対応付けます。
void __ttag(__int64)	変換ハッシュ・エントリ・タグを生成します。ttag r=r 命令に対応付けます。
void __itcd(__int64 pa)	エントリをデータ変換キャッシュに挿入します (itc.d 命令をマップします)。
void __itci(__int64 pa)	エントリを命令変換キャッシュに挿入します (itc.i を対応付けます)。
void __itrd(__int64 whichTransReg, __int64 pa)	itr.d 命令を対応付けます。
void __itri(__int64 whichTransReg, __int64 pa)	itr.i 命令を対応付けます。
void __ptce(__int64 va)	ptc.e 命令を対応付けます。
void ptcl(int64 va, int64 pagesz)	ローカル変換キャッシュをパージします。ptc.l r, r 命令に対応付けます。
void ptcg(int64 va, int64 pagesz)	グローバル変換キャッシュをパージします。ptc.g r, r 命令に対応付けます。
void __ptcga(__int64 va, __int64 pagesz)	グローバル変換キャッシュと ALAT をパージします。ptc.ga r, r 命令に対応付けます。
void ptri(int64 va, int64 pagesz)	変換レジスタをパージします。ptr.i r, r 命令に対応付けます。
void ptrd(int64 va, int64 pagesz)	変換レジスタをパージします。ptr.d r, r 命令に対応付けます。

`__int64 __tpa(__int64 va)`	`tpa` 命令を対応付けます。
`void __invalat(void)`	`ALAT` を無効化します。`invala` 命令に対応付けます。
`void __invala (void)`	`void __invalat(void)` と同じです。
`void __invala_gr(const int whichGeneralReg)`	`whichGeneralReg` = 0-127
`void __invala_fr(const int whichFloatReg)`	`whichFloatReg` = 0-127
`void __break(const int)`	ブレーク命令と即値を生成します。
`void __nop(const int)`	`nop` 命令を生成します。
`void __debugbreak(void)`	デバッグブレーク命令フォルトを生成します。
`void __fc(__int64)`	引数で指定されたアドレスに関連するキャッシュ・ラインをフラッシュします。`fc` 命令に対応付けます。
`void __sum(int mask)`	`PSR` のユーザ・マスク・ビットを設定します。`sum imm24` 命令に対応付けます。
`void __rum(int mask)`	ユーザマスクをリセットします。
`__int64 _ReturnAddress(void)`	呼び出し元のアドレスを返します。
`void __lfetch(int lfhint, void *y)`	`lfetch.lfhint` 命令を生成します。第 1 引数の値はヒントタイプを指定します。
`void __lfetch fault(int lfhint, void *y)`	`lfetch.fault.lfhint` 命令を生成します。第 1 引数の値はヒントタイプを指定します。
`void __lfetch_excl(int lfhint, void *y)`	`lfetch.excl.lfhint` 命令を生成します。第 1 引数の値 {0|1|2|3} はヒントタイプを指定します。
`void __lfetch_fault_excl(int lfhint, void *y)`	`lfetch.fault.excl.lfhint` 命令を生成します。第 1 引数の値はヒントタイプを指定します。
`unsigned int __cacheSize(unsigned int cacheLevel)`	`__cacheSize(n)` は `n` レベルのキャッシュのサイズをバイトで返します。1 は一次キャッシュを表します。キャッシュ・レベルが存在しない場合、0 が返されます。たとえば、アプリケーションはキャッシュ・サイズをクエリし、行列を操作するアルゴリズムでブロックサイズを選択するのにキャッシュ・サイズを使用します。
`void __memory_barrier(void)`	コンパイラがデータアクセス命令をスケジューリングしないバリアを生成します。コンパイラはメモリバリアのレジストリにローカルデータを割り当てます。しかしグローバルデータは割り当てません。
`void __ssm(int mask)`	システムマスクを設定します。`ssm imm24` 命令に対応付けます。
`void __rsm(int mask)`	PSR のシステム・マスク・ビットをリセットします。`rsm imm24` 命令に対応付けます。

変換組込み関数

Itanium® アーキテクチャ用の組込み関数のプロトタイプは、ヘッダファイル ia64intrin.h 内にあります。

組込み関数	説明
__int64 _m_to_int64(__m64 a)	__m64 型の a を __int64 型に変換します。Itanium ベースのシステムでは、両方のデータ型が同じレジスタに置かれるため、nop に変換します。
__m64 _m_from_int64(__int64 a)	__int64 型の a を __m64 型に変換します。Itanium ベースのシステムでは、両方のデータ型が同じレジスタに置かれるため、nop に変換します。
__int64 __round_double_to_int64(double d)	倍精度引数を符号付き整数に変換します。
unsigned __int64 __getf_exp(double d)	getf.exp 命令をマップし、オペランドの 16 ビット指数と符号を返します。

getReg() および setReg() のレジスタ名

getReg() と setReg() の組込み関数のプロトタイプは、ヘッダファイル ia64regs.h 内にあります。

名前	whichReg
_IA64_REG_IP	1016
_IA64_REG_PSR	1019
_IA64_REG_PSR_L	1019

一般的な整数レジスタ

名前	whichReg
_IA64_REG_GP	1025
_IA64_REG_SP	1036
_IA64_REG_TP	1037

アプリケーション・レジスタ

名前	whichReg
_IA64_REG_AR_KR0	3072
_IA64_REG_AR_KR1	3073
_IA64_REG_AR_KR2	3074

_IA64_REG_AR_KR3	3075
_IA64_REG_AR_KR4	3076
_IA64_REG_AR_KR5	3077
_IA64_REG_AR_KR6	3078
_IA64_REG_AR_KR7	3079
_IA64_REG_AR_RSC	3088
_IA64_REG_AR_BSP	3089
_IA64_REG_AR_BSPSTORE	3090
_IA64_REG_AR_RNAT	3091
_IA64_REG_AR_FCR	3093
_IA64_REG_AR_EFLAG	3096
_IA64_REG_AR_CSD	3097
_IA64_REG_AR_SSD	3098
_IA64_REG_AR_CFLAG	3099
_IA64_REG_AR_FSR	3100
_IA64_REG_AR_FIR	3101
_IA64_REG_AR_FDR	3102
_IA64_REG_AR_CCV	3104
_IA64_REG_AR_UNAT	3108
_IA64_REG_AR_FPSR	3112
_IA64_REG_AR_ITC	3116
_IA64_REG_AR_PFS	3136
_IA64_REG_AR_LC	3137
_IA64_REG_AR_EC	3138

コントロール・レジスタ

名前	whichReg
_IA64_REG_CR_DCR	4096
_IA64_REG_CR_ITM	4097
_IA64_REG_CR_IVA	4098
_IA64_REG_CR_PTA	4104
_IA64_REG_CR_IPSR	4112
_IA64_REG_CR_ISR	4113
_IA64_REG_CR_IIP	4115
_IA64_REG_CR_IFA	4116

_IA64_REG_CR_ITIR	4117
_IA64_REG_CR_IIPA	4118
_IA64_REG_CR_IFS	4119
_IA64_REG_CR_IIM	4120
_IA64_REG_CR_IHA	4121
_IA64_REG_CR_LID	4160
_IA64_REG_CR_IVR	4161 *
_IA64_REG_CR_TPR	4162
_IA64_REG_CR_EOI	4163
_IA64_REG_CR_IRR0	4164 *
_IA64_REG_CR_IRR1	4165 *
_IA64_REG_CR_IRR2	4166 *
_IA64_REG_CR_IRR3	4167 *
_IA64_REG_CR_ITV	4168
_IA64_REG_CR_PMV	4169
_IA64_REG_CR_CMCV	4170
_IA64_REG_CR_LRR0	4176
_IA64_REG_CR_LRR1	4177

* getReg のみ

getIndReg() および setIndReg() の間接レジスタ

名前	whichReg
_IA64_REG_INDR_CPUID	9000 *
_IA64_REG_INDR_DBR	9001
_IA64_REG_INDR_IBR	9002
_IA64_REG_INDR_PKR	9003
_IA64_REG_INDR_PMC	9004
_IA64_REG_INDR_PMD	9005
_IA64_REG_INDR_RR	9006
_IA64_REG_INDR_RESERVED	9007

* getIndReg のみ

マルチメディア乗算

Itanium® アーキテクチャ用の組込み関数のプロトタイプは、ヘッダファイル ia64intrin.h 内にあります。

組込み関数	対応する命令
__int64 _m64_czx1l(__m64 a)	czx1.l (Compute Zero Index)
__int64 _m64_czx1r(__m64 a)	czx1.r (Compute Zero Index)
__int64 _m64_czx2l(__m64 a)	czx2.l (Compute Zero Index)
__int64 _m64_czx2r(__m64 a)	czx2.r (Compute Zero Index)
__m64 _m64_mix1l(__m64 a, __m64 b)	mix1.l (Mix)
__m64 _m64_mix1r(__m64 a, __m64 b)	mix1.r (Mix)
__m64 _m64_mix2l(__m64 a, __m64 b)	mix2.l (Mix)
__m64 _m64_mix2r(__m64 a, __m64 b)	mix2.r (Mix)
__m64 _m64_mix4l(__m64 a, __m64 b)	mix4.l (Mix)
__m64 _m64_mix4r(__m64 a, __m64 b)	mix4.r (Mix)
__m64 _m64_mux1(__m64 a, const int n)	mux1 (Mux)
__m64 _m64_mux2(__m64 a, const int n)	mux2 (Mux)
__m64 _m64_padd1uus(__m64 a, __m64 b)	padd1.uus (Parallel add)
__m64 _m64_padd2uus(__m64 a, __m64 b)	padd2.uus (Parallel add)
__m64 _m64_pavg1_nraz(__m64 a, __m64 b)	pavg1 (Parallel average)
__m64 _m64_pavg2_nraz(__m64 a, __m64 b)	pavg2 (Parallel average)
__m64 _m64_pavgsub1(__m64 a, __m64 b)	pavgsub1 (Parallel average subtract)
__m64 _m64_pavgsub2(__m64 a, __m64 b)	pavgsub2 (Parallel average subtract)
__m64 _m64_pmpy2r(__m64 a, __m64 b)	pmpy2.r (Parallel multiply)
__m64 _m64_pmpy2l(__m64 a, __m64 b)	pmpy2.l (Parallel multiply)
__m64 __m64 pmpyshr2(__m64 a, __m64 b, const int count)	pmpyshr2 (Parallel multiply and shift right)
__m64 _m64_pmpyshr2u(__m64 a, __m64 b, const int count)	pmpyshr2.u (Parallel multiply and shift right)
__m64 _m64_pshladd2(__m64 a, const int count, __m64 b)	pshladd2 (Parallel shift left and add)
__m64 _m64_pshradd2(__m64 a, const int count, __m64 b)	pshradd2 (Parallel shift right and add)
__m64 _m64_psub1uus(__m64 a, __m64 b)	psub1.uus (Parallel subtract)

__m64 _m64_psub2uus(__m64 a, __m64 b)	psub2.uus (Parallel subtract)

__int64 _m64_czx1l(__m64 a)

64 ビット値 a の中の 0 の要素を、最上位の要素から最下位の要素に向かってスキャンし、最初に見つかった 0 の要素のインデックスを返します。要素の幅は 8 ビットであるため、結果の範囲は 0～7 になります。0 の要素が見つからない場合は、デフォルトでは結果は 8 になります。

__int64 _m64_czx1r(__m64 a)

64 ビット値 a の中の 0 の要素を、最下位の要素から最上位の要素に向かってスキャンし、最初に見つかった 0 の要素のインデックスを返します。要素の幅は 8 ビットであるため、結果の範囲は 0～7 になります。0 の要素が見つからない場合は、デフォルトでは結果は 8 になります。

__int64 _m64_czx2l(__m64 a)

64 ビット値 a の中の 0 の要素を、最上位の要素から最下位の要素に向かってスキャンし、最初に見つかった 0 の要素のインデックスを返します。要素の幅は 16 ビットであるため、結果の範囲は 0～3 になります。0 の要素が見つからない場合は、デフォルトでは結果は 4 になります。

__int64 _m64_czx2r(__m64 a)

64 ビット値 a の中の 0 の要素を、最下位の要素から最上位の要素に向かってスキャンし、最初に見つかった 0 の要素のインデックスを返します。要素の幅は 16 ビットであるため、結果の範囲は 0～3 になります。0 の要素が見つからない場合は、デフォルトでは結果は 4 になります。

__m64 _m64_mix1l(__m64 a, __m64 b)

64 ビット値 a と b を、図 1 に示すように、1 バイトグループ単位で左から順にインターリーブし、得られた結果を返します。

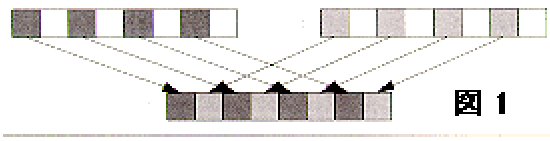
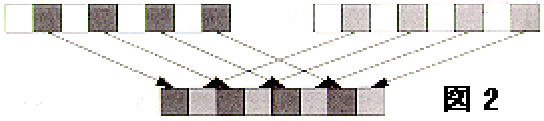


図 1

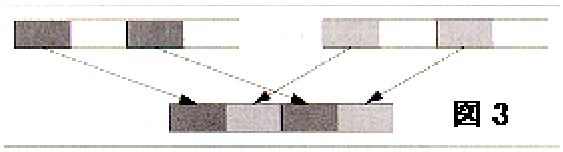
__m64 _m64_mix1r(__m64 a, __m64 b)

64 ビット値 a と b を、図 2 に示すように、1 バイトグループ単位で右から順にインターリーブし、得られた結果を返します。



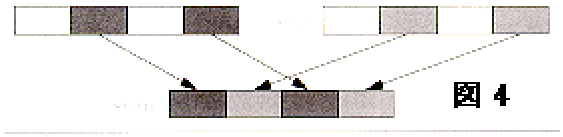

```
__m64 _m64_mix2l(__m64 a, __m64 b)
```

64 ビット値 a と b を、図 3 に示すように、2 バイトグループ単位で左から順にインターリーブし、得られた結果を返します。



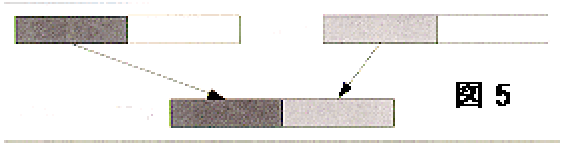

```
__m64 _m64_mix2r(__m64 a, __m64 b)
```

64 ビット値 a と b を、図 4 に示すように、4 バイトグループ単位で右から順にインターリーブし、得られた結果を返します。



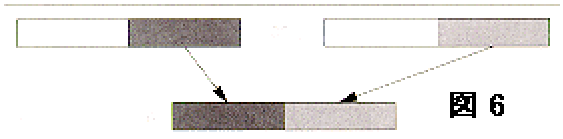

```
__m64 _m64_mix4l(__m64 a, __m64 b)
```

64 ビット値 a と b を、図 5 に示すように、4 バイトグループ単位で左から順にインターリーブし、得られた結果を返します。




```
__m64 _m64_mix4r(__m64 a, __m64 b)
```

64 ビット値 a と b を、図 6 に示すように、4 バイトグループ単位で右から順にインターリーブし、得られた結果を返します。



```
__m64 _m64_mux1(__m64 a, const int n)
```

n の値に基づいて、図 7 に示すように a の並べ替えを実行し、その結果を返します。表 1 に、指定可能な n の値を示します。

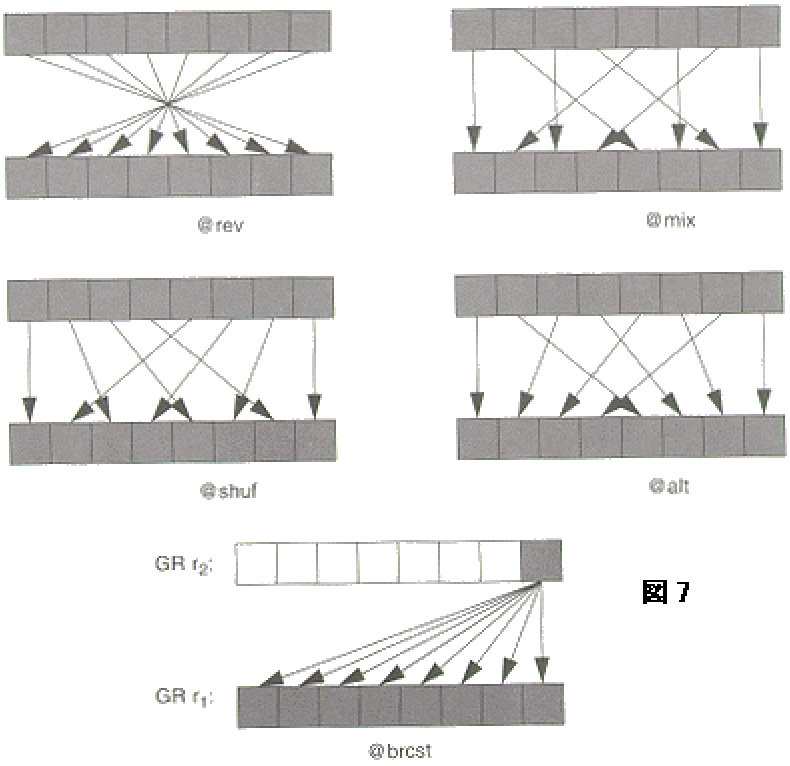


図 7

表 1. m64_mux1 の n の値

	n
@brcst	0
@mix	8
@shuf	9
@alt	0xA
@rev	0xB

```
__m64 _m64_mux2(__m64 a, const int n)
```

n の値に基づいて、図 8 に示すように a の並べ替えを実行し、その結果を返します。

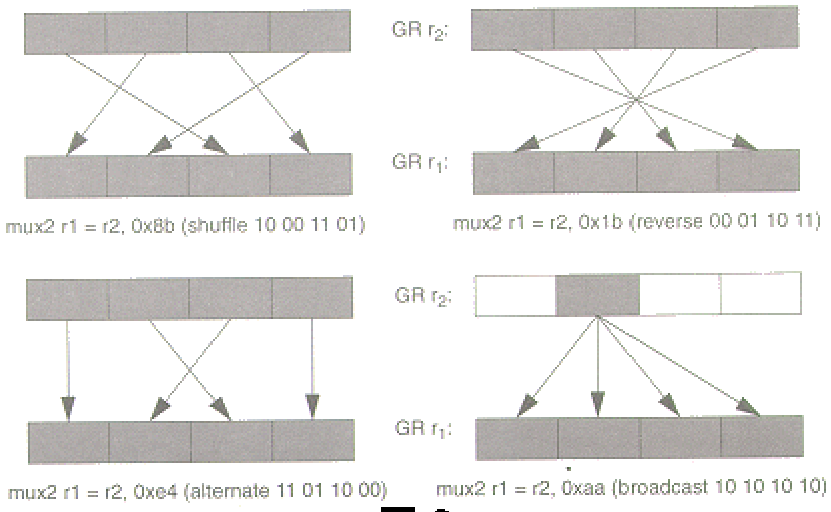


図 8

```
__m64 _m64_pavgsub1(__m64 a, __m64 b)
```

a の符号なしデータ要素 (バイト) から b の符号なしデータ要素 (バイト) を引き、減算の結果をそれぞれ 1 ポジションだけ右にシフトします。各要素の上位ビットは、減算のボロービットで埋められます。

```
__m64 _m64_pavgsub2(__m64 a, __m64 b)
```

a の符号なしデータ要素 (ダブルバイト) から b の符号なしデータ要素 (ダブルバイト) を引き、減算の結果をそれぞれ 1 ポジションだけ右にシフトします。各要素の上位ビットは、減算のボロービットで埋められます。

```
__m64 _m64_pmpy2l(__m64 a, __m64 b)
```

図 9 に示すように、a の 2 つの符号付き 16 ビットデータ要素に、最上位の要素から順に、それに対応する b の 2 つの符号付き 16 ビットデータ要素を掛けて、2 つの 32 ビットの結果を返します。

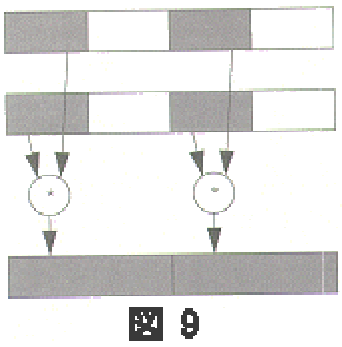


図 9

```
__m64 _m64_pmpy2r(__m64 a, __m64 b)
```

図 10 に示すように、a の 2 つの符号付き 16 ビットデータ要素に、最下位の要素から順に、それに対応する b の 2 つの符号付き 16 ビットデータ要素を掛けて、2 つの 32 ビットの結果を返します。

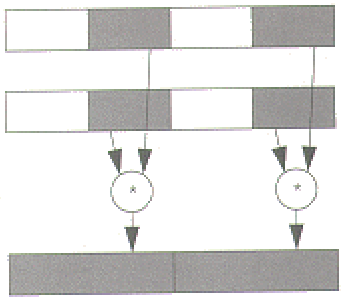


図 10

```
__m64 _m64_pmpyshr2(__m64 a, __m64 b, const int count)
```

a の 4 つの符号付き 16 ビットデータ要素に、それに対応する b の符号付き 16 ビットデータ要素を掛けて、4 つの 32 ビットの積を求めます。それぞれの積を、count ビットだけ右にシフトします。シフトしたそれぞれの積の最下位 16 ビットから、4 つの 16 ビットの結果を生成し、1 つの 64 ビットワードとして返します。

```
__m64 _m64_pmpyshr2u(__m64 a, __m64 b, const int count)
```

a の 4 つの符号なし 16 ビットデータ要素に、それに対応する b の符号なし 16 ビットデータ要素を掛けて、4 つの 32 ビットの積を求めます。それぞれの積を、count ビットだけ右にシフトします。シフトしたそれぞれの積の最下位 16 ビットから、4 つの 16 ビットの結果を生成し、1 つの 64 ビットワードとして返します。

```
__m64 _m64_pshladd2(__m64 a, const int count, __m64 b)
```

a を count ビットだけ左にシフトして、b に加算します。結果の上位 32 ビットを 0 にクリアして、b のビット [31:30] を結果のビット [62:61] にコピーし、その結果を返します。結果を返します。

```
__m64 _m64_pshradd2(__m64 a, const int count, __m64 b)
```

a の 4 つの符号付き 16 ビットデータ要素を、それぞれ count ビットだけ右にシフトします (各要素の上位ビットは、a のデータ要素の符号ビットの初期値で埋められます)。次に、これらの要素を、b の 4 つの

符号付き 16 ビットデータ要素に加算し、その結果を返します。結果を返します。

```
__m64 _m64_padd1uus(__m64 a, __m64 b)
```

8 つの 1 バイト要素として、a を b に加算します。a の要素は符号なしデータとして処理され、b の要素は符号付きデータとして処理されます。演算の結果は符号なしデータとして処理され、1 つの 64 ビットワードとして返されます。

```
__m64 _m64_padd2uus(__m64 a, __m64 b)
```

4 つの 16 ビット要素として、a を b に加算します。a の要素は符号なしデータとして処理され、b の要素は符号付きデータとして処理されます。演算の結果は符号なしデータとして処理され、1 つの 64 ビットワードとして返されます。

```
__m64 _m64_psub1uus(__m64 a, __m64 b)
```

8 つの 1 バイト要素として、b から a を引きます。a の要素は符号なしデータとして処理され、b の要素は符号付きデータとして処理されます。演算の結果は符号なしデータとして処理され、1 つの 64 ビットワードとして返されます。

```
__m64 _m64_psub2uus(__m64 a, __m64 b)
```

4 つの 16 ビット要素として、b から a を引きます。a の要素は符号なしデータとして処理され、b の要素は符号付きデータとして処理されます。演算の結果は符号なしデータとして処理され、1 つの 64 ビットワードとして返されます。

```
__m64 _m64_pavg1_nraz(__m64 a, __m64 b)
```

a の符号なしのバイトデータ要素を、b の符号なしのバイトデータ要素に加算し、それぞれの加算の結果を 1 ポジションだけ右にシフトします。各要素の上位ビットは、和のキャリービットで埋められます。

```
__m64 _m64_pavg2_nraz(__m64 a, __m64 b)
```

a の符号なしの 16 ビットデータ要素を、b の符号なしの 16 ビットデータ要素に加算し、それぞれの加算の結果を 1 ポジションだけ右にシフトします。各要素の上位ビットは、和のキャリービットで埋められます。

同期プリミティブ

同期プリミティブ組込み関数を使用して、さまざまな演算を行うことができます。これらの演算を行うことに加えて、各組込み関数には 2 つの主要なプロパティがあります:

- 実行された関数はアトミックになることが保証されます
- 各組込み関数には、コンパイラまたはプロセッサのいずれかの組込み関数演算で、可視データへのメモリ参照の移動を制限する特定のメモリ・バリア・プロパティがあります

次の組込み関数の場合、<type> は 32 ビット整数または 64 ビット整数のいずれかです。

アトミック fetch-and-op 演算

```
<type> __sync_fetch_and_add(<type> *ptr,<type> val)
<type> __sync_fetch_and_and(<type> *ptr,<type> val)
<type> __sync_fetch_and_nand(<type> *ptr,<type> val)
<type> __sync_fetch_and_or(<type> *ptr,<type> val)
<type> __sync_fetch_and_sub(<type> *ptr,<type> val)
<type> __sync_fetch_and_xor(<type> *ptr,<type> val)
```

アトミック op-and-fetch 演算

```
<type> __sync_add_and_fetch(<type> *ptr,<type> val)
<type> __sync_sub_and_fetch(<type> *ptr,<type> val)
<type> __sync_or_and_fetch(<type> *ptr,<type> val)
<type> __sync_and_and_fetch(<type> *ptr,<type> val)
<type> __sync_nand_and_fetch(<type> *ptr,<type> val)
<type> __sync_xor_and_fetch(<type> *ptr,<type> val)
```

アトミック compare-and-swap 演算

```
<type> __sync_val_compare_and_swap(<type> *ptr, <type>
old_val, <type> new_val)
int __sync_bool_compare_and_swap(<type> *ptr, <type>
old_val, <type> new_val)
```

アトミック synchronize 演算

```
void __sync_synchronize (void);
```

アトミック lock-test-and-set 演算

```
<type> __sync_lock_test_and_set(<type> *ptr,<type> val)
```

アトミック lock-release 演算

```
void __sync_lock_release(<type> *ptr)
```

その他の組込み関数

```
void* __get_return_address(unsigned int level);
```

この組込み関数は、現在の関数の復帰アドレスを返します。`level` 引数は定数値でなければなりません。値が 0 の場合、現在の関数の復帰アドレスを返します。他の値の場合、0 を返します。Linux システムでは、この組込み関数は `__builtin_return_address` と同義です。名前と引数は gcc* 互換です。

```
void __set_return_address(void* addr);
```

この組込み関数は、現在の関数の復帰アドレスを引数で示されたアドレスで上書きします。現在の呼び出しから戻ると、プログラム実行は指定されたアドレスから続行されます。

```
void* __get_frame_address(unsigned int level);
```

この組込み関数は、現在の関数のフレームアドレスを返します。`level` 引数は定数値でなければなりません。値が 0 の場合、現在の関数のフレームアドレスを返します。他の値の場合、0 を返します。Linux システムでは、この組込み関数は `__builtin_frame_address` と同義です。名前と引数は gcc* 互換です。

データのアライメント、メモリの割り当て組込み関数、およびインライン・アセンブリ

概要: データのアライメント、メモリの割り当て組込み関数、およびインライン・アセンブリ

このセクションでは、組込み関数の使用をサポートする機能について説明します。次の項目について説明します:

- アライメントのサポート
- アライメントの合ったメモリブロックの割り当てと解放
- インライン・アセンブリ

アライメントのサポート

組込み関数のパフォーマンスを向上させるには、データをアライメントする必要があります。例えば、ストリーミング SIMD 拡張命令を使用する場合は、メモリ操作の際にデータのアライメントを 16 バイトに合わせて、パフォーマンスを向上させてください。特に、_mm_load、_mm_store の組込み関数にアドレスを渡したときは、__m128 オブジェクトのアライメントを合わせなければなりません。float 型の配列を宣言し、キャストすることによってそれを __m128 オブジェクトとして扱う場合は、float 型配列のアライメントが正しく揃うようにする必要があります。

より正確なデータのアライメントを合わせるためにコンパイラへ命令するときは、__declspec(align) を使用してください。そうすると、IA-32、Itanium® ベースのシステムの両方でアライメントの正確性が上がります。例えば int 型のデータ・オブジェクトは、特に指定しなければ 4 の倍数 (すなわち int のサイズ) のバイトアドレスに配置されます。一方、__declspec(align) を使えば、4 の代わりに 8、16、32 のいずれかの倍数のアドレスを使用するようにコンパイラに命令できます。ただし IA-32 の場合は次の制約があります:

- 32 バイトアドレスは、静的に割り当てなければなりません。
- 16 バイトアドレスは、ローカルに割り当てたり、静的に割り当てたりできます。

この機能を使用して、キャッシュ・ラインの使用効率を最適化できます。よく使用する小さなオブジェクトをいくつか組み合わせて 1 つの構造体 (struct) を作り、その構造体を強制的にキャッシュ・ラインの先頭に配置すれば、どのオブジェクトにアクセスしたときでも、すぐにそのオブジェクトがキャッシュにロードされるため、パフォーマンスが大きく向上します。

この拡張属性の構文は、次のとおりです:

```
align(n)
```

ここで、*n* は、2 の何乗かを示す値 (32 以下) です。データのアライメントは、この値のバイト境界に合わせられます。

警告

このリリースでは、__declspec(align(8)) が正しく動作しません。代わりに __declspec(align(16)) を使用してください。

注

対象となるデータ型のサイズより小さい値を指定した場合は、無効になります。言い換えれば、データは、元々設定されているアライメントの最大値か、`__declspec(align)` で指定したアライメントかのいずれかに揃います。

変数の種類が静的か自動かに関わらず、個々の変数についてアライメント要求を設定できます。特に指定しない限り、グローバル変数と静的変数には静的記憶域期間が設定されて、ローカル変数には自動記憶域期間が設定されています。パラメータのアライメントは調整できません。`struct` フィールド、`class` フィールドのアライメントの調整もできません。ただし `struct` (または `union`、`class`) のアライメントは拡げられます。その場合は、該当する型のオブジェクトがすべて影響を受けます。

例えば、ある関数が 2 次元配列の添字にローカル変数 `i`、`j` を使用する場合、これらの変数は次のように宣言されます。これらの組込み関数は、次のグループに分類されます:

```
int i, j;
```

2 つの変数は、通常は組み合わせて使用します。しかし、これらの変数が異なるキャッシュ・ラインに割り当てられると、パフォーマンスが低下します。これを防ぐには、次のように 2 つの変数を宣言します:

```
__declspec(align(8)) struct { int i, j; } sub;
```

これで、コンパイラは、2 つの変数を必ず同じキャッシュ・ラインに割り当てます。C++ では、`struct` 変数名 (上の例では sub) が省略できます。ただし C の場合は必要であり、`i`、`j` への参照を、`sub.i`、`sub.j` のように記述しなければなりません。

この添字ペアの付いた関数を多く使用する場合は、次の例のように、その関数に対して `struct` 型を宣言して使用するほうが便利です:

```
typedef struct __declspec(align(8)) { int i, j; } Sub;
```

キーワード `struct` の後に `__declspec(align)` を置くと、該当する型のオブジェクトすべてに対してしかるべきアライメント要求が出ます。ただし、各パラメータの配置は `__declspec(align)` に影響されません。必要であれば、アライメントの正しく揃っているローカル変数にパラメータの値が代入できます。

また、配列などのグローバル変数についても、アライメントを指定できます。

```
__declspec(align(16)) float array[1000];
```

アライメントの合ったメモリブロックの割り当てと解放

_mm_malloc および _mm_free 組込み関数を使用して、アライメントの合ったメモリブロックの割り当てと解放を実行できます。これらの組込み関数は、libirc.a 内にある malloc および free 関数に基づいています。malloc.h をインクルードする必要があります。これらの組込み関数の構文は、次のとおりです:

```
void* _mm_malloc (int size, int align)

void _mm_free (void *p)
```

_mm_malloc ルーチンには、追加のパラメータを 1 つ指定する必要があります。このパラメータは、アライメントの制約条件です。この制約条件は、2 の何乗かを示す値です。_mm_malloc で返すポインタは、指定した境界にアライメントが合っているのを保証します。

注

_mm_malloc を使用して割り当てられたメモリは、_mm_free を使用して解放しなければなりません。_mm_malloc で割り当てられたメモリ上で free を呼び出したり、malloc で割り当てられたメモリ上で _mm_free を呼び出したりすると、予測できない動作が発生します。

インライン・アセンブリ

デフォルトでは、C、C++、および数値演算の各標準ライブラリ関数のいくつかはコンパイラによってインライン化されます。通常、この処理によってプログラムの実行速度が速くなります。

ライブラリ関数をインライン展開すると、予期しない結果になる場合があります。インライン化されたライブラリ関数では、errno 変数は設定されません。したがって、errno 変数を設定するかしないかによって動作が異なるコードに対しては、-nolib_inline オプションを使用しなければなりません。そうすれば、ライブラリ関数のインライン展開は禁止されます。また、コンパイラから提供されるライブラリ関数と同じ名前を持つ関数はライブラリ関数と見なすため、元々の呼び出し命令は、インライン化されたものと置き換わります。したがって、既知のライブラリ・ルーチンのいずれかと同じ名前を持つ関数がプログラムの中で定義している場合は、その関数そのものが必ず使用されるように -nolib_inline オプションを使用する必要があります。

注

ライブラリ関数の自動インライン展開は、プロシージャ間の最適化処理中にコンパイラが行うインライン展開とは関連がありません。例えば、次のコマンドを実行すると、sum.cpp というプログラムがコンパイルされます。このとき、ライブラリ関数の展開は行いませんが、プロシージャ間の最適化 (IPO) によるインライン展開は行います:

```
prompt>icpc -ip -nolib_inline sum.cpp
```

IPO の詳細は、「プロシージャ間の最適化」を参照してください。

MASM* スタイルのインライン・アセンブリ

インテル® C++ コンパイラは、-use_msasm オプションによって MASM* スタイルのインライン・アセンブリをサポートします。構文については、MASM のマニュアルを参照してください。

GNU* 式スタイルのインライン・アセンブリ (IA-32 のみ)

インテル C++ コンパイラは、GNU 式 スタイルのインライン・アセンブリをサポートします。構文は、次のとおりです:

```
asm-keyword [ volatile-keyword ] ( asm-template [ asm-interface ] ) ;
```

構文要素	説明
asm-keyword	asm 文は asm キーワードで始まります。または、互換性のため、__asm もしくは __asm__ が使用されることもあります。
volatile-keyword	オプションの volatile キーワードが指定されたら、asm は volatile です。2 つの volatile asm 文は、お互いに移動しません。volatile 変数への参照は、volatile asm へ相対移動しません。または、互換性のため、__volatile もしくは __volatile__ が使用されることもあります。
asm-template	asm-template は、アセンブリ・コードを出力する方法を指定する C 言語の ASCII 文字列です。テンプレートの多くは固定文字列です。代入ディレクティブ以外のすべては、アセンブリにそのまま渡されます。代入ディテクティブの構文は、% の後に 1 または 2 文字続きます。サポートされた代入ディテクティブは、次のセクションで指定されます。
asm-interface	asm-interface は次の 3 つの部分で構成されます: 1. output-list (オプション) 2. input-list (オプション) 3. clobber-list (オプション) これらは、コロン (:) で区切ります。output-list がなく、input-list が指定される場合、output-list の代わりに、input-list は 2 つのコロン (::) に続きます。asm-interface がすべて省略された場合、

	volatile-keyword の指定の有無にかかわらず、asm 文は volatile とみなされます。
output-list	output-list は、カンマで区切られた 1 つ以上の output-specs から構成されます。asm-template に代入するために、各 output-spec には番号が付けられます。 output-list の最初のオペランドは 0 で、次は 1 のようになります。番号付けは、output-list から input-list へ続行します。オペランドの合計数は 0-9 個です。
input-list	output-list と類似して、input-list はカンマで区切られた 1 つ以上の input-specs から構成されます。asm-template に代入するために、各 input-spec には番号が付けられます。番号は、output-list のオペランドから続きます。
clobber-list	clobber-list は asm が特定のマシンレジスタを使用または変更することをコンパイラに伝えます。特定のマシンレジスタは直接 asm にコードされるか、アセンブリの命令によって暗黙的に変更されます。clobber-list は、カンマで区切られた clobber-specs のリストです。
input-spec	input-specs は、挿入されたアセンブリの命令によって必要とされる値の式をコンパイラに伝えます。asm の必須入力をすべて示すのに、実際には asm-template に参照されない input-spec を一覧表示できます。
clobber-spec	各 clobber-spec は、壊れた 1 つのマシンレジスタ名を指定します。レジスタ名は、オプションで先頭に % を使用できます。有効なレジスタ名: eax, ebx, ecx, edx, esi, edi, ebp, esp, ax, bx, cx, dx, si, di, bp, sp, al, bl, cl, dl, ah, bh, ch, dh, st, st(1) - st(7), mm0 - mm7, xmm0 - xmm7, および cc また、clobber-spec で"メモリ" を指定することもできます。これを指定すると、コンパイラはレジスタにキャッシュされたデータを asm 文に渡さないようにします。

各種のプロセッサでの組込み関数の使用

各種のプロセッサでの組込み関数の使用

このセクションでは、各種のアーキテクチャ上での組込み関数のパフォーマンスを比較する一連の表を記載しています。各アーキテクチャ上で組込み関数を使用する前に、次の点に注意してください。

- 組込み関数を使用すると、一部の IA プロセッサ上で正常に動作しないコードが生成される可能性があります。したがって、プログラマが CPUID 命令を使用してプロセッサを検出し、適切なコードを生成しなければなりません。
- 組込み関数は、特定のプロセッサ向けにではなく、プロセッサ・ファミリごとに使用してください。組込み関数がどちらのファミリ (IA-32 または Itanium® プロセッサ) 上でサポートされているかは、互換性ではなく、主にパフォーマンスによって決められています。両方のファミリでパフォーマンスが向上する場合は、同じ組込み関数が使用されます。

すべての IA プロセッサでサポートされる組込み関数

次の組込み関数は、組込み関数を使用しない同等のコードと比べてパフォーマンスが大きく向上します。

int abs(int)
long labs(long)
unsigned long __lrotl(unsigned long value, int shift)
unsigned long __lrotr(unsigned long value, int shift)
unsigned int __rotl(unsigned int value, int shift)
unsigned int __rotr(unsigned int value, int shift)
__int64 __i64_rotl(__int64 value, int shift)
__int64 __i64_rotr(__int64 value, int shift)
double fabs(double)
double log(double)
float logf(float)
double log10(double)
float log10f(float)
double exp(double)
float expf(float)
double pow(double, double)
float powf(float, float)
double sin(double)
float sinf(float)
double cos(double)
float cosf(float)
double tan(double)
float tanf(float)
double acos(double)
float acosf(float)
double acosh(double)
float acoshf(float)
double asin(double)
float asinf(float)
double asinh(double)
float asinhf(float)
double atan(double)
float atanf(float)

double atanh(double)
float atanhf(float)
float cabs(double)*
double ceil(double)
float ceilf(float)
double cosh(double)
float coshf(float)
float fabsf(float)
double floor(double)
float floorf(float)
double fmod(double)
float fmodf(float)
double hypot(double, double)
float hypotf(float)
double rint(double)
float rintf(float)
double sinh(double)
float sinhf(float)
float sqrtf(float)
double tanh(double)
float tanhf(float)
char *_strset(char *, _int32)
void *memcmp(const void *cs, const void *ct, size_t n)
void *memcpy(void *s, const void *ct, size_t n)
void *memset(void * s, int c, size_t n)
char *Strcat(char * s, const char * ct)
int *strcmp(const char *, const char *)
char *strcpy(char * s, const char * ct)
size_t strlen(const char * cs)
int strncmp(char *, char *, int)
int strncpy(char *, char *, int)
void *__alloca(int)
int _setjmp(jmp_buf)
_exception_code(void)
_exception_info(void)
_abnormal_termination(void)
void _enable()

void _disable()
int _in_byte(int)
int _in_dword(int)
int _in_word(int)
int _inp(int)
int _inpd(int)
int _inpw(int)
int _out_byte(int, int)
int _out_dword(int, int)
int _out_word(int, int)
int _outp(int, int)
int _outpd(int, int)
int _outpw(int, int)

MMX® テクノロジの組込み関数

表中の項目の意味は、次のとおりです:

- A = 組込み関数を使用しない同等のコードと比べて、パフォーマンスが大きく向上すると予想されるもの。
- B = 組込み関数を使用しないソースコードを使用した方がよいもの。組込み関数はネイティブ命令に直接に対応付けられますが、パフォーマンスはほとんど向上しません。
- C = 特定のマイクロアーキテクチャでは対応する命令が存在しないもの。組込み関数を使用すると、パフォーマンスが大きく低下します。

組込み関数名	別名	すべての IA	MMX® テクノロジ ストリーミング SIMD 拡張命令 ストリーミング SIMD 拡張命令 2	Itanium® アーキテクチャ
_m_empty	_mm_empty	N/A	A	B
_m_from_int	_mm_cvtsi32_si64	N/A	A	A
_m_to_int	_mm_cvtsi64_si32	N/A	A	A

_m_packsswb	_mm_packs_pi16	N/A	A	A
_m_packssdw	_mm_packs_pi32	N/A	A	A
_m_packuswb	_mm_packs_pu16	N/A	A	A
_m_punpckhbw	_mm_unpackhi_pi8	N/A	A	A
_m_punpckhwd	_mm_unpackhi_pi16	N/A	A	A
_m_punpckhdq	_mm_unpackhi_pi32	N/A	A	A
_m_punpcklbw	_mm_unpacklo_pi8	N/A	A	A
_m_punpcklwd	_mm_unpacklo_pi16	N/A	A	A
_m_punpckldq	_mm_unpacklo_pi32	N/A	A	A
_m_paddb	_mm_add_pi8	N/A	A	A
_m_paddw	_mm_add_pi16	N/A	A	A
_m_paddd	_mm_add_pi32	N/A	A	A
_m_paddsb	_mm_adds_pi8	N/A	A	A
_m_paddsw	_mm_adds_pi16	N/A	A	A
_m_paddusb	_mm_adds_pu8	N/A	A	A
_m_paddusw	_mm_adds_pu16	N/A	A	A
_m_psubb	_mm_sub_pi8	N/A	A	A
_m_psubw	_mm_sub_pi16	N/A	A	A
_m_psubd	_mm_sub_pi32	N/A	A	A
_m_psubsb	_mm_subs_pi8	N/A	A	A
_m_psubsw	_mm_subs_pi16	N/A	A	A
_m_psubusb	_mm_subs_pu8	N/A	A	A
_m_psubusw	_mm_subs_pu16	N/A	A	A
_m_pmaddwd	_mm_madd_pi16	N/A	A	C
_m_pmulhw	_mm_mulhi_pi16	N/A	A	A
_m_pmullw	_mm_mullo_pi16	N/A	A	A
_m_psllw	_mm_sll_pi16	N/A	A	A
_m_psllwi	_mm_slli_pi16	N/A	A	A
_m_pslld	_mm_sll_pi32	N/A	A	A
_m_pslldi	_mm_slli_pi32	N/A	A	A
_m_psllq	_mm_sll_si64	N/A	A	A
_m_psllqi	_mm_slli_si64	N/A	A	A
_m_psraw	_mm_sra_pi16	N/A	A	A
_m_psrawi	_mm_srai_pi16	N/A	A	A

_m_psrad	_mm_sra_pi32	N/A	A	A
_m_psradi	_mm_srai_pi32	N/A	A	A
_m_psrlw	_mm_srl_pi16	N/A	A	A
_m_psrlwi	_mm_srli_pi16	N/A	A	A
_m_psrld	_mm_srl_pi32	N/A	A	A
_m_psrldi	_mm_srli_pi32	N/A	A	A
_m_psrlq	_mm_srl_si64	N/A	A	A
_m_psrlqi	_mm_srli_si64	N/A	A	A
_m_pand	_mm_and_si64	N/A	A	A
_m_pandn	_mm_andnot_si64	N/A	A	A
_m_por	_mm_or_si64	N/A	A	A
_m_pxor	_mm_xor_si64	N/A	A	A
_m_pcmpeqb	_mm_cmpeq_pi8	N/A	A	A
_m_pcmpeqw	_mm_cmpeq_pi16	N/A	A	A
_m_pcmpeqd	_mm_cmpeq_pi32	N/A	A	A
_m_pcmpgtb	_mm_cmpgt_pi8	N/A	A	A
_m_pcmpgtw	_mm_cmpgt_pi16	N/A	A	A
_m_pcmpgtd	_mm_cmpgt_pi32	N/A	A	A
_mm_setzero_si64		N/A	A	A
_mm_set_pi32		N/A	A	A
_mm_set_pi16		N/A	A	C
_mm_set_pi8		N/A	A	C
_mm_set1_pi32		N/A	A	A
_mm_set1_pi16		N/A	A	A
_mm_set1_pi8		N/A	A	A
_mm_setr_pi32		N/A	A	A
_mm_setr_pi16		N/A	A	C
_mm_setr_pi8		N/A	A	C

_mm_empty は、Itanium 命令では、ソースの互換性のためにのみ NOP としてサポートされています。

ストリーミング SIMD 拡張命令の組込み関数

通常のストリーミング SIMD 拡張命令の組込み関数は、4 つの 32 ビット単精度浮動小数点値を操作します。Itanium® ベースのシステム上では、加算や比較などの基本

演算に 2 つの SIMD 命令が必要です。2 つの SIMD 命令を同じサイクルで実行できるため、スループットは、1 サイクル当たり 1 つのストリーミング SIMD 拡張命令基本演算、すなわち 1 サイクル当たり 4 つの 32 ビット単精度浮動小数点値の演算になります。

表中の項目の意味は、次のとおりです:

- A = 組込み関数を使用しない同等のコードと比べて、パフォーマンスが大きく向上すると予想されるもの。
- B = 組込み関数を使用しないソースコードを使用した方がよいもの。組込み関数はネイティブ命令に直接に対応付けられますが、パフォーマンスはほとんど向上しません。
- C = 特定のマイクロアーキテクチャでは対応する命令が存在しないもの。組込み関数を使用すると、パフォーマンスが大きく低下します。

組込み関数の名前	別名	すべての IA	MMX® テクノロジ	ストリーミング SIMD 拡張命令 ストリーミング SIMD 拡張命令 2	Itanium アーキテクチャ
_mm_add_ss		N/A	N/A	B	B
_mm_add_ps		N/A	N/A	A	A
_mm_sub_ss		N/A	N/A	B	B
_mm_sub_ps		N/A	N/A	A	A
_mm_mul_ss		N/A	N/A	B	B
_mm_mul_ps		N/A	N/A	A	A
_mm_div_ss		N/A	N/A	B	B
_mm_div_ps		N/A	N/A	A	A
_mm_sqrt_ss		N/A	N/A	B	B
_mm_sqrt_ps		N/A	N/A	A	A
_mm_rcp_ss		N/A	N/A	B	B
_mm_rcp_ps		N/A	N/A	A	A
_mm_rsqrt_ss		N/A	N/A	B	B
_mm_rsqrt_ps		N/A	N/A	A	A

_mm_min_ss		N/A	N/A	B	B
_mm_min_ps		N/A	N/A	A	A
_mm_max_ss		N/A	N/A	B	B
_mm_max_ps		N/A	N/A	A	A
_mm_and_ps		N/A	N/A	A	A
_mm_andnot_ps		N/A	N/A	A	A
_mm_or_ps		N/A	N/A	A	A
_mm_xor_ps		N/A	N/A	A	A
_mm_cmpeq_ss		N/A	N/A	B	B
_mm_cmpeq_ps		N/A	N/A	A	A
_mm_cmplt_ss		N/A	N/A	B	B
_mm_cmplt_ps		N/A	N/A	A	A
_mm_cmple_ss		N/A	N/A	B	B
_mm_cmple_ps		N/A	N/A	A	A
_mm_cmpgt_ss		N/A	N/A	B	B
_mm_cmpgt_ps		N/A	N/A	A	A
_mm_cmpge_ss		N/A	N/A	B	B
_mm_cmpge_ps		N/A	N/A	A	A
_mm_cmpneq_ss		N/A	N/A	B	B
_mm_cmpneq_ps		N/A	N/A	A	A
_mm_cmpnlt_ss		N/A	N/A	B	B
_mm_cmpnlt_ps		N/A	N/A	A	A
_mm_cmpnle_ss		N/A	N/A	B	B
_mm_cmpnle_ps		N/A	N/A	A	A
_mm_cmpngt_ss		N/A	N/A	B	B
_mm_cmpngt_ps		N/A	N/A	A	A
_mm_cmpnge_ss		N/A	N/A	B	B
_mm_cmpnge_ps		N/A	N/A	A	A
_mm_cmpord_ss		N/A	N/A	B	B
_mm_cmpord_ps		N/A	N/A	A	A
_mm_cmpunord_ss		N/A	N/A	B	B
_mm_cmpunord_ps		N/A	N/A	A	A
_mm_comieq_ss		N/A	N/A	B	B
_mm_comilt_ss		N/A	N/A	B	B

_mm_comile_ss		N/A	N/A	B	B
_mm_comigt_ss		N/A	N/A	B	B
_mm_comige_ss		N/A	N/A	B	B
_mm_comineq_ss		N/A	N/A	B	B
_mm_ucomieq_ss		N/A	N/A	B	B
_mm_ucomilt_ss		N/A	N/A	B	B
_mm_ucomile_ss		N/A	N/A	B	B
_mm_ucomigt_ss		N/A	N/A	B	B
_mm_ucomige_ss		N/A	N/A	B	B
_mm_ucomineq_ss		N/A	N/A	B	B
_mm_cvt_ss2si	_mm_cvtss_si32	N/A	N/A	A	B
_mm_cvt_ps2pi	_mm_cvtps_pi32	N/A	N/A	A	A
_mm_cvtt_ss2si	_mm_cvttss_si32	N/A	N/A	A	B
_mm_cvtt_ps2pi	_mm_cvttps_pi32	N/A	N/A	A	A
_mm_cvt_si2ss	_mm_cvtsi32_ss	N/A	N/A	A	B
_mm_cvt_pi2ps	_mm_cvtpi32_ps	N/A	N/A	A	C
_mm_cvtpi16_ps		N/A	N/A	A	C
_mm_cvtpu16_ps		N/A	N/A	A	C
_mm_cvtpi8_ps		N/A	N/A	A	C
_mm_cvtpu8_ps		N/A	N/A	A	C
_mm_cvtpi32x2_ps		N/A	N/A	A	C
_mm_cvtps_pi16		N/A	N/A	A	C
_mm_cvtps_pi8		N/A	N/A	A	C
_mm_move_ss		N/A	N/A	A	A
_mm_shuffle_ps		N/A	N/A	A	A
_mm_unpackhi_ps		N/A	N/A	A	A
_mm_unpacklo_ps		N/A	N/A	A	A
_mm_movehl_ps		N/A	N/A	A	A
_mm_movelh_ps		N/A	N/A	A	A
_mm_movemask_ps		N/A	N/A	A	C
_mm_getcsr		N/A	N/A	A	A
_mm_setcsr		N/A	N/A	A	A
_mm_loadh_pi		N/A	N/A	A	A
_mm_loadl_pi		N/A	N/A	A	A

_mm_load_ss		N/A	N/A	A	B
_mm_load_ps1	_mm_load1_ps	N/A	N/A	A	A
_mm_load_ps		N/A	N/A	A	A
_mm_loadu_ps		N/A	N/A	A	A
_mm_loadr_ps		N/A	N/A	A	A
_mm_storeh_pi		N/A	N/A	A	A
_mm_storel_pi		N/A	N/A	A	A
_mm_store_ss		N/A	N/A	A	A
_mm_store_ps		N/A	N/A	A	A
_mm_store_ps1	_mm_store1_ps	N/A	N/A	A	A
_mm_storeu_ps		N/A	N/A	A	A
_mm_storer_ps		N/A	N/A	A	A
_mm_set_ss		N/A	N/A	A	A
_mm_set_ps1	_mm_set1_ps	N/A	N/A	A	A
_mm_set_ps		N/A	N/A	A	A
_mm_setr_ps		N/A	N/A	A	A
_mm_setzero_ps		N/A	N/A	A	A
_mm_prefetch		N/A	N/A	A	A
_mm_stream_pi		N/A	N/A	A	A
_mm_stream_ps		N/A	N/A	A	A
_mm_sfence		N/A	N/A	A	A
_m_pextrw	_mm_extract_pi16	N/A	N/A	A	A
_m_pinsrw	_mm_insert_pi16	N/A	N/A	A	A
_m_pmaxsw	_mm_max_pi16	N/A	N/A	A	A
_m_pmaxub	_mm_max_pu8	N/A	N/A	A	A
_m_pminsw	_mm_min_pi16	N/A	N/A	A	A
_m_pminub	_mm_min_pu8	N/A	N/A	A	A
_m_pmovmskb	_mm_movemask_pi8	N/A	N/A	A	C
_m_pmulhuw	_mm_mulhi_pu16	N/A	N/A	A	A
_m_pshufw	_mm_shuffle_pi16	N/A	N/A	A	A
_m_maskmovq	_mm_maskmove_si64	N/A	N/A	A	C
_m_pavgb	_mm_avg_pu8	N/A	N/A	A	A
_m_pavgw	_mm_avg_pu16	N/A	N/A	A	A
_m_psadbw	_mm_sad_pu8	N/A	N/A	A	A

ストリーミング SIMD 拡張命令 2 の組込み関数

ストリーミング SIMD 拡張命令 2 では、128 ビットデータ (2 つの 64 ビット倍精度浮動小数点値) を操作します。インテル® Itanium® プロセッサは倍精度演算の並列処理をサポートしていないため、ストリーミング SIMD 拡張命令 2 は Itanium ベースのシステム上では使用できません。

表中の項目の意味は、次のとおりです:

- A = 組込み関数を使用しない同等のコードと比べて、パフォーマンスが大きく向上すると予想されるもの。
- B = 組込み関数を使用しないソースコードを使用した方がよいもの。組込み関数はネイティブ命令に直接に対応付けられますが、パフォーマンスはほとんど向上しません。
- C = 特定のマイクロアーキテクチャでは対応する命令が存在しないもの。組込み関数を使用すると、パフォーマンスが大きく低下します。

組込み関数	すべての IA	MMX® テクノロジ	ストリーミング SIMD 拡張命令	ストリーミング SIMD 拡張命令 2	Itanium アーキテクチャ
_mm_add_sd	N/A	N/A	N/A	A	N/A
_mm_add_pd	N/A	N/A	N/A	A	N/A
_mm_sub_sd	N/A	N/A	N/A	A	N/A
_mm_sub_pd	N/A	N/A	N/A	A	N/A
_mm_mul_sd	N/A	N/A	N/A	A	N/A
_mm_mul_pd	N/A	N/A	N/A	A	N/A
_mm_sqrt_sd	N/A	N/A	N/A	A	N/A
_mm_sqrt_pd	N/A	N/A	N/A	A	N/A
_mm_div_sd	N/A	N/A	N/A	A	N/A
_mm_div_pd	N/A	N/A	N/A	A	N/A
_mm_min_sd	N/A	N/A	N/A	A	N/A
_mm_min_pd	N/A	N/A	N/A	A	N/A
_mm_max_sd	N/A	N/A	N/A	A	N/A
_mm_max_pd	N/A	N/A	N/A	A	N/A
_mm_and_pd	N/A	N/A	N/A	A	N/A
_mm_andnot_pd	N/A	N/A	N/A	A	N/A
_mm_or_pd	N/A	N/A	N/A	A	N/A

_mm_xor_pd	N/A	N/A	N/A	A	N/A
_mm_cmpeq_sd	N/A	N/A	N/A	A	N/A
_mm_cmpeq_pd	N/A	N/A	N/A	A	N/A
_mm_cmplt_sd	N/A	N/A	N/A	A	N/A
_mm_cmplt_pd	N/A	N/A	N/A	A	N/A
_mm_cmple_sd	N/A	N/A	N/A	A	N/A
_mm_cmple_pd	N/A	N/A	N/A	A	N/A
_mm_cmpgt_sd	N/A	N/A	N/A	A	N/A
_mm_cmpgt_pd	N/A	N/A	N/A	A	N/A
_mm_cmpge_sd	N/A	N/A	N/A	A	N/A
_mm_cmpge_pd	N/A	N/A	N/A	A	N/A
_mm_cmpneq_sd	N/A	N/A	N/A	A	N/A
_mm_cmpneq_pd	N/A	N/A	N/A	A	N/A
_mm_cmpnlt_sd	N/A	N/A	N/A	A	N/A
_mm_cmpnlt_pd	N/A	N/A	N/A	A	N/A
_mm_cmpnle_sd	N/A	N/A	N/A	A	N/A
_mm_cmpnle_pd	N/A	N/A	N/A	A	N/A
_mm_cmpngt_sd	N/A	N/A	N/A	A	N/A
_mm_cmpngt_pd	N/A	N/A	N/A	A	N/A
_mm_cmpnge_sd	N/A	N/A	N/A	A	N/A
_mm_cmpnge_pd	N/A	N/A	N/A	A	N/A
_mm_cmpord_pd	N/A	N/A	N/A	A	N/A
_mm_cmpord_sd	N/A	N/A	N/A	A	N/A
_mm_cmpunord_pd	N/A	N/A	N/A	A	N/A
_mm_cmpunord_sd	N/A	N/A	N/A	A	N/A
_mm_comieq_sd	N/A	N/A	N/A	A	N/A
_mm_comilt_sd	N/A	N/A	N/A	A	N/A
_mm_comile_sd	N/A	N/A	N/A	A	N/A
_mm_comigt_sd	N/A	N/A	N/A	A	N/A
_mm_comige_sd	N/A	N/A	N/A	A	N/A
_mm_comineq_sd	N/A	N/A	N/A	A	N/A
_mm_ucomieq_sd	N/A	N/A	N/A	A	N/A
_mm_ucomilt_sd	N/A	N/A	N/A	A	N/A
_mm_ucomile_sd	N/A	N/A	N/A	A	N/A

_mm_ucomigt_sd	N/A	N/A	N/A	A	N/A
_mm_ucomige_sd	N/A	N/A	N/A	A	N/A
_mm_ucomineq_sd	N/A	N/A	N/A	A	N/A
_mm_cvtepi32_pd	N/A	N/A	N/A	A	N/A
_mm_cvtpd_epi32	N/A	N/A	N/A	A	N/A
_mm_cvttpd_epi32	N/A	N/A	N/A	A	N/A
_mm_cvtepi32_ps	N/A	N/A	N/A	A	N/A
_mm_cvtps_epi32	N/A	N/A	N/A	A	N/A
_mm_cvttps_epi32	N/A	N/A	N/A	A	N/A
_mm_cvtpd_ps	N/A	N/A	N/A	A	N/A
_mm_cvtps_pd	N/A	N/A	N/A	A	N/A
_mm_cvtsd_ss	N/A	N/A	N/A	A	N/A
_mm_cvtss_sd	N/A	N/A	N/A	A	N/A
_mm_cvtsd_si32	N/A	N/A	N/A	A	N/A
_mm_cvttsd_si32	N/A	N/A	N/A	A	N/A
_mm_cvtsi32_sd	N/A	N/A	N/A	A	N/A
_mm_cvtpd_pi32	N/A	N/A	N/A	A	N/A
_mm_cvttpd_pi32	N/A	N/A	N/A	A	N/A
_mm_cvtpi32_pd	N/A	N/A	N/A	A	N/A
_mm_unpackhi_pd	N/A	N/A	N/A	A	N/A
_mm_unpacklo_pd	N/A	N/A	N/A	A	N/A
_mm_unpacklo_pd	N/A	N/A	N/A	A	N/A
_mm_shuffle_pd	N/A	N/A	N/A	A	N/A
_mm_load_pd	N/A	N/A	N/A	A	N/A
_mm_load1_pd	N/A	N/A	N/A	A	N/A
_mm_loadr_pd	N/A	N/A	N/A	A	N/A
_mm_loadu_pd	N/A	N/A	N/A	A	N/A
_mm_load_sd	N/A	N/A	N/A	A	N/A
_mm_loadh_pd	N/A	N/A	N/A	A	N/A
_mm_loadl_pd	N/A	N/A	N/A	A	N/A
_mm_set_sd	N/A	N/A	N/A	A	N/A
_mm_set1_pd	N/A	N/A	N/A	A	N/A
_mm_set_pd	N/A	N/A	N/A	A	N/A
_mm_setr_pd	N/A	N/A	N/A	A	N/A

_mm_setzero_pd	N/A	N/A	N/A	A	N/A
_mm_move_sd	N/A	N/A	N/A	A	N/A
_mm_store_sd	N/A	N/A	N/A	A	N/A
_mm_store1_pd	N/A	N/A	N/A	A	N/A
_mm_store_pd	N/A	N/A	N/A	A	N/A
_mm_storeu_pd	N/A	N/A	N/A	A	N/A
_mm_storer_pd	N/A	N/A	N/A	A	N/A
_mm_storeh_pd	N/A	N/A	N/A	A	N/A
_mm_storel_pd	N/A	N/A	N/A	A	N/A
_mm_add_epi8	N/A	N/A	N/A	A	N/A
_mm_add_epi16	N/A	N/A	N/A	A	N/A
_mm_add_epi32	N/A	N/A	N/A	A	N/A
_mm_add_si64	N/A	N/A	N/A	A	N/A
_mm_add_epi64	N/A	N/A	N/A	A	N/A
_mm_adds_epi8	N/A	N/A	N/A	A	N/A
_mm_adds_epi16	N/A	N/A	N/A	A	N/A
_mm_adds_epu8	N/A	N/A	N/A	A	N/A
_mm_adds_epu16	N/A	N/A	N/A	A	N/A
_mm_avg_epu8	N/A	N/A	N/A	A	N/A
_mm_avg_epu16	N/A	N/A	N/A	A	N/A
_mm_madd_epi16	N/A	N/A	N/A	A	N/A
_mm_max_epi16	N/A	N/A	N/A	A	N/A
_mm_max_epu8	N/A	N/A	N/A	A	N/A
_mm_min_epi16	N/A	N/A	N/A	A	N/A
_mm_min_epu8	N/A	N/A	N/A	A	N/A
_mm_mulhi_epi16	N/A	N/A	N/A	A	N/A
_mm_mulhi_epu16	N/A	N/A	N/A	A	N/A
_mm_mullo_epi16	N/A	N/A	N/A	A	N/A
_mm_mul_su32	N/A	N/A	N/A	A	N/A
_mm_mul_epu32	N/A	N/A	N/A	A	N/A
_mm_sad_epu8	N/A	N/A	N/A	A	N/A
_mm_sub_epi8	N/A	N/A	N/A	A	N/A
_mm_sub_epi16	N/A	N/A	N/A	A	N/A
_mm_sub_epi32	N/A	N/A	N/A	A	N/A

_mm_sub_si64	N/A	N/A	N/A	A	N/A
_mm_sub_epi64	N/A	N/A	N/A	A	N/A
_mm_subs_epi8	N/A	N/A	N/A	A	N/A
_mm_subs_epi16	N/A	N/A	N/A	A	N/A
_mm_subs_epu8	N/A	N/A	N/A	A	N/A
_mm_subs_epu16	N/A	N/A	N/A	A	N/A
_mm_and_si128	N/A	N/A	N/A	A	N/A
_mm_andnot_si128	N/A	N/A	N/A	A	N/A
_mm_or_si128	N/A	N/A	N/A	A	N/A
_mm_xor_si128	N/A	N/A	N/A	A	N/A
_mm_slli_si128	N/A	N/A	N/A	A	N/A
_mm_slli_epi16	N/A	N/A	N/A	A	N/A
_mm_sll_epi16	N/A	N/A	N/A	A	N/A
_mm_slli_epi32	N/A	N/A	N/A	A	N/A
_mm_sll_epi32	N/A	N/A	N/A	A	N/A
_mm_slli_epi64	N/A	N/A	N/A	A	N/A
_mm_sll_epi64	N/A	N/A	N/A	A	N/A
_mm_srai_epi16	N/A	N/A	N/A	A	N/A
_mm_sra_epi16	N/A	N/A	N/A	A	N/A
_mm_srai_epi32	N/A	N/A	N/A	A	N/A
_mm_sra_epi32	N/A	N/A	N/A	A	N/A
_mm_srli_si128	N/A	N/A	N/A	A	N/A
_mm_srli_epi16	N/A	N/A	N/A	A	N/A
_mm_srl_epi16	N/A	N/A	N/A	A	N/A
_mm_srli_epi32	N/A	N/A	N/A	A	N/A
_mm_srl_epi32	N/A	N/A	N/A	A	N/A
_mm_srli_epi64	N/A	N/A	N/A	A	N/A
_mm_srl_epi64	N/A	N/A	N/A	A	N/A
_mm_cmpeq_epi8	N/A	N/A	N/A	A	N/A
_mm_cmpeq_epi16	N/A	N/A	N/A	A	N/A
_mm_cmpeq_epi32	N/A	N/A	N/A	A	N/A
_mm_cmpgt_epi8	N/A	N/A	N/A	A	N/A
_mm_cmpgt_epi16	N/A	N/A	N/A	A	N/A
_mm_cmpgt_epi32	N/A	N/A	N/A	A	N/A

_mm_cmplt_epi8	N/A	N/A	N/A	A	N/A
_mm_cmplt_epi16	N/A	N/A	N/A	A	N/A
_mm_cmplt_epi32	N/A	N/A	N/A	A	N/A
_mm_cvtsi32_si128	N/A	N/A	N/A	A	N/A
_mm_cvtsi128_si32	N/A	N/A	N/A	A	N/A
_mm_packs_epi16	N/A	N/A	N/A	A	N/A
_mm_packs_epi32	N/A	N/A	N/A	A	N/A
_mm_packus_epi16	N/A	N/A	N/A	A	N/A
_mm_extract_epi16	N/A	N/A	N/A	A	N/A
_mm_insert_epi16	N/A	N/A	N/A	A	N/A
_mm_movemask_epi8	N/A	N/A	N/A	A	N/A
_mm_shuffle_epi32	N/A	N/A	N/A	A	N/A
_mm_shufflehi_epi16	N/A	N/A	N/A	A	N/A
_mm_shufflelo_epi16	N/A	N/A	N/A	A	N/A
_mm_unpackhi_epi8	N/A	N/A	N/A	A	N/A
_mm_unpackhi_epi16	N/A	N/A	N/A	A	N/A
_mm_unpackhi_epi32	N/A	N/A	N/A	A	N/A
_mm_unpackhi_epi64	N/A	N/A	N/A	A	N/A
_mm_unpacklo_epi8	N/A	N/A	N/A	A	N/A
_mm_unpacklo_epi16	N/A	N/A	N/A	A	N/A
_mm_unpacklo_epi32	N/A	N/A	N/A	A	N/A
_mm_unpacklo_epi64	N/A	N/A	N/A	A	N/A
_mm_move_epi64	N/A	N/A	N/A	A	N/A
_mm_movpi64_epi64	N/A	N/A	N/A	A	N/A
_mm_movepi64_pi64	N/A	N/A	N/A	A	N/A
_mm_load_si128	N/A	N/A	N/A	A	N/A
_mm_loadu_si128	N/A	N/A	N/A	A	N/A
_mm_loadl_epi64	N/A	N/A	N/A	A	N/A
_mm_set_epi64	N/A	N/A	N/A	A	N/A
_mm_set_epi32	N/A	N/A	N/A	A	N/A
_mm_set_epi16	N/A	N/A	N/A	A	N/A
_mm_set_epi8	N/A	N/A	N/A	A	N/A
_mm_set1_epi64	N/A	N/A	N/A	A	N/A
_mm_set1_epi32	N/A	N/A	N/A	A	N/A

_mm_set1_epi16	N/A	N/A	N/A	A	N/A
_mm_set1_epi8	N/A	N/A	N/A	A	N/A
_mm_setr_epi64	N/A	N/A	N/A	A	N/A
_mm_setr_epi32	N/A	N/A	N/A	A	N/A
_mm_setr_epi16	N/A	N/A	N/A	A	N/A
_mm_setr_epi8	N/A	N/A	N/A	A	N/A
_mm_setzero_si128	N/A	N/A	N/A	A	N/A
_mm_store_si128	N/A	N/A	N/A	A	N/A
_mm_storeu_si128	N/A	N/A	N/A	A	N/A
_mm_storel_epi64	N/A	N/A	N/A	A	N/A
_mm_maskmoveu_si128	N/A	N/A	N/A	A	N/A
_mm_stream_pd	N/A	N/A	N/A	A	N/A
_mm_stream_si128	N/A	N/A	N/A	A	N/A
_mm_clflush	N/A	N/A	N/A	A	N/A
_mm_lfence	N/A	N/A	N/A	A	N/A
_mm_mfence	N/A	N/A	N/A	A	N/A
_mm_stream_si32	N/A	N/A	N/A	A	N/A
_mm_pause	N/A	N/A	N/A	A	N/A

インテル® C++ クラス・ライブラリ

クラス・ライブラリの概要

クラス・ライブラリの概要

インテル® C++ クラス・ライブラリを使用すると SIMD (Single-Instruction, Multiple-Data) 演算を実行できます。SIMD 演算の原理は、マイクロプロセッサ・アーキテクチャを並列処理に活用することです。並列処理を使用すると、より少ないクロックサイクル数で、より高いデータ・スループットが得られます。その目的は、オーディオ、ビデオ、グラフィックなど複雑で大量の計算を必要とするデータ・ビット・ストリームの処理効率を高めることです。

ハードウェアとソフトウェアの要件

クラス・ライブラリを使用するには、インテル® C++ コンパイラ (バージョン 4.0 以上) をインストールしなければなりません。インテル C++ クラス・ライブラリとは、次の表に示すように、各種インテル® プロセッサで使用できるすべての拡張命令の中からいくつか選び出した関数のことです。

クラス・ライブラリを使用するプロセッサの必要条件

ヘッダファイル	拡張命令セット	対応プロセッサ
ivec.h	MMX® テクノロジ	MMX® テクノロジ Pentium® プロセッサ、Pentium II プロセッサ、Pentium Pentium III プロセッサ、Pentium 4 プロセッサ、インテル® Xeon™ プロセッサ、および Itanium® プロセッサ
fvec.h	ストリーミング SIMD 拡張命令	Pentium III プロセッサ、Pentium 4 プロセッサ、インテル Xeon プロセッサ、および Itanium プロセッサ
dvec.h	ストリーミング SIMD 拡張命令 2	Pentium 4 プロセッサおよびインテル Xeon プロセッサのみ

クラスについて

SIMD 演算用のインテル® C++ クラス・ライブラリには次のものがあります:

- 整数ベクトル (Ivec) クラス
- 浮動小数点ベクトル (Fvec) クラス

これら演算の定義は、ivec.h、fvec.h、および dvec.h という 3 つのヘッダファイルに記述されています。クラス自体がこのように区分されているわけではありません。各クラスの名前は、基本となる演算の種類に従って付けられたものです。ヘッダファイルはアーキテクチャの種類に従って区分されています:

- ivec.h は MMX® テクノロジ命令のアーキテクチャ専用です
- fvec.h はストリーミング SIMD 拡張命令のアーキテクチャ専用です
- dvec.h はストリーミング SIMD 拡張命令 2 のアーキテクチャ専用です

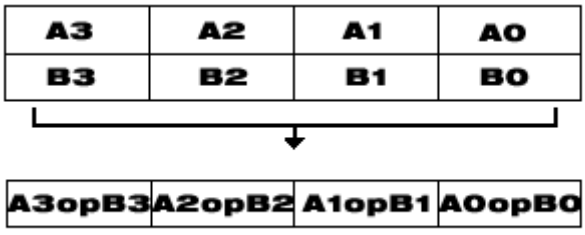
ストリーミング SIMD 拡張命令 2 の組込み関数は Itanium® ベースのシステムでは使用できません。ヘッダファイル mmclass.h には、Itanium アーキテクチャで使用可能な各クラスが含まれています。

本書は、インテル® アーキテクチャ用のコードを作成するプログラマに向けて書かれたものですが、中でも SIMD 命令を活用するコードを作成するプログラマを対象としています。C++ と C++ の各クラスの使用に精通している必要があります。

ライブラリの詳細

SIMD 演算用のインテル® C++ クラス・ライブラリは、「クラス・ライブラリを使用するプロセッサの必要条件」に示した各種プロセッサ用の基本命令を利用するための便利なインターフェイスとなります。プロセッサ命令のこのような拡張機能によって、SIMD (single instruction-multiple data) 手法を用いた並列処理が可能になります。SIMD のデータフローを次の図に示します。

SIMD のデータフロー



特にこの命令では、命令 1 つで演算が 4 回実行できるため、効率が 4 倍改善されます。

このような新しいプロセッサ命令は、インライン・アセンブリ、組込み関数、または C++ SIMD クラスのいずれを使用しても実装できます。その 3 種類のインターフェイスについて、32 ビット浮動小数点値を 4 つ加算するのに必要なコーディングを比較してみてください:

インライン・アセンブリ、組込み関数、クラス・ライブラリの比較

インライン・アセンブリ	組込み関数	SIMD クラス・ライブラリ
`... __m128 a,b,c; __asm{ movaps xmm0,b movaps xmm1,c addps xmm0,xmm1 movaps a, xmm0 } ...`	`#include <mmintrin.h> ... __m128 a,b,c; a = _mm_add_ps(b,c); ...`	`#include <fvec.h> ... F32vec4 a,b,c; a = b +c; ...`

上の表は、単精度浮動小数点値を 2 つ加算するコードについて、インライン・アセンブリ、組込み関数、および SIMD クラス・ライブラリを用いた場合をそれぞれ示したものです。インテル C++ SIMD クラス・ライブラリでコーディングをするのがいかに簡単かがわかります。キー入力の数が減り、コードの行数が減るだけでなく、表記についても C++ の標準表記と似ているため、他の手法よりも簡単に実装できます。

C++ クラスと SIMD 演算

SIMD 演算用の C++ クラスは、配列 (ベクトルデータ) を並列処理するときに使用するのが基本です。例として、2 つのベクトル `A` と `B` の加算を考えてみます。各ベクトルは 4 つの要素で構成されているとします。整数ベクトル (`Ivec`) クラスを使用して、各配列から取り出した要素 `A[i]` と `B[i]` を下の例のように加算します。

ループを使用して複数の要素を加算するときの一般的な方法

```
short a[4], b[4], c[4];
for (i=0; i<4; i++) /* needs four iterations */
c[i] = a[i] + b[i]; /* returns c[0], c[1], c[2], c[3] *
```

次の例を見ると、`Ivec` クラスを使用すれば 1 回の演算で同じ結果が得られることがわかります。

Ivec クラスを使用して複数の要素を加算する SIMD 手法

```
sIs16vec4 ivecA, ivecB, ivec C; /*needs one iteration */
ivecC = ivecA + ivecB; /*returns ivecC0, ivecC1, ivecC2,
ivecC3 */
```

使用できるクラス

並列処理は、C++ では通常それほど簡単には実装できませんが、インテル® C++ SIMD クラスを使用すればそれが可能です。次の表は、インテル C++ SIMD クラスでクラスとライブラリがどのように使用されるかを列挙したものです。

SIMD ベクトルクラス

命令セット	クラス	符号の有無	データ型	サイズ	要素数	ヘッダファイル
MMX® テクノロジ (IA-32 および Itanium® ベースのシステムで利用可)	I64vec1	不定	__m64	64	1	ivec.h
	I32vec2	不定	int	32	2	ivec.h
	Is32vec2	符号付き	int	32	2	ivec.h
	Iu32vec2	符号なし	int	32	2	ivec.h
	I16vec4	不定	short	16	4	ivec.h
	Is16vec4	符号付き	short	16	4	ivec.h
	Iu16vec4	符号なし	short	16	4	ivec.h
	I8vec8	不定	char	8	8	ivec.h
	Is8vec8	符号付き	char	8	8	ivec.h
	Iu8vec8	符号なし	char	8	8	ivec.h
ストリーミング SIMD 拡張命令 (IA-32 および Itanium ベースのシステムで利用可)	F32vec4	符号付き	float	32	4	fvec.h
	F32vec1	符号付き	float	32	1	fvec.h
ストリーミング SIMD 拡張命令 2 (IA-32 ベースのシステムでのみ利用可)	F64vec2	符号付き	double	64	2	dvec.h
	I128vec1	不定	__m128i	128	1	dvec.h
	I64vec2	不定	long int	64	4	dvec.h
	Is64vec2	符号付き	long int	64	4	dvec.h
	Iu64vec2	符号なし	long int	32	4	dvec.h
	I32vec4	不定	int	32	4	dvec.h
	Is32vec4	符号付き	int	32	4	dvec.h
	Iu32vec4	符号なし	int	32	4	dvec.h
	I16vec8	不定	int	16	8	dvec.h

	Is16vec8	符号付き	int	16	8	dvec.h
	Iu16vec8	符号なし	int	16	8	dvec.h
	I8vec16	不定	char	8	16	dvec.h
	Is8vec16	符号付き	char	8	16	dvec.h
	Iu8vec16	符号なし	char	8	16	dvec.h

ほとんどのクラスは、どのデータ型についても同じような機能を持っていて、利用できるすべての組込み関数で表現されています。ただし一部の機能については、データ型が変わるときにその機能を維持しようとするとパフォーマンスが下がる場合があるため、個々のクラスからは除外しています。

注

即値をとるためにクラスの中に簡単に表現できない組込み関数は実装していません。(例えば、_mm_shuffle_ps, _mm_shuffle_pi16, _mm_extract_pi16, _mm_insert_pi16)

ヘッダファイルを使用したクラスへのアクセス

必要なクラス・ヘッダファイルは、インテル C++ コンパイラと一緒にインクルード・ディレクトリにインストールされます。各クラスを有効にするときは、次の表に示すように、プログラム・ファイルの中で #include ディレクティブを使用してください。

クラスを有効にするためのインクルード・ディレクティブ

拡張命令セット	インクルード・ディレクティブ
MMX テクノロジ	#include <ivec.h>
ストリーミング SIMD 拡張命令	#include <fvec.h>
ストリーミング SIMD 拡張命令 2	#include <dvec.h>

上の表で、dvec.h は fvec.h の内容を含みます。同様に fvec.h は ivec.h の内容を含みます。Ivec クラスと Fvec クラスの両方を使用する場合は、fvec.h をインクルードするだけで済みます。同様に、ストリーミング SIMD 拡張命令 2 を利用する場合に、その 3 つとも含むすべてのクラスを使用するときは、dvec.h ファイルをインクルードするだけで済みます。

使用上の注意事項

C++ クラスの使用方法については、一般的なガイドラインがいくつかあります。クラスごとの使用規則については、「整数ベクトルクラス」および「浮動小数点ベクトルクラス」を参照してください。

MMX テクノロジ・レジスタの消去

Ivec クラスと Fvec クラスを同時に使用している場合は、Ivec クラスから呼び出される MMX 命令と、Fvec クラスから呼び出されるインテル x87 アーキテクチャ浮動小数点命令がプログラムの中で混在していることがあります。次の Fvec 関数の中には浮動小数点命令が含まれています。

- fvec コンストラクタ
- デバッグ関数 (cout および要素アクセス)
- rsqrt_nr

注

MMX テクノロジ・レジスタは浮動小数点レジスタ上にエイリアス化されています。したがって、x87 浮動小数点命令を発行する前には、EMMS 命令の組込み関数を使用して MMX テクノロジ・ステートを消去する必要があります。次に例を示します。

`ivecA = ivecA & ivecB;`	`/* Ivec logical operation that uses MMX instructions */`
`empty ();`	`/* clear state */`
`cout << f32vec4a;`	`/* F32vec4 operation that uses x87 floating-point instructions */`

警告

MMX テクノロジ・レジスタを消去しないとレジスタステートが不正な状態になります。そのため、処理が不正確になったり、パフォーマンスが低下したりすることがあります。

EMMS 命令のガイドラインに従う

「EMMS 命令を使用する際のガイドライン」に従うことを強くお勧めします。Ivec クラスを使用してコーディングを行う前に、このトピックを参照してください。

機能

C++ SIMD の各クラスには、次のような基本的な機能があります:

- 演算
- 水平データ移動
- 分岐の圧縮/削除
- キャッシング・ヒント

目的の結果を得るためには、これら諸機能についてそれぞれ理解し、これら諸機能が互いにどのように作用し合うのかを理解することが重要です。

演算

SIMD C++ の各クラスは、シフトや飽和をはじめとしてほとんどの算術演算用の垂直演算子に対応しています。

演算子の種類としては、+、-、*、/、逆数 (rcp、rcp_nr)、平方根 (sqrt)、平方根の逆数 (rsqrt、rsqrt_nr) があります。

rcp および rsqrt という演算は、非常に短いレイテンシで近似値の計算ができる新しい命令のことで、最低でも 12 ビットの精度の結果が得られます。rcp_nr および rsqrt_nr という演算は、ソフトウェア的にいくつか改良がなされていて、パフォーマンスにはごく小さい影響しか与えずに精度の高い近似値が得られます。("nr" は、近似値を使用してパフォーマンスを改善する数学的手法の 1 つである "Newton-Raphson 法" の頭文字です)。

水平データのサポート

C++ SIMD の各クラスは、一部の算術演算については水平処理もできます。"水平" とは、1 つのベクトルに含まれているすべての要素について、横方向に端から端まで計算することを指します。一方、"垂直" とは、2 つの異なるベクトルについて、縦方向に要素単位で計算していくことを指します。

例えば add_horizontal、unpack_low、pack_sat という各関数は、水平データに対応しています。このように水平データに対応していると、SIMD 命令の能力をすべては利用できない一部のアルゴリズムも使用できるようになります。

シャッフル組込み関数も水平データフローのもう 1 つの例です。シャッフル組込み関数は、即値引数を使用するため、C++ の各クラスの中には示されていません。ただし、C++ クラスでは、シャッフル組込み関数を他の C++ 関数と混在できます。次に例を示します:

```
F32vec4 fveca, fvecb, fvecd;
fveca += fvecb;
fvecd = _mm_shuffle_ps(fveca,fvecb,0);
```

一般に、水平データフローを用いると、どの命令の場合も、実装に際しては少し無駄な部分ができます。できる限り、アルゴリズムを実装するときは水平機能は使用しないでください。

分岐の圧縮/削除

SIMD アーキテクチャでの分岐処理は、複雑で面倒なことが多く、場合によっては分岐予測の精度が下がったり、コードのサイズが増えたりします。SIMD C++ クラスには、論理演算、最大値/最小値関数、条件付き選択、比較を使うことによって分岐をなくすための関数がいくつか用意されています。次の例について考えてみます:

```
short a[4], b[4], c[4];
for (i=0; i<4; i++)
c[i] = a[i] > b[i] ? a[i] : b[i];
```

この演算は `i` の値とは無関係です。`i` の値が変わるごとに、その結果は、実際の値に応じて `A` か `B` のいずれかになります。分岐をなくす簡単な方法の 1 つは、次のように `select_gt` 関数を使うことです:

```
Is16vec4 a, b, c
c = select_gt(a, b, a, b)
```

キャッシング・ヒント

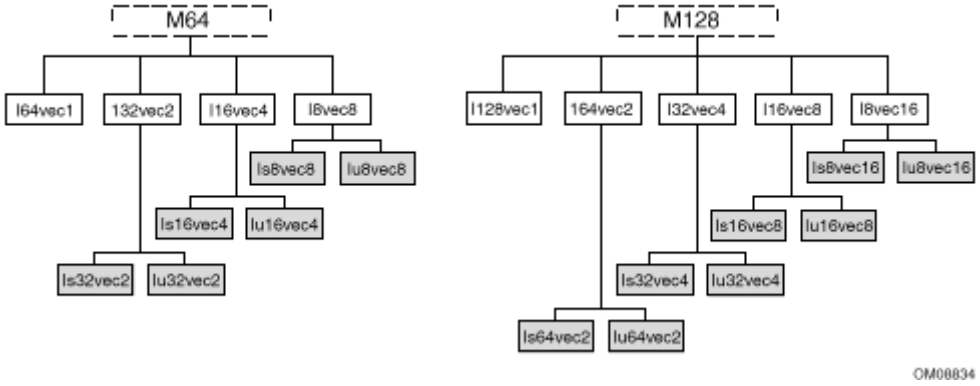
ストリーミング SIMD 拡張命令には、プリフェッチとストリーミング・ヒントという機能があります。データをプリフェッチすると、メモリ・レイテンシの影響を最小限に抑えられます。ストリーミング・ヒントを使用すると、キャッシュしないほうが望ましいデータを示すことができます。これにより、キャッシュしたほうが望ましいデータのパフォーマンスが上がります。

整数ベクトルクラス

概要: 整数ベクトルクラス

`Ivec` クラスは、さまざまなサイズの整数ベクトルを用いる SIMD 処理へのインターフェイスの機能を提供します。このクラスの派生関係図を次に示します。

Ivec クラスの派生関係図



M64 および M128 という両クラスが __m64 および __m128i という両データ型を定義しています。この 2 つのクラスから、残りの Ivec クラスが派生します。第 1 世代の子クラスは、128、64、32、16、8 というビットサイズだけで区分され、それぞれ I128vec1、I64vec1、164vec2、I32vec2、I32vec4、I16vec4、I16vec8、I8vec16、I8vec8 というクラスになります。そのうちの I128vec1 と I64vec1 以外は、符号の有無と飽和処理の有無を指定する必要があります。

警告

M64 データ型と M128 データ型は混在させないでください。そうしないと、予期しない動作をします。

符号の有無は、クラス名の中の s と u という文字で表します:

```
Is64vec2
Iu64vec2
Is32vec4
Iu32vec4
Is16vec8
Iu16vec8
Is8vec16
Iu8vec16
Is32vec2
Iu32vec2
Is16vec4
Iu16vec4
Is8vec8
Iu8vec8
```

用語、規則、および構文

本章では、各クラスとそれに関連した処理について説明するときに特殊な用語と構文を用いています。以下で、その説明をします。

Ivec クラスの構文規則

クラス名は、データ型、符号の有無、ビットサイズ、要素数を表現したものです。一般的な形式で表すと次のようになります:

```
<type><signedness><bits>vec<elements>

{ F | I } { s | u } { 64 | 32 | 16 | 8 } vec { 8 | 4 | 2 | 1 }
```

各アイテムの意味は次のとおりです:

type	浮動小数点 (F) または整数 (I) を示します。
signedness	符号付き (s) または符号なし (u) を示します。Ivec クラスの場合は、このフィールドが空のままだと中間クラスを表します。符号なしの Fvec クラスはないため、Fvec クラスの場合、このフィールドは空です。
bits	要素あたりのビット数です。
elements	要素の個数です。

特殊な用語と規則

本書では、各クラスと演算についてその機能と特性を定義するときに次の用語を用いています。

- **最も近い共通の親クラス**: サイズの同じ 2 種類のクラスの中間クラスまたは親クラスです。例えば、Iu8vec8 と Is8vec8 の「最も近い共通の親クラス」は I8vec8 です。同様に、Iu8vec8 と I16vec4 の「最も近い共通の親クラス」は M64 です。
- **キャスト**: クラスのデータ型を変更します。データ型の異なる複数のオペランドを使用して演算を行う場合は、その戻り値を同じ 1 種類のデータ型にする必要があります。したがって、1 種類以上のデータ型を同じ種類の規定のデータ型に変換する必要があります。この変換処理のことを ″型キャスト″ と言います。型キャストは自動的に行われる場合もありますが、特殊な構文を使用して明示的に行う必要もあります。
- **演算子の多重定義**: 何らかのクラスのデータ型をユーザが 1 つ定義し、その 1 つのデータ型に対して複数の演算子を利用できる機能です。いったん変数を宣言すれば、加算、減算、乗算はもとより、複数の演算を連続して実行でき

ます。どのクラスファミリも、規定の範囲の演算子が利用できますが、ヘッダファイルに定義されている、型キャストと演算子の多重定義に関する規則および制約に従う必要があります。本書では、型キャストや演算子の多重定義などの規則に次の表記を使用しています。

クラス名の表記法

クラス名	説明
I[s|u][N]vec[N]	I128vec1 以外および I64vec1 以外の任意の値
I64vec1	__m64 データ型
I[s|u]64vec2	2 つの 64 ビット値 (符号付き、または符号なし)
I[s|u]32vec4	4 つの 32 ビット値 (符号付き、または符号なし)
I[s|u]8vec16	8 つの 16 ビット値 (符号付き、または符号なし)
I[s|u]16vec8	16 の 8 ビット値 (符号付き、または符号なし)
I[s|u]32vec2	2 つの 32 ビット値 (符号付き、または符号なし)
I[s|u]16vec4	4 つの 16 ビット値 (符号付き、または符号なし)
I[s|u]8vec8	8 つの 8 ビット値 (符号付き、または符号なし)

演算子の規則

Ivec クラスで演算子を使用するときは、次の構文規則のいずれかに従う必要があります:

```
[ Ivec_Class ] R = [ Ivec_Class ] A
[ operator ][ Ivec_Class ] B
```

例 1:
```
I64vec1 R = I64vec1 A & I64vec1 B;
```

```
[ Ivec_Class ] R =[ operator ] ([ Ivec_Class ]
A,[ Ivec_Class ] B)
```

例 2:
```
I64vec1 R = andnot(I64vec1 A, I64vec1 B);
```

```
[ Ivec_Class ] R [ operator ]= [ Ivec_Class ] A
```

例 3:
```
I64vec1 R &= I64vec1 A;
```

[operator] は演算子 (例えば、&、|、^)

[Ivec_Class] は Ivec クラス

R、A、B は、該当する Ivec クラスを使用して宣言された変数

次の表は、符号とサイズを対象とした自動的型キャストと明示的型キャストを列挙したものです。"明示的" は、異なるデータ型をいくつか混在させるときは必ず型キャストを明示的に指定しなければならないという意味です。"自動的" は、複数のデータ型を混在させたときにコンパイラが自動的に型キャストを行うという意味です。

主要演算子の規則一覧

演算子	符号の型キャスト	サイズの型キャスト	型キャストに関するその他の要件
代入	N/A	N/A	N/A
論理	自動的	自動的 (左辺)	代入演算子の右辺に論理式以外の式が使用されていて、かつその中に、異なるデータ型が混在している場合は、明示的な型キャストが必要です。
加算と減算	自動的	明示的	N/A
乗算	自動的	明示的	N/A
シフト	自動的	明示的	算術演算シフトを確実に行うには型キャストが必要です。
比較	自動的	明示的	「<」または「>」の比較を行う場合は、符号付きクラスについては明示的な型キャストが必要です。
条件付き選択	自動的	明示的	「<」または「>」の比較を行う場合は、符号付きクラスについては明示的な型キャストが必要です。

データの宣言と初期化

次の表は、すべてのクラスサイズについて、コンストラクタの宣言とデータ型の初期化の例を列挙したものです。どの値についても、最上位要素が左側、最下位要素が右側で初期化されます。

Ivec クラスのデータ型の宣言と初期化

操作	クラス	構文
宣言	M128	`I128vec1 A; Iu8vec16 A;`
宣言	M64	`I64vec1 A; Iu8vec16 A;`
__m128 の初期化	M128	`I128vec1 A(__m128 m); Iu16vec8(__m128 m);`
__m64 の初期化	M64	`I64vec1 A(__m64 m);Iu8vec8 A(__m64 m);`

__int64 の初期化	M64	I64vec1 A = __int64 m; Iu8vec8 A =__int64 m;
int i の初期化	M64	I64vec1 A = int i; Iu8vec8 A = int i;
int の初期化	I32vec2	I32vec2 A(int A1, int A0); Is32vec2 A(signed int A1, signed int A0); Iu32vec2 A(unsigned int A1, unsigned int A0);
int の初期化	I32vec4	I32vec4 A(short A3, short A2, short A1, short A0); Is32vec4 A(signed short A3, ..., signed short A0); Iu32vec4 A(unsigned short A3, ..., unsigned short A0);
short int の初期化	I16vec4	I16vec4 A(short A3, short A2, short A1, short A0); Is16vec4 A(signed short A3, ..., signed short A0); Iu16vec4 A(unsigned short A3, ..., unsigned short A0);
short int の初期化	I16vec8	I16vec8 A(short A7, short A6, ..., short A1, short A0); Is16vec8 A(signed A7, ..., signed short A0); Iu16vec8 A(unsigned short A7, ..., unsigned short A0);
char の初期化	I8vec8	I8vec8 A(char A7, char A6, ..., char A1, char A0); Is8vec8 A(signed char A7, ..., signed char A0); Iu8vec8 A(unsigned char A7, ..., unsigned char A0);
char の初期化	I8vec16	I8vec16 A(char A15, ..., char A0); Is8vec16 A(signed char A15, ..., signed char A0); Iu8vec16 A(unsigned char A15, ..., unsigned char A0);

代入演算子

どの Ivec オブジェクトも別の Ivec オブジェクトに代入できます。Ivec オブジェクトから別の Ivec オブジェクトに代入するときの変換は自動的に行われます。

代入演算子の例

```
Is16vec4 A;

Is8vec8 B;

I64vec1 C;

A = B; /* assign Is8vec8 to Is16vec4 */
```

```
B = C; /* assign I64vec1 to Is8vec8 */

B = A & C; /* assign M64 result of '&' to Is8vec8 */
```

論理演算子

論理演算子では、次の表に列挙した記号と組込み関数を使用します。

ビット単位演算	演算子の記号		構文の使用方法		対応する組込み関数
	標準	代入付き	標準	代入付き	
AND	&	&=	R = A & B	R &= A	_mm_and_si64 _mm_and_si128
OR	|	|=	R = A | B	R |= A	_mm_and_si64 _mm_and_si128
XOR	^	^=	R = A^B	R ^= A	_mm_and_si64 _mm_and_si128
ANDNOT	andnot	N/A	R = A andnot B	N/A	_mm_and_si64 _mm_and_si128

論理演算子と例外

```
/* A and B converted to M64. Result assigned to Iu8vec8.*/

I64vec1 A;
Is8vec8 B;
Iu8vec8 C;
C = A & B;

/* Same size and signedness operators return the nearest
common ancestor.*/

I32vec2 R = Is32vec2 A ^ Iu32vec2 B;

/* A&B returns M64, which is cast to Iu8vec8.*/

C = Iu8vec8(A&B)+ C;
```

A と B が同じクラスの場合、戻り値はそれと同じ型になります。A と B とが別のクラスの場合、戻り値は、両者に共通する親データ型のうち最も近いデータ型になります。

クラスを複数組み合わせた場合に論理演算子の戻り値がどうなるかを次の表に示します。これは、A と B が別のクラスの場合に適用されるものです。

Ivec 論理演算子の多重定義

戻り値 (R)	AND	OR	XOR	NAND	オペランド A	オペランド B
I64vec1 R	&	|	^	andnot	I[s|u]64vec2 A	I[s|u]64vec2 B
I64vec2 R	&	|	^	andnot	I[s|u]64vec2 A	I[s|u]64vec2 B
I32vec2 R	&	|	^	andnot	I[s|u]32vec2 A	I[s|u]32vec2 B
I32vec4 R	&	|	^	andnot	I[s|u]32vec4 A	I[s|u]32vec4 B
I16vec4 R	&	|	^	andnot	I[s|u]16vec4 A	I[s|u]16vec4 B
I16vec8 R	&	|	^	andnot	I[s|u]16vec8 A	I[s|u]16vec8 B
I8vec8 R	&	|	^	andnot	I[s|u]8vec8 A	I[s|u]8vec8 B
I8vec16 R	&	|	^	andnot	I[s|u]8vec16 A	I[s|u]8vec16 B

代入付き論理演算子の場合は次表のように、戻り値 R は常に、事前に宣言した値 R と同じデータ型になります。

代入付き Ivec 論理演算子の多重定義

戻り値の型	左辺 (R)	AND	OR	XOR	右辺 (任意の Ivec 型)
I128vec1	I128vec1 R	&=	|=	^=	I[s|u][N]vec[N] A;
I64vec1	I64vec1 R	&=	|=	^=	I[s|u][N]vec[N] A;
I64vec2	I64vec2 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]32vec4	I[x]32vec4 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]32vec2	I[x]32vec2 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]16vec8	I[x]16vec8 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]16vec4	I[x]16vec4 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]8vec16	I[x]8vec16 R	&=	|=	^=	I[s|u][N]vec[N] A;
I[x]8vec8	I[x]8vec8 R	&=	|=	^=	I[s|u][N]vec[N] A;

加算演算子と減算演算子

加算演算子と減算演算子は、右辺の各オペランドの符号が揃っていない場合は、共通の親クラスのうち最も近い親クラスを返します。使用方法の例と例外をいくつか次のコードに示します。

加算演算子および減算演算子の構文の使用例

```
/* Return nearest common ancestor type, I16vec4 */

Is16vec4 A;
```

```
Iu16vec4 B;

I16vec4 C;

C = A + B;

/* Returns type left-hand operand type */

Is16vec4 A;

Iu16vec4 B;

A += B;

B -= A;

/* Explicitly convert B to Is16vec4 */

Is16vec4 A,C;

Iu32vec24 B;

C = A + C;

C = A + (Is16vec4)B;
```

加算演算子、減算演算子、およびそれらに対応する組込み関数

操作	記号	構文	対応する組込み関数
加算	+ +=	R = A + B R += A	_mm_add_epi64 _mm_add_epi32 _mm_add_epi16 _mm_add_epi8 _mm_add_pi32 _mm_add_pi16 _mm_add_pi8
減算	- -=	R = A - B R -= A	_mm_sub_epi64 _mm_sub_epi32 _mm_sub_epi16 _mm_sub_epi8 _mm_sub_pi32 _mm_sub_pi16 _mm_sub_pi8

次の表は、右辺の各オペランドの符号が異なっている場合に、クラスを複数組み合わせたときの、加算と減算の戻り値を列挙したものです。この 2 つのオペランドは同じサイズでなければなりません。サイズが異なる場合は、型キャストを明示的に示す必要があります。

加算演算子と減算演算子の多重定義

戻り値	使用できる演算子		右辺のオペランド	
R	加算	減算	A	B
I64vec2 R	+	-	I[s|u]64vec2 A	I[s|u]64vec2 B
I32vec4 R	+	-	I[s|u]32vec4 A	I[s|u]32vec4 B
I32vec2 R	+	-	I[s|u]32vec2 A	I[s|u]32vec2 B
I16vec8 R	+	-	I[s|u]16vec8 A	I[s|u]16vec8 B
I16vec4 R	+	-	I[s|u]16vec4 A	I[s|u]16vec4 B
I8vec8 R	+	-	I[s|u]8vec8 A	I[s|u]8vec8 B
I8vec16 R	+	-	I[s|u]8vec2 A	I[s|u]8vec16 B

次の表は、代入付きの加算演算子と減算演算子の各種オペランドについて、戻り値のデータ型を列挙したものです。戻り値のサイズと符号の有無は、左辺のオペランドによって決まります。右辺のオペランドは左辺のオペランドと同じサイズでなければなりません。サイズが異なる場合は、明示的な型キャストを行う必要があります。

代入付きの加算演算子と減算演算子

戻り値 (R)	左辺 (R)	加算	減算	右辺 (A)
I[x]32vec4	I[x]32vec2 R	+=	-=	I[s|u]32vec4 A;
I[x]32vec2 R	I[x]32vec2 R	+=	-=	I[s|u]32vec2 A;
I[x]16vec8	I[x]16vec8	+=	-=	I[s|u]16vec8 A;
I[x]16vec4	I[x]16vec4	+=	-=	I[s|u]16vec4 A;
I[x]8vec16	I[x]8vec16	+=	-=	I[s|u]8vec16 A;
I[x]8vec8	I[x]8vec8	+=	-=	I[s|u]8vec8 A;

乗算演算子

乗算演算子では、I[s|u]16vec4 または I[s|u]16vec8 のいずれかのクラスのデータ型しか演算に使用できません。また、戻り値もそのいずれかのデータ型になります。次に例を示します。

乗算演算子の構文の使用例

```
/* Explicitly convert B to Is16vec4 */

Is16vec4 A,C;
```

```
Iu32vec2 B;

C = A * C;

C = A * (Is16vec4)B;

/* Return nearest common ancestor type, I16vec4 */

Is16vec4 A;

Iu16vec4 B;

I16vec4 C;

C = A + B;

/* The mul_high and mul_add functions take Is16vec4 data
only */

Is16vec4 A,B,C,D;

C = mul_high(A,B);

D = mul_add(A,B);
```

対応する組込み関数と乗算演算子

記号		構文の使用方法	組込み関数
*	*=	R = A * B R *= A	_mm_mullo_pi16 _mm_mullo_epi16
mul_high	N/A	R = mul_high(A, B)	_mm_mulhi_pi16 _mm_mulhi_epi16
mul_add	N/A	R = mul_high(A, B)	_mm_madd_pi16 _mm_madd_epi16

次の表に示すように、この乗算演算子の戻り値は常に、共通する親データ型のうち最も近いデータ型になります。2 つのオペランドのサイズは 16 ビットでなければなりません。それ以外の場合は、型キャストを明示的に使用する必要があります。

乗算演算子の多重定義

R	乗算	A	B
I16vec4 R	*	I[s|u]16vec4 A	I[s|u]16vec4 B
I16vec8 R	*	I[s|u]16vec8 A	I[s|u]16vec8 B
Is16vec4 R	mul_add	Is16vec4 A	Is16vec4 B
Is16vec8	mul_add	Is16vec8 A	Is16vec8 B

Is32vec2 R	mul_high	Is16vec4 A	Is16vec4 B
Is32vec4 R	mul_high	s16vec8 A	Is16vec8 B

次の表は、代入付き乗算演算子を用いた場合の戻り値のデータ型を列挙したものです。オペランドのサイズはすべて 16 ビットでなければなりません。オペランドのサイズが正しくない場合は、明示的な型キャストを使用する必要があります。

代入付き乗算

戻り値 (R)	左辺 (R)	乗算	右辺 (A)
I[x]16vec8	I[x]16vec8	*=	I[s|u]16vec8 A;
I[x]16vec4	I[x]16vec4	*=	I[s|u]16vec4 A;

シフト演算子

右シフトの引数は、整数値か Ivec 値であれば何でもかまわず、暗黙的に M64 データ型に変換されます。<< という演算子の 1 番目のオペランドまたは左側のオペランドには、I[s|u]8vec[8|16] 以外のどの型でも使用できます。

シフト演算子の構文の使用例

```
/* Automatic size and sign conversion */

Is16vec4 A,C;

Iu32vec2 B;

C = A;

/* A&B returns I16vec4, which must be cast to Iu16vec4

to ensure logical shift, not arithmetic shift */

Is16vec4 A, C;

Iu16vec4 B, R;

R = (Iu16vec4)(A & B) C;

/* A&B returns I16vec4, which must be cast to Is16vec4

to ensure arithmetic shift, not logical shift */

R = (Is16vec4)(A & B) C;
```

シフト演算子と対応する組込み関数

操作	記号	構文の使用方法	組込み関数
左シフト	<< &=	R = A << B R &= A	_mm_sll_si64 _mm_slli_si64 _mm_sll_pi32 _mm_slli_pi32 _mm_sll_pi16 _mm_slli_pi16
右シフト	>>	R = A >> B R >>= A	_mm_srl_si64 _mm_srli_si64 _mm_srl_pi32 _mm_srli_pi32 _mm_srl_pi16 _mm_srli_pi16 _mm_sra_pi32 _mm_srai_pi32 _mm_sra_pi16 _mm_srai_pi16

符号付きのデータ型を右にシフトするときは算術シフトを使用します。符号なしクラスおよび中間クラスのときはすべて、論理シフトが使用されます。次の表は、最初の引数の型によって戻り値の型がどう決まるかを示したものです。

シフト演算子の多重定義

操作	R	右シフト		左シフト		A	B
論理	I64vec1	>>	>>=	<<	<<=	I64vec1 A;	I64vec1 B;
論理	I32vec2	>>	>>=	<<	<<=	I32vec2 A	I32vec2 B;
算術	Is32vec2	>>	>>=	<<	<<=	Is32vec2 A	I[s|u][N]vec[N] B;
論理	Iu32vec2	>>	>>=	<<	<<=	Iu32vec2 A	I[s|u][N]vec[N] B;
論理	I16vec4	>>	>>=	<<	<<=	I16vec4 A	I16vec4 B
算術	Is16vec4	>>	>>=	<<	<<=	Is16vec4 A	I[s|u][N]vec[N] B;
論理	Iu16vec4	>>	>>=	<<	<<=	Iu16vec4 A	I[s|u][N]vec[N] B;

比較演算子

「＝」および「≠」の比較については、オペランドは符号が異なっていてもかまいませんが、サイズは同じでなければなりません。「＜」および「＞」の比較については、オペランドは符号もサイズも同じでなければなりません。

比較演算子の構文の使用例

```
/* The nearest common ancestor is returned for compare
```

```
for equal/not-equal operations */
Iu8vec8 A;
Is8vec8 B;
I8vec8 C;
C = cmpneq(A,B);
/* Type cast needed for different-sized elements for
equal/not-equal comparisons */
Iu8vec8 A, C;
Is16vec4 B;
C = cmpeq(A,(Iu8vec8)B);
/* Type cast needed for sign or size differences for
less-than and greater-than comparisons */
Iu16vec4 A;
Is16vec4 B, C;
C = cmpge((Is16vec4)A,B);
C = cmpgt(B,C);
```

比較演算子と対応する組込み関数

比較の条件	演算子	構文	組込み関数	
等しい	cmpeq	R = cmpeq(A, B)	_mm_cmpeq_pi32 _mm_cmpeq_pi16 _mm_cmpeq_pi8	
等しくない	cmpneq	R = cmpneq(A, B)	_mm_cmpeq_pi32 _mm_cmpeq_pi16 _mm_cmpeq_pi8	_mm_andnot_si64
より大きい	cmpgt	R = cmpgt(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8	
以上	cmpge	R = cmpge(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8	_mm_andnot_si64
より小さい	cmplt	R = cmplt(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8	

以下	cmple	R = cmple(A, B)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8	_mm_andnot_si64

比較演算子のオペランドは、そのサイズも符号も、次の「比較演算子の多重定義」の表に従っていなければなりません。

比較演算子の多重定義

R	比較条件	A	B
I32vec2 R	cmpeq cmpne	I[s|u]32vec2 B	I[s|u]32vec2 B
I16vec4 R		I[s|u]16vec4 B	I[s|u]16vec4 B
I8vec8 R		I[s|u]8vec8 B	I[s|u]8vec8 B
I32vec2 R	cmpgt cmpge cmplt cmple	Is32vec2 B	Is32vec2 B
I16vec4 R		Is16vec4 B	Is16vec4 B
I8vec8 R		Is8vec8 B	Is8vec8 B

条件付き選択演算子

条件付き選択演算子の場合は、3 番目と 4 番目のオペランドによって戻り値のデータ型が決まります。3 番目と 4 番目のオペランドがサイズが同じで符号が異なる場合、戻り値は共通の親データ型のうち最も近いデータ型になります。

条件付き選択演算子の構文の使用方法

```
/* Return the nearest common ancestor data type if third
and fourth operands are of the same size, but different
signs */

I16vec4 R = select_neq(Is16vec4, Is16vec4, Is16vec4,
Iu16vec4);

/* Conditional Select for Equality */

R0 := (A0 == B0) ? C0 : D0;

R1 := (A1 == B1) ? C1 : D1;

R2 := (A2 == B2) ? C2 : D2;

R3 := (A3 == B3) ? C3 : D3;
```

```
/* Conditional Select for Inequality */

R0 := (A0 != B0) ? C0 : D0;

R1 := (A1 != B1) ? C1 : D1;

R2 := (A2 != B2) ? C2 : D2;

R3 := (A3 != B3) ? C3 : D3;
```

条件付き選択演算子と対応する組込み関数

条件付き選択の条件	演算子	構文	対応する組込み関数	その他の組込み関数 (すべてに適用)
等しい	select_eq	R = select_eq(A, B, C, D)	_mm_cmpeq_pi32 _mm_cmpeq_pi16 _mm_cmpeq_pi8	_mm_and_si64 _mm_or_si64 _mm_andnot_si64
等しくない	select_neq	R = select_neq(A, B, C, D)	_mm_cmpeq_pi32 _mm_cmpeq_pi16 _mm_cmpeq_pi8	
より大きい	select_gt	R = select_gt(A, B, C, D)	_mm_cmpgt_pi32 _mm_cmpgt_pi16 _mm_cmpgt_pi8	
以上	select_ge	R = select_gt(A, B, C, D)	_mm_cmpge_pi32 _mm_cmpge_pi16 _mm_cmpge_pi8	
より小さい	select_lt	R = select_lt(A, B, C, D)	_mm_cmplt_pi32 _mm_cmplt_pi16 _mm_cmplt_pi8	
以下	select_le	R = select_le(A, B, C, D)	_mm_cmple_pi32 _mm_cmple_pi16 _mm_cmple_pi8	

条件付き選択演算のオペランドはすべて同じサイズでなければなりません。戻り値のデータ型は、オペランド C および D に共通する親データ型の中でも最も近いデータ型になります。「＞」および「＜」という比較を用いて条件付き選択演算を行う場合は、1 番目と 2 番目のオペランドの符号は次の表に従わなければなりません。

条件付き選択演算子の多重定義

R	比較条件	A、B	C	D
I32vec2 R	select_eq select_ne	I[s|u]32vec2	I[s|u]32vec2	I[s|u]32vec2
I16vec4 R		I[s|u]16vec4	I[s|u]16vec4	I[s|u]16vec4

I8vec8 R	select_gt	I[s|u]8vec8	I[s|u]8vec8	I[s|u]8vec8
I32vec2 R	select_gt select_ge select_lt select_le	Is32vec2	Is32vec2	Is32vec2
I16vec4 R		Is16vec4	Is16vec4	Is16vec4
I8vec8 R		Is8vec8	Is8vec8	Is8vec8

次の表は、任意の要素数に対する R0 から R7 までの戻り値の対応表です。戻り値が 4 つ未満のときも、これと同じ戻り値の対応表を適用します。

条件付き選択演算子の戻り値の対応表

戻り値	オペランド A、B							オペランド C、D
	A0	使用できる演算子						B0	
R0:=	A0	==	!=	>	>=	<	<=	B0	? C0 : D0;
R1:=	A0	==	!=	>	>=	<	<=	B0	? C1 : D1;
R2:=	A0	==	!=	>	>=	<	<=	B0	? C2 : D2;
R3:=	A0	==	!=	>	>=	<	<=	B0	? C3 : D3;
R4:=	A0	==	!=	>	>=	<	<=	B0	? C4 : D4;
R5:=	A0	==	!=	>	>=	<	<=	B0	? C5 : D5;
R6:=	A0	==	!=	>	>=	<	<=	B0	? C6 : D6;
R7:=	A0	==	!=	>	>=	<	<=	B0	? C7 : D7;

デバッグ

デバッグ演算は、MMX® テクノロジ命令のコンパイラ組込み関数との対応関係はありません。これらはプログラムのデバッグにのみ使用されます。これらの演算を使用するとパフォーマンスが下がる場合がありますので、デバッグ以外には使用しないでください。

出力

A の 4 つの 32 ビット値が出力バッファに格納され、次の形式で出力されます (デフォルトは 10 進数形式)。

```
cout << Is32vec4 A;

cout << Iu32vec4 A;

cout << hex << Iu32vec4 A; /* print in hex format */
```

```
"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```
対応する組込み関数: なし

A の 2 つの 32 ビット値が出力バッファに格納され、次の形式で出力されます (デフォルトは 10 進数形式)。

```
cout << Is32vec2 A;

cout << Iu32vec2 A;

cout << hex << Iu32vec2 A; /* print in hex format */

"[1]:A1 [0]:A0"
```
対応する組込み関数: なし

A の 8 つの 16 ビット値が出力バッファに格納され、次の形式で出力されます (デフォルトは 10 進数形式)。

```
cout << Is16vec8 A;

cout << Iu16vec8 A;

cout << hex << Iu16vec8 A; /* print in hex format */

"[7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1
[0]:A0"
```
対応する組込み関数: なし

A の 4 つの 16 ビット値が出力バッファに格納され、次の形式で出力されます (デフォルトは 10 進数形式)。

```
cout << Is16vec4 A;

cout << Iu16vec4 A;

cout << hex << Iu16vec4 A; /* print in hex format */

"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```
対応する組込み関数: なし

A の 16 の 8 ビット値が出力バッファに格納され、次の形式で出力されます (デフォルトは 10 進数形式)。

```
cout << Is8vec16 A; cout << Iu8vec16 A; cout << hex <<
Iu8vec8 A;

/* print in hex format instead of decimal*/
```

```
"[15]:A15 [14]:A14 [13]:A13 [12]:A12 [11]:A11 [10]:A10
[9]:A9 [8]:A8 [7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3
[2]:A2 [1]:A1 [0]:A0"
```

対応する組込み関数: なし

A の 8 つの 8 ビット値が出力バッファに格納され、次の形式で出力されます (デフォルトは 10 進数形式)。

```
cout << Is8vec8 A; cout << Iu8vec8 A;cout << hex <<
Iu8vec8 A;

/* print in hex format instead of decimal*/

"[7]:A7 [6]:A6 [5]:A5 [4]:A4 [3]:A3 [2]:A2 [1]:A1
[0]:A0"
```

対応する組込み関数: なし

要素アクセス演算子

```
int R = Is64vec2 A[i];

unsigned int R = Iu64vec2 A[i];

int R = Is32vec4 A[i];

unsigned int R = Iu32vec4 A[i];

int R = Is32vec2 A[i];

unsigned int R = Iu32vec2 A[i];

short R = Is16vec8 A[i];

unsigned short R = Iu16vec8 A[i];

short R = Is16vec4 A[i];

unsigned short R = Iu16vec4 A[i];

signed char R = Is8vec16 A[i];

unsigned char R = Iu8vec16 A[i];

signed char R = Is8vec8 A[i];

unsigned char R = Iu8vec8 A[i];
```

A の要素 i にアクセスして読み取ります。DEBUG が有効でユーザが A 以外の要素にアクセスしようとした場合、診断メッセージが出力され、プログラムは途中で終了します。

対応する組込み関数: なし

要素代入演算子

```
Is64vec2 A[i] = int R;

Is32vec4 A[i] = int R;

Iu32vec4 A[i] = unsigned int R;

Is32vec2 A[i] = int R;

Iu32vec2 A[i] = unsigned int R;

Is16vec8 A[i] = short R;

Iu16vec8 A[i] = unsigned short R;

Is16vec4 A[i] = short R;

Iu16vec4 A[i] = unsigned short R;

Is8vec16 A[i] = signed char R;

Iu8vec16 A[i] = unsigned char R;

Is8vec8 A[i] = signed char R;

Iu8vec8 A[i] = unsigned char R;
```

A の要素 i に R を代入します。DEBUG が有効でユーザが A 以外の要素に何らかの値を代入しようとした場合、診断メッセージが表示され、プログラムは途中で終了します。

対応する組込み関数: なし

アンパック演算子

A の上位半分から取り出した 64 ビット値を、B の上位半分から取り出した 64 ビット値とインターリーブします。

```
I364vec2 unpack_high(I64vec2 A, I64vec2 B);

Is64vec2 unpack_high(Is64vec2 A, Is64vec2 B);

Iu64vec2 unpack_high(Iu64vec2 A, Iu64vec2 B);

R0 = A1;
R1 = B1;
```

対応する組込み関数: _mm_unpackhi_epi64

A の上位半分から取り出した 2 つの 32 ビット値を、B の上位半分から取り出した 2 つの 32 ビット値とインターリーブします。

```
I32vec4 unpack_high(I32vec4 A, I32vec4 B);

Is32vec4 unpack_high(Is32vec4 A, Is32vec4 B);

Iu32vec4 unpack_high(Iu32vec4 A, Iu32vec4 B);

R0 = A1;
R1 = B1;
R2 = A2;
R3 = B2;
```

対応する組込み関数: _mm_unpackhi_epi32

A の上位半分から取り出した 32 ビット値を、B の上位半分から取り出した 32 ビット値とインターリーブします。

```
I32vec2 unpack_high(I32vec2 A, I32vec2 B);

Is32vec2 unpack_high(Is32vec2 A, Is32vec2 B);

Iu32vec2 unpack_high(Iu32vec2 A, Iu32vec2 B);

R0 = A1;
R1 = B1;
```

対応する組込み関数: _mm_unpackhi_pi32

A の上位半分から取り出した 4 つの 16 ビット値を、B の上位半分から取り出した 2 つの 16 ビット値とインターリーブします。

```
I16vec8 unpack_high(I16vec8 A, I16vec8 B);

Is16vec8 unpack_high(Is16vec8 A, Is16vec8 B);
```

```
Iu16vec8 unpack_high(Iu16vec8 A, Iu16vec8 B);

R0 = A2;
R1 = B2;
R2 = A3;
R3 = B3;
```

対応する組込み関数: `_mm_unpackhi_epi16`

A の上位半分から取り出した 2 つの 16 ビット値を、B の上位半分から取り出した 2 つの 16 ビット値とインターリーブします。

```
I16vec4 unpack_high(I16vec4 A, I16vec4 B);

Is16vec4 unpack_high(Is16vec4 A, Is16vec4 B);

Iu16vec4 unpack_high(Iu16vec4 A, Iu16vec4 B);

R0 = A2;R1 = B2;
R2 = A3;R3 = B3;
```

対応する組込み関数: `_mm_unpackhi_pi16`

A の上位半分から取り出した 4 つの 8 ビット値を、B の上位半分から取り出した 2 つの 8 ビット値とインターリーブします。

```
I8vec8 unpack_high(I8vec8 A, I8vec8 B);

Is8vec8 unpack_high(Is8vec8 A, I8vec8 B);

Iu8vec8 unpack_high(Iu8vec8 A, I8vec8 B);

R0 = A4;
R1 = B4;
R2 = A5;
R3 = B5;
R4 = A6;
R5 = B6;
R6 = A7;
R7 = B7;
```

対応する組込み関数: `_mm_unpackhi_pi8`

A の上位半分から取り出した 16 の 8 ビット値を、B の上位半分から取り出した 4 つの 8 ビット値とインターリーブします。

```
I8vec16 unpack_high(I8vec16 A, I8vec16 B);

Is8vec16 unpack_high(Is8vec16 A, I8vec16 B);

Iu8vec16 unpack_high(Iu8vec16 A, I8vec16 B);
```

```
R0 = A8;
R1 = B8;
R2 = A9;
R3 = B9;
R4 = A10;
R5 = B10;
R6 = A11;
R7 = B11;
R8 = A12;
R8 = B12;
R2 = A13;
R3 = B13;
R4 = A14;
R5 = B14;
R6 = A15;
R7 = B15;
```

対応する組込み関数: `_mm_unpackhi_epi16`

A の下位半分から取り出した 32 ビット値を、B の下位半分から取り出した 32 ビット値とインターリーブします。

```
R0 = A0;
R1 = B0;
```

対応する組込み関数: `_mm_unpacklo_epi32`

A の下位半分から取り出した 64 ビット値を、B の下位半分から取り出した 64 ビット値とインターリーブします。

```
I64vec2 unpack_low(I64vec2 A, I64vec2 B);

Is64vec2 unpack_low(Is64vec2 A, Is64vec2 B);

Iu64vec2 unpack_low(Iu64vec2 A, Iu64vec2 B);

R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```

対応する組込み関数: `_mm_unpacklo_epi32`

A の下位半分から取り出した 2 つの 32 ビット値を、B の下位半分から取り出した 2 つの 32 ビット値とインターリーブします。

```
I32vec4 unpack_low(I32vec4 A, I32vec4 B);

Is32vec4 unpack_low(Is32vec4 A, Is32vec4 B);

Iu32vec4 unpack_low(Iu32vec4 A, Iu32vec4 B);
```

```
R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
```
対応する組込み関数: _mm_unpacklo_epi32

A の下位半分から取り出した 32 ビット値を、B の下位半分から取り出した 32 ビット値とインターリーブします。

```
I32vec2 unpack_low(I32vec2 A, I32vec2 B);

Is32vec2 unpack_low(Is32vec2 A, Is32vec2 B);

Iu32vec2 unpack_low(Iu32vec2 A, Iu32vec2 B);

R0 = A0;
R1 = B0;
```
対応する組込み関数: _mm_unpacklo_pi32

A の下位半分から取り出した 2 つの 16 ビット値を、B の下位半分から取り出した 2 つの 16 ビット値とインターリーブします。

```
I16vec8 unpack_low(I16vec8 A, I16vec8 B);

Is16vec8 unpack_low(Is16vec8 A, Is16vec8 B);

Iu16vec8 unpack_low(Iu16vec8 A, Iu16vec8 B);

R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
```
対応する組込み関数: _mm_unpacklo_epi16

A の下位半分から取り出した 2 つの 16 ビット値を、B の下位半分から取り出した 2 つの 16 ビット値とインターリーブします。

```
I16vec4 unpack_low(I16vec4 A, I16vec4 B);

Is16vec4 unpack_low(Is16vec4 A, Is16vec4 B);

Iu16vec4 unpack_low(Iu16vec4 A, Iu16vec4 B);

R0 = A0;
R1 = B0;
```

```
R2 = A1;
R3 = B1;
```
対応する組込み関数: _mm_unpacklo_pi16

A の下位半分から取り出した 4 つの 8 ビット値を、B の下位半分から取り出した 4 つの 8 ビット値とインターリーブします。

```
I8vec16 unpack_low(I8vec16 A, I8vec16 B);

Is8vec16 unpack_low(Is8vec16 A, Is8vec16 B);

Iu8vec16 unpack_low(Iu8vec16 A, Iu8vec16 B);

R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
R8 = A4;
R9 = B4;
R10 = A5;
R11 = B5;
R12 = A6;
R13 = B6;
R14 = A7;
R15 = B7;
```
対応する組込み関数: _mm_unpacklo_epi8

A の下位半分から取り出した 4 つの 8 ビット値を、B の下位半分から取り出した 4 つの 8 ビット値とインターリーブします。

```
I8vec8 unpack_low(I8vec8 A, I8vec8 B);

Is8vec8 unpack_low(Is8vec8 A, Is8vec8 B);

Iu8vec8 unpack_low(Iu8vec8 A, Iu8vec8 B);

R0 = A0;
R1 = B0;
R2 = A1;
R3 = B1;
R4 = A2;
R5 = B2;
R6 = A3;
R7 = B3;
```
対応する組込み関数: _mm_unpacklo_pi8

パック演算子

A と B に含まれている 8 つの 32 ビット値を 8 つの 16 ビット値 (符号付き、飽和あり) にパックします。

```
Is16vec8 pack_sat(Is32vec2 A,Is32vec2 B);
```
対応する組込み関数: _mm_packs_epi32

A と B に含まれている 4 つの 32 ビット値を 8 つの 16 ビット値 (符号付き、飽和あり) にパックします。

```
Is16vec4 pack_sat(Is32vec2 A,Is32vec2 B);
```
対応する組込み関数: _mm_packs_pi32

A と B に含まれている 16 の 16 ビット値を 16 の 8 ビット値 (符号付き、飽和あり) にパックします。

```
Is8vec16 pack_sat(Is16vec4 A,Is16vec4 B);
```
対応する組込み関数: _mm_packs_epi16

A と B に含まれている 8 つの 16 ビット値を 8 つの 8 ビット値 (符号付き、飽和あり) にパックします。

```
Is8vec8 pack_sat(Is16vec4 A,Is16vec4 B);
```
対応する組込み関数: _mm_packs_pi16

A と B に含まれている 16 の 16 ビット値を 16 の 8 ビット値 (符号なし、飽和あり) にパックします。

```
Iu8vec16 packu_sat(Is16vec4 A,Is16vec4 B);
```
対応する組込み関数: _mm_packus_epi16

A と B に含まれている 8 つの 16 ビット値を 8 つの 8 ビット値 (符号なし、飽和あり) にパックします。

```
Iu8vec8 packu_sat(Is16vec4 A,Is16vec4 B);
```
対応する組込み関数: _mm_packs_pu16

MMX® テクノロジ・ステート消去演算子

MMX® テクノロジ・レジスタを空にして MMX テクノロジ・ステートを消去します。EMMS 命令の組込み関数の使用方法に関するガイドラインをお読みください。

```
void empty(void);
```
対応する組込み関数: _mm_empty

ストリーミング SIMD 拡張命令用の整数組込み関数

注

次の機能を使用する場合は、ヘッダファイル fvec.h をインクルードする必要があります。

A と B に含まれているそれぞれの符号付き整数ワードの中で、要素単位で見たときの最大値を計算します。

```
Is16vec4 simd_max(Is16vec4 A, Is16vec4 B);
```
対応する組込み関数: _mm_max_pi16

A と B に含まれているそれぞれの符号付き整数ワードの中で、要素単位で見たときの最小値を計算します。

```
Is16vec4 simd_min(Is16vec4 A, Is16vec4 B);
```
対応する組込み関数: _mm_min_pi16

A と B に含まれているそれぞれの符号なしバイトの中で、要素単位で見たときの最大値を計算します。

```
Iu8vec8 simd_max(Iu8vec8 A, Iu8vec8 B);
```
対応する組込み関数: _mm_max_pu8

A と B に含まれているそれぞれの符号なしバイトの中で、要素単位で見たときの最小値を計算します。

```
Iu8vec8 simd_min(Iu8vec8 A, Iu8vec8 B);
```
対応する組込み関数: _mm_min_pu8

A に含まれているすべてのバイトの最上位ビットから 8 ビットマスクを 1 つ作成します。

```
int move_mask(I8vec8 A);
```
対応する組込み関数: _mm_movemask_pi8

条件に従って、A のバイト要素をいくつかアドレス p にストアします。セレクタ B の各バイトの上位ビットによって、それに対応する A の各バイトがストアされるかどうかが決まります。

```
void mask_move(I8vec8 A, I8vec8 B, signed char *p);
```
対応する組込み関数: `_mm_maskmove_si64`

当該キャッシュ・データに影響を与えることなく、A に含まれているデータをアドレス p にストアします。A は Ivec 型であれば何でもかまいません。

```
void store_nta(__m64 *p, M64 A);
```
対応する組込み関数: `_mm_stream_pi`

A と B に含まれているそれぞれの符号なし 8 ビット整数すべてについて、要素単位で見たときの平均値を計算します。

```
Iu8vec8 simd_avg(Iu8vec8 A, Iu8vec8 B);
```
対応する組込み関数: `_mm_avg_pu8`

A と B に含まれているそれぞれの符号なし 16 ビット整数すべてについて、要素単位で見たときの平均値を計算します。

```
Iu16vec4 simd_avg(Iu16vec4 A, Iu16vec4 B);
```
対応する組込み関数: `_mm_avg_pu16`

Fvec と Ivec 間の変換

A の倍精度浮動小数点値の下位側を、切り捨てモードで 1 つの 32 ビット整数に変換します。

```
int F64vec2ToInt(F64vec42 A);
r := (int)A0;
```

A の 4 つの浮動小数点値を、2 つの最下位倍精度浮動小数点値に変換します。

```
F64vec2 F32vec4ToF64vec2(F32vec4 A);
r0 := (double)A0;
r1 := (double)A1;
```

A の 2 つの倍精度浮動小数点値を、2 つの単精度浮動小数点値に変換します。

```
F32vec4 F64vec2ToF32vec4(F64vec2 A);
r0 := (float)A0;
r1 := (float)A1;
```

B に含まれている符号付き整数 (int) を、1 つの倍精度浮動小数点値に変換し、その上位側の倍精度値を A から演算結果に渡します。

```
F64vec2 InttoF64vec2(F64vec2 A, int B);
r0 := (double)B;
r1 := A1;
```

A の浮動小数点値の下位側を、切り捨てモードで 1 つの 32 ビット整数に変換します。

```
int F32vec4ToInt(F32vec4 A);
r := (int)A0;
```

A の浮動小数点値の下位側の 2 つを、切り捨てモードで 2 つの 32 ビット整数に変換し、その整数をパックド形式で返します。

```
Is32vec2 F32vec4ToIs32vec2 (F32vec4 A);
r0 := (int)A0;
r1 := (int)A1;
```

32 ビット整数値 B を、1 つの浮動小数点値に変換します。その上位 3 つの浮動小数点値は A から渡されます。

```
F32vec4 IntToF32vec4(F32vec4 A, int B);
r0 := (float)B;
r1 := A1;
r2 := A2;
r3 := A3;
```

B の中にパックド形式で入っている 2 つの 32 ビット整数値を、2 つの浮動小数点値に変換します。その上位 2 つの浮動小数点値は A から渡されます。

```
F32vec4 Is32vec2ToF32vec4(F32vec4 A, Is32vec2 B);
r0 := (float)B0;
r1 := (float)B1;
r2 := A2;
r3 := A3;
```

概要: 浮動小数点ベクトルクラス

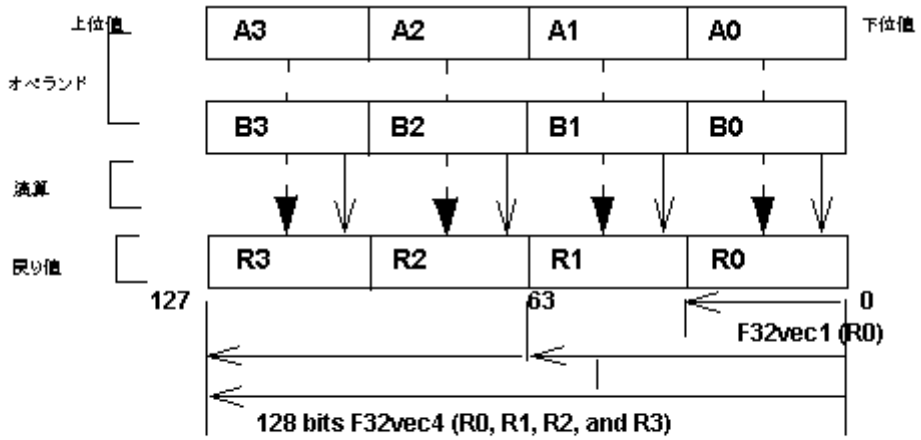
概要: 浮動小数点ベクトルクラス

浮動小数点ベクトルクラス、F64vec2、F32vec4、F32vec1 は、SIMD 演算へのインターフェイスとなります。このクラスの仕様は次のとおりです:

```
F64vec2 A(double x, double y);

F32vec4 A(float z, float y, float x, float w);

F32vec1 B(float w);
```

パックド浮動小数点の入力値は、下図に示すとおり最下位値を一番右側にして表現します。

単精度浮動小数点の要素



F32vec34は、パックド単精度浮動小数点値を4個(R0、R1、R2、およびR3)返します。
F32vec2は、単精度浮動小数点値を1個(R0)返します。

Fvec の表記法

本書では、構文と戻り値に次の表記法を使用しています。

Fvec クラスの構文の表記法

Fvec クラスの構文には次の表記を使用します:

```
[Fvec_Class] R = [Fvec_Class] A [operator][Ivec_Class] B;
```

例 1: `F64vec2 R = F64vec2 A & F64vec2 B;`

```
[Fvec_Class] R = [operator]([Fvec_Class] A,[Fvec_Class] B);
```

例 2: `F64vec2 R = andnot(F64vec2 A, F64vec2 B);`

```
[Fvec_Class] R [operator]= [Fvec_Class] A;
```

例 3: `F64vec2 R &= F64vec2 A;`

各アイテムの意味は次のとおりです:

`[operator]` は演算子 (例えば、&、|、^)

[Fvec_Class] は、Fvec クラス (F64vec2、F32vec4、または F32vec1)

R、A、B は、それぞれ宣言済みの Fvec 変数

戻り値の表記法

Fvec クラスはパックド要素をいくつか含むため、通常その戻り値は下記の表「戻り値表記の対応表」に示した規則に従います。F32vec4 は 4 つの単精度浮動小数点値 (R0,R1,R2,R3) を返します。F64vec2 は 2 つの倍精度浮動小数点値を返し、F32vec1 は最下位単精度浮動小数点値 (R0) を返します。

戻り値表記の対応表

例 1:	例 2:	例 3:	F32vec4	F64vec2	F32vec1
R0 := A0 & B0;	R0 := A0 andnot B0;	R0 &= A0;	x	x	x
R1 := A1 & B1;	R1 := A1 andnot B1;	R1 &= A1;	x	x	N/A
R2 := A2 & B2;	R2 := A2 andnot B2;	R2 &= A2;	x	N/A	N/A
R3 := A3 & B3	R3 := A3 andhot B3;	R3 &= A3;	x	N/A	N/A

データのアライメント

ストリーミング SIMD 拡張命令を使用してメモリを操作する場合は、できる限り、16 バイトでアライメントされたデータで行ってください。

F32vec4 および F64vec2 オブジェクトの各変数は、デフォルトで正しくアライメントされています。浮動小数点配列は自動的にはアライメントされないので注意してください。16 バイトでアライメントするには、__declspec を次のように使用します:

```
__declspec( align(16) ) float A[4];
```

変換

```
__m128d mm = A & B; /* where A,B are F64vec2 object
variables */

__m128 mm = A & B; /* where A,B are F32vec4 object
variables */

__m128 mm = A & B; /* where A,B are F32vec1 object
variables */
```

Fvec オブジェクトのすべての変数は __m128 データ型に暗黙的に変換できます。例えば、F32vec4 オブジェクトも F32vec1 オブジェクトも、その各変数に対して実行された計算結果は __m128 データ型に変換できます。

コンストラクタと初期化

次の表に、Fvec クラスで F32vec オブジェクトを作成して初期化する方法を示します。

Fvec クラスのコンストラクタと初期化

例	組込み関数	戻り値
コンストラクタの宣言		

`F64vec2 A;`
`F32vec4 B;`
`F32vec1 C;`
	N/A	N/A

__m128 オブジェクトの初期化		

`F64vec2 A(__m128d mm);`
`F32vec4 B(__m128 mm);`
`F32vec1 C(__m128 mm);`
	N/A	N/A

double 型の初期化		

`/* Initializes two doubles. */`
`F64vec2 A(double d0, double d1);`
`F64vec2 A = F64vec2(double d0, double d1);`
	_mm_set_pd	A0 := d0;
		A1 := d1;

`F64vec2 A(double d0);`
`/* Initializes both return values`
`with the same double precision value */.`
	_mm_set1_pd	A0 := d0;
		A1 := d0;

float 型の初期化		

`F32vec4 A(float f3, float f2,`
`float f1, float f0);`
`F32vec4 A = F32vec4(float f3, float f2,`
`float f1, float f0);`
	_mm_set_ps	A0 := f0;
		A1 := f1;
		A2 := f2;
		A3 := f3;

`F32vec4 A(float f0);`
`/* Initializes all return values`
`with the same floating point value. */`
	_mm_set1_ps	A0 := f0;
		A1 := f0;
		A2 := f0;
		A3 := f0;

`F32vec4 A(double d0);`
`/* Initialize all return values with`
`the same double-precision value. */`
	_mm_set1_ps(d)	A0 := d0;
		A1 := d0;
		A2 := d0;
		A3 := d0;

`F32vec1 A(double d0);`
`/* Initializes the lowest value of A`
`with d0 and the other values with 0.*/`
	_mm_set_ss(d)	A0 := d0;
		A1 := 0;
		A2 := 0;
		A3 := 0;

`F32vec1 B(float f0);`	_mm_set_ss	B0 := f0;

/* Initializes the lowest value of B with f0 and the other values with 0.*/		B1 := 0; B2 := 0; B3 := 0;
F32vec1 B(int I); /* Initializes the lowest value of B with f0, other values are undefined.*/	_mm_cvtsi32_ss	B0 := f0; B1 := {} B2 := {} B3 := {}

算術演算子

次の表は、Fvec クラスの算術演算子と一般的な構文を列挙したものです。この算術演算子はそれぞれ「標準演算」と「高度な演算」に分かれています。この 2 つについてはこのセクションの後半で詳しく解説します。

Fvec 算術演算子

分類	操作	演算子	一般的な構文
標準演算	加算	+ +=	R = A + B; R += A;
	減算	- -=	R = A - B; R -= A;
	乗算	* *=	R = A * B; R *= A;
	除算	/ /=	R = A / B; R /= A;
高度な演算	平方根	sqrt	R = sqrt(A);
	逆数 (ニュートン・ラフソン法)	rcp rcp_nr	R = rcp(A); R = rcp_nr(A);
	平方根の逆数 (ニュートン・ラフソン法)	rsqrt rsqrt_nr	R = rsqrt(A); R = rsqrt_nr(A);

「標準算術演算子」の使用方法

次の 2 つの表は、標準算術演算子のクラスごとに戻り値を列挙したものです。「戻り値の表記法」の前半に述べた構文が使われます。

標準算術演算の戻り値の対応表

R	A	演算子				B	F32vec4	F64vec2	F32vec1
R0:=	A0	+	-	*	/	B0			
R1:=	A1	+	-	*	/	B1			N/A

R2:=	A2	+	-	*	/	B2		N/A	N/A
R3:=	A3	+	-	*	/	B3		N/A	N/A

代入付き算術演算の戻り値の対応表

R	演算子				A	F32vec4	F64vec2	F32vec1
R0:=	+=	-=	*=	/=	A0			
R1:=	+=	-=	*=	/=	A1			N/A
R2:=	+=	-=	*=	/=	A2		N/A	N/A
R3:=	+=	-=	*=	/=	A3		N/A	N/A

次の表は、標準算術演算子の構文と組込み関数を列挙したものです。

Fvec クラスの標準算術演算

操作	戻り値	構文の使用例	組込み関数
加算	4 float	`F32vec4 R = F32vec4 A + F32vec4 B;` `F32vec4 R += F32vec4 A;`	`_mm_add_ps`
	2 double	`F64vec2 R = F64vec2 A + F32vec2 B;` `F64vec2 R += F64vec2 A;`	`_mm_add_pd`
	1 float	`F32vec1 R = F32vec1 A + F32vec1 B;` `F32vec1 R += F32vec1 A;`	`_mm_add_ss`
減算	4 float	`F32vec4 R = F32vec4 A - F32vec4 B;` `F32vec4 R -= F32vec4 A;`	`_mm_sub_ps`
	2 double	`F64vec2 R - F64vec2 A + F32vec2 B;` `F64vec2 R -= F64vec2 A;`	`_mm_sub_pd`
	1 float	`F32vec1 R = F32vec1 A - F32vec1 B;` `F32vec1 R -= F32vec1 A;`	`_mm_sub_ss`
乗算	4 float	`F32vec4 R = F32vec4 A * F32vec4 B;` `F32vec4 R *= F32vec4 A;`	`_mm_mul_ps`
	2 double	`F64vec2 R = F64vec2 A * F364vec2 B;` `F64vec2 R *= F64vec2 A;`	`_mm_mul_pd`
	1 float	`F32vec1 R = F32vec1 A * F32vec1 B;` `F32vec1 R *= F32vec1 A;`	`_mm_mul_ss`
除算	4 float	`F32vec4 R = F32vec4 A / F32vec4 B;` `F32vec4 R /= F32vec4 A;`	`_mm_div_ps`
	2 double	`F64vec2 R = F64vec2 A / F64vec2 B;` `F64vec2 R /= F64vec2 A;`	`_mm_div_pd`
	1 float	`F32vec1 R = F32vec1 A / F32vec1 B;` `F32vec1 R /= F32vec1 A;`	`_mm_div_ss`

「高度な算術演算子」の使用方法

次の表は、高度な算術演算子のクラスごとに戻り値を列挙したものです。「戻り値の表記法」の前半に述べた構文が使われます。

高度な算術演算子の戻り値の対応表

R	演算子					A	F32vec4	F64vec2	F32vec1
R0:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A0			
R1:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A1			N/A
R2:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A2		N/A	N/A
R3:=	sqrt	rcp	rsqrt	rcp_nr	rsqrt_nr	A3		N/A	N/A
f :=	add_horizontal			(A0 + A1 + A2 + A3)				N/A	N/A
d :=	add_horizontal			(A0 + A1)			N/A		N/A

次の表に、高度な算術演算子の例をいくつか示します。

Fvec クラス用の高度な算術演算

戻り値	構文の使用例	組込み関数
4 float	`F32vec4 R = sqrt(F32vec4 A);`	`_mm_sqrt_ps`
2 double	`F64vec2 R = sqrt(F64vec2 A);`	`_mm_sqrt_pd`
1 float	`F32vec1 R = sqrt(F32vec1 A);`	`_mm_sqrt_ss`
逆数		
4 float	`F32vec4 R = rcp(F32vec4 A);`	`_mm_rcp_ps`
2 double	`F64vec2 R = rcp(F64vec2 A);`	`_mm_rcp_pd`
1 float	`F32vec1 R = rcp(F32vec1 A);`	`_mm_rcp_ss`
平方根の逆数		
4 float	`F32vec4 R = rsqrt(F32vec4 A);`	`_mm_rsqrt_ps`
2 double	`F64vec2 R = rsqrt(F64vec2 A);`	`_mm_rsqrt_pd`
1 float	`F32vec1 R = rsqrt(F32vec1 A);`	`_mm_rsqrt_ss`
逆数 (ニュートン・ラフソン法)		

4 float	`F32vec4 R = rcp_nr(F32vec4 A);`	`_mm_sub_ps`
		`_mm_add_ps`
		`_mm_mul_ps`
		`_mm_rcp_ps`

2 double	`F64vec2 R = rcp_nr(F64vec2 A);`	`_mm_sub_pd`
		`_mm_add_pd`
		`_mm_mul_pd`
		`_mm_rcp_pd`

1 float	`F32vec1 R = rcp_nr(F32vec1 A);`	`_mm_sub_ss`
		`_mm_add_ss`
		`_mm_mul_ss`
		`_mm_rcp_ss`

平方根の逆数 (ニュートン・ラフソン法)		
4 float	F32vec4 R = rsqrt_nr(F32vec4 A);	_mm_sub_pd _mm_mul_pd _mm_rsqrt_ps
2 double	F64vec2 R = rsqrt_nr(F64vec2 A);	_mm_sub_pd _mm_mul_pd _mm_rsqrt_pd
1 float	F32vec1 R = rsqrt_nr(F32vec1 A);	_mm_sub_ss _mm_mul_ss _mm_rsqrt_ss
水平加算		
1 float	float f = add_horizontal(F32vec4 A);	_mm_add_ss _mm_shuffle_ss
1 double	double d = add_horizontal(F64vec2 A);	_mm_add_sd _mm_shuffle_sd

最小値演算子と最大値演算子

A と B の 2 つの倍精度浮動小数点値の最小値を計算します。

```
F64vec2 R = simd_min(F64vec2 A, F64vec2 B)
R0 := min(A0,B0);
R1 := min(A1,B1);
```

対応する組込み関数: _mm_min_pd

A と B の 4 つの単精度浮動小数点値の最小値を計算します。

```
F32vec4 R = simd_min(F32vec4 A, F32vec4 B)
R0 := min(A0,B0);
R1 := min(A1,B1);
R2 := min(A2,B2);
R3 := min(A3,B3);
```

対応する組込み関数: _mm_min_ps

A と B の 最下位単精度浮動小数点値の最小値を計算します。

```
F32vec1 R = simd_min(F32vec1 A, F32vec1 B)
R0 := min(A0,B0);
```

対応する組込み関数: _mm_min_ss

A と B の 2 つの倍精度浮動小数点値の最大値を計算します。

```
F64vec2 simd_max(F64vec2 A, F64vec2 B)
R0 := max(A0,B0);
R1 := max(A1,B1);
```

対応する組込み関数: _mm_max_pd

A と B の 4 つの単精度浮動小数点値の最大値を計算します。

```
F32vec4 R = simd_man(F32vec4 A, F32vec4 B)
R0 := max(A0,B0);
R1 := max(A1,B1);
R2 := max(A2,B2);
R3 := max(A3,B3);
```
対応する組込み関数: _mm_max_ps

A と B の 最下位単精度浮動小数点値の最大値を計算します。

```
F32vec1 simd_max(F32vec1 A, F32vec1 B)
R0 := max(A0,B0);
```
対応する組込み関数: _mm_max_ss

論理演算子

次の表は、Fvec クラスの論理演算子と一般的な構文を列挙したものです。F32vec1 クラスの論理演算子は下位 32 ビットだけを使用します。

Fvec 論理演算子の戻り値の対応表

ビット単位演算	演算子	一般的な構文
AND	& &=	R = A & B; R &= A;
OR	| |=	R = A | B; R |= A;
XOR	^ ^=	R = A ^ B; R ^= A;
andnot	andnot	R = andnot(A);

次の表は、論理演算子の標準的な構文と、対応する組込み関数を示したものです。F32vec1 クラスには対応するスカラ組込み関数がなく、パックドベクトル組込み関数の下位 32 ビットにアクセスする点に注意してください。

Fvec クラスの論理演算

操作	戻り値	構文の使用例	組込み関数
AND	4 float	F32vec4 & = F32vec4 A & F32vec4 B; F32vec4 & &= F32vec4 A;	_mm_and_ps
	2 double	F64vec2 R = F64vec2 A & F32vec2 B; F64vec2 R &= F64vec2 A;	_mm_and_pd
	1 float	F32vec1 R = F32vec1 A & F32vec1 B; F32vec1 R &= F32vec1 A;	_mm_and_ps

OR	4 float	F32vec4 R = F32vec4 A | F32vec4 B; F32vec4 R |= F32vec4 A;	_mm_or_ps
	2 double	F64vec2 R = F64vec2 A | F32vec2 B; F64vec2 R |= F64vec2 A;	_mm_or_pd
	1 float	F32vec1 R = F32vec1 A | F32vec1 B; F32vec1 R |= F32vec1 A;	_mm_or_ps
XOR	4 float	F32vec4 R = F32vec4 A ^ F32vec4 B; F32vec4 R ^= F32vec4 A;	_mm_xor_ps
	2 double	F64vec2 R = F64vec2 A ^ F364vec2 B; F64vec2 R ^= F64vec2 A;	_mm_xor_pd
	1 float	F32vec1 R = F32vec1 A ^ F32vec1 B; F32vec1 R ^= F32vec1 A;	_mm_xor_ps
ANDNOT	2 double	F64vec2 R = andnot(F64vec2 A, F64vec2 B);	_mm_andnot_pd

比較演算子

このセクションでは、A と B の単精度浮動小数点値を比較する演算子について説明します。Fvec クラスのオブジェクト同士を比較した場合は、比較されたオブジェクトと同じクラスで戻り値が返ってきます。

次の表は、Fvec クラスの比較演算子を示したものです。

比較演算子と対応する組込み関数

比較の条件	演算子	構文
等しい	cmpeq	R = cmpeq(A, B)
等しくない	cmpneq	R = cmpneq(A, B)
より大きい	cmpgt	R = cmpgt(A, B)
以上	cmpge	R = cmpge(A, B)
より大きくない	cmpngt	R = cmpngt(A, B)
以上でない	cmpnge	R = cmpnge(A, B)
より小さい	cmplt	R = cmplt(A, B)
以下	cmple	R = cmple(A, B)
より小さくない	cmpnlt	R = cmpnlt(A, B)
以下でない	cmpnle	R = cmpnle(A, B)

比較演算子

比較した結果が真の場合、そのマスクは浮動小数点値ごとに 0xffffffff にセットされます。同様に、偽の場合は 0x00000000 にセットされます。次の表は、比較演算子のクラスごとに戻り値を列挙したものです。「戻り値の表記法」の前半に述べた構文が使われます。

比較演算子の戻り値の対応表

R	A0	任意の演算子	B	真の場合	偽の場合	F32vec4	F64vec2	F32vec1
R0:=	(A1 !(A1	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B1) B1)	0xffffffff	0x0000000	X	X	X
R1:=	(A1 !(A1	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B2) B2)	0xffffffff	0x0000000	X	X	N/A
R2:=	(A1 !(A1	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B3) B3)	0xffffffff	0x0000000	X	N/A	N/A
R3:=	A3	cmp[eq | lt | le | gt | ge] cmp[ne | nlt | nle | ngt | nge]	B3) B3)	0xffffffff	0x0000000	X	N/A	N/A

次の表に、比較演算と組込み関数の例をいくつか示します。

Fvec クラスの比較演算

戻り値	構文の使用例	組込み関数
等しいかどうかの比較		
4 float	F32vec4 R = cmpeq(F32vec4 A);	_mm_cmpeq_ps
2 double	F64vec2 R = cmpeq(F64vec2 A);	_mm_cmpeq_pd
1 float	F32vec1 R = cmpeq(F32vec1 A);	_mm_cmpeq_ss

等しくないかどうかの比較		
4 float	F32vec4 R = cmpneq(F32vec4 A);	_mm_cmpneq_ps
2 double	F64vec2 R = cmpneq(F64vec2 A);	_mm_cmpneq_pd
1 float	F32vec1 R = cmpneq(F32vec1 A);	_mm_cmpneq_ss
より小さいかどうかの比較		
4 float	F32vec4 R = cmplt(F32vec4 A);	_mm_cmplt_ps
2 double	F64vec2 R = cmplt(F64vec2 A);	_mm_cmplt_pd
1 float	F32vec1 R = cmplt(F32vec1 A);	_mm_cmplt_ss
以下かどうかの比較		
4 float	F32vec4 R = cmple(F32vec4 A);	_mm_cmple_ps
2 double	F64vec2 R = cmple(F64vec2 A);	_mm_cmple_pd
1 float	F32vec1 R = cmple(F32vec1 A);	_mm_cmple_pd
より大きいかどうかの比較		
4 float	F32vec4 R = cmpgt(F32vec4 A);	_mm_cmpgt_ps
2 double	F64vec2 R = cmpgt(F32vec42 A);	_mm_cmpgt_pd
1 float	F32vec1 R = cmpgt(F32vec1 A);	_mm_cmpgt_ss
以上かどうかの比較		
4 float	F32vec4 R = cmpge(F32vec4 A);	_mm_cmpge_ps
2 double	F64vec2 R = cmpge(F64vec2 A);	_mm_cmpge_pd
1 float	F32vec1 R = cmpge(F32vec1 A);	_mm_cmpge_ss
より小さくないかどうかの比較		
4 float	F32vec4 R = cmpnlt(F32vec4 A);	_mm_cmpnlt_ps
2 double	F64vec2 R = cmpnlt(F64vec2 A);	_mm_cmpnlt_pd
1 float	F32vec1 R = cmpnlt(F32vec1 A);	_mm_cmpnlt_ss
以下でないかどうかの比較		
4 float	F32vec4 R = cmpnle(F32vec4 A);	_mm_cmpnle_ps
2 double	F64vec2 R = cmpnle(F64vec2 A);	_mm_cmpnle_pd
1 float	F32vec1 R = cmpnle(F32vec1 A);	_mm_cmpnle_ss
より大きくないかどうかの比較		
4 float	F32vec4 R = cmpngt(F32vec4 A);	_mm_cmpngt_ps
2 double	F64vec2 R = cmpngt(F64vec2 A);	_mm_cmpngt_pd
1 float	F32vec1 R = cmpngt(F32vec1 A);	_mm_cmpngt_ss
以上でないかどうかの比較		
4 float	F32vec4 R = cmpnge(F32vec4 A);	_mm_cmpnge_ps

2 double	`F64vec2 R = cmpnge(F64vec2 A);`	`_mm_cmpnge_pd`
1 float	`F32vec1 R = cmpnge(F32vec1 A);`	`_mm_cmpnge_ss`

Fvec クラスの条件付き選択演算子

各条件付き関数は、単精度浮動小数点値である A と B を比較します。パラメータ C、D は戻り値に使用されます。Fvec クラスのオブジェクト同士を比較したときの戻り値のデータ型は、比較したオブジェクトと同じデータ型になります。

Fvec クラスの条件付き選択演算子

条件付き選択の条件	演算子	構文
等しい	`select_eq`	`R = select_eq(A, B)`
等しくない	`select_neq`	`R = select_neq(A, B)`
より大きい	`select_gt`	`R = select_gt(A, B)`
以上	`select_ge`	`R = select_ge(A, B)`
より大きくない	`select_gt`	`R = select_gt(A, B)`
以上でない	`select_ge`	`R = select_ge(A, B)`
より小さい	`select_lt`	`R = select_lt(A, B)`
以下	`select_le`	`R = select_le(A, B)`
より小さくない	`select_nlt`	`R = select_nlt(A, B)`
以下でない	`select_nle`	`R = select_nle(A, B)`

条件付き選択演算子の使用方法

条件付き選択演算子では、比較結果が真の場合は戻り値が C に格納され、偽の場合は D に格納されます。次の表は、条件付き選択演算子のクラスごとに戻り値を列挙したものです。「戻り値の表記法」の前半に述べた構文が使われます。

条件付き選択演算子の戻り値の対応表

R	A0	演算子	B	C	D	F32vec4	F64vec2	F32vec1
R0:=	(A1 !(A1	`select [eq | lt | le | gt | ge]` `select_[ne | nlt | nle | ngt | nge]`	B0) B0)	C0 C0	D0 D0	X	X	X
R1:=	(A2 !(A2	`select [eq | lt | le | gt | ge]`	B1) B1)	C1 C1	D1 D1	X	X	N/A

		select_[ne | nlt | nle | ngt | nge]						
R2:=	(A2 !(A2	select [eq | lt | le | gt | ge] select_[ne | nlt | nle | ngt | nge]	B2) B2)	C2 C2	D2 D2	X	N/A	N/A
R3:=	(A3 !(A3	select [eq | lt | le | gt | ge] select_[ne | nlt | nle | ngt | nge]	B3) B3)	C3 C3	D3 D3	X	N/A	N/A

次の表に、条件付き選択演算および対応する組込み関数の例をいくつか示します。

Fvec クラスの条件付き選択演算

戻り値	構文の使用例	組込み関数
等しいかどうかの比較		
4 float	F32vec4 R = select_eq(F32vec4 A);	_mm_cmpeq_ps
2 double	F64vec2 R = select_eq(F64vec2 A);	_mm_cmpeq_pd
1 float	F32vec1 R = select_eq(F32vec1 A);	_mm_cmpeq_ss
等しくないかどうかの比較		
4 float	F32vec4 R = select_neq(F32vec4 A);	_mm_cmpneq_ps
2 double	F64vec2 R = select_neq(F64vec2 A);	_mm_cmpneq_pd
1 float	F32vec1 R = select_neq(F32vec1 A);	_mm_cmpneq_ss
より小さいかどうかの比較		
4 float	F32vec4 R = select_lt(F32vec4 A);	_mm_cmplt_ps
2 double	F64vec2 R = select_lt(F64vec2 A);	_mm_cmplt_pd
1 float	F32vec1 R = select_lt(F32vec1 A);	_mm_cmplt_ss
以下かどうかの比較		
4 float	F32vec4 R = select_le(F32vec4 A);	_mm_cmple_ps
2 double	F64vec2 R = select_le(F64vec2 A);	_mm_cmple_pd
1 float	F32vec1 R = select_le(F32vec1 A);	_mm_cmple_ps
より大きいかどうかの比較		
4 float	F32vec4 R = select_gt(F32vec4 A);	_mm_cmpgt_ps
2 double	F64vec2 R = select_gt(F64vec2 A);	_mm_cmpgt_pd
1 float	F32vec1 R = select_gt(F32vec1 A);	_mm_cmpgt_ss
以上かどうかの比較		
4 float	F32vec1 R = select_ge(F32vec4 A);	_mm_cmpge_ps
2 double	F64vec2 R = select_ge(F64vec2 A);	_mm_cmpge_pd

1 float	F32vec1 R = select_ge(F32vec1 A);	_mm_cmpge_ss
より小さくないかどうかの比較		
4 float	F32vec1 R = select_nlt(F32vec4 A);	_mm_cmpnlt_ps
2 double	F64vec2 R = select_nlt(F64vec2 A);	_mm_cmpnlt_pd
1 float	F32vec1 R = select_nlt(F32vec1 A);	_mm_cmpnlt_ss
以下でないかどうかの比較		
4 float	F32vec1 R = select_nle(F32vec4 A);	_mm_cmpnle_ps
2 double	F64vec2 R = select_nle(F64vec2 A);	_mm_cmpnle_pd
1 float	F32vec1 R = select_nle(F32vec1 A);	_mm_cmpnle_ss
より大きくないかどうかの比較		
4 float	F32vec1 R = select_ngt(F32vec4 A);	_mm_cmpngt_ps
2 double	F64vec2 R = select_ngt(F64vec2 A);	_mm_cmpngt_pd
1 float	F32vec1 R = select_ngt(F32vec1 A);	_mm_cmpngt_ss
以上でないかどうかの比較		
4 float	F32vec1 R = select_nge(F32vec4 A);	_mm_cmpnge_ps
2 double	F64vec2 R = select_nge(F64vec2 A);	_mm_cmpnge_pd
1 float	F32vec1 R = select_nge(F32vec1 A);	_mm_cmpnge_ss

キャッシュ操作

A の 2 つの倍精度浮動小数点値を (一時的ではなく) ストアします。16 バイトでアライメントされたアドレスが 1 つ必要です。

```
void store_nta(double *p, F64vec2 A);
```
対応する組込み関数: _mm_stream_pd

A の 4 つの単精度浮動小数点値を (一時的ではなく) ストアします。16 バイトでアライメントされたアドレスが 1 つ必要です。

```
void store_nta(float *p, F32vec4 A);
```
対応する組込み関数: _mm_stream_ps

デバッグ

デバッグ演算は、MMX® テクノロジ命令やストリーミング SIMD 拡張命令のコンパイラ組込み関数との対応関係はありません。これらはプログラムのデバッグにのみ使用

されます。これらの演算を使用するとパフォーマンスが下がる場合がありますので、デバッグ以外には使用しないでください。

出力演算

A の 2 つの倍精度浮動小数点値が出力バッファに格納され、次のように 10 進数形式で出力されます:

```
cout << F64vec2 A;
"[1]:A1 [0]:A0"
```
対応する組込み関数: なし

A の 4 つの単精度浮動小数点値が出力バッファに格納され、次のように 10 進数形式で出力されます:

```
cout << F32vec4 A;
"[3]:A3 [2]:A2 [1]:A1 [0]:A0"
```
対応する組込み関数: なし

A の最下位単精度浮動小数点値が出力バッファに格納され出力されます。

```
cout << F32vec1 A;
```
対応する組込み関数: なし

要素アクセス演算

```
double d = F64vec2 A[int i]
```

対応する浮動小数点値を変更することなく、A の 2 つの倍精度浮動小数点値のうちの 1 つを読み取ります。i の許容値は 0 と 1 です。次に例を示します:

DEBUG が有効で i が許容値 (0 または 1) のいずれでもない場合、診断メッセージが出力され、プログラムは途中で終了します。

```
double d = F64vec2 A[1];
```
対応する組込み関数: なし

対応する浮動小数値を変更せずに、A の 4 つの単精度浮動小数点値のうちの 1 つを読み取ります。i の許容値は 0、1、2、3 です。次に例を示します:

```
float f = F32vec4 A[int i]
```

DEBUG が有効で i が許容値 (0～3) のいずれでもない場合、診断メッセージが出力され、プログラムは途中で終了します。

```
float f = F32vec4 A[2];
```
対応する組込み関数: なし

要素代入演算

```
F64vec4 A[int i] = double d;
```

A の 2 つの倍精度浮動小数点値のうちの 1 つを変更します。整数 i の許容値は 0 と 1 です。次に例を示します:

```
F32vec4 A[1] = double d;
F32vec4 A[int i] = float f;
```

A の 4 つの単精度浮動小数点値のうちの 1 つを変更します。整数 i の許容値は 0、1、2、3 です。次に例を示します:

DEBUG が有効で整数 i が許容値 (0〜3) のいずれでもない場合、診断メッセージが出力され、プログラムは途中で終了します。

```
F32vec4 A[3] = float f;
```
対応する組込み関数: なし

ロード演算子とストア演算子

2 つの倍精度浮動小数点値をロードし、それらを 2 つの浮動小数点値 A にコピーします。アライメントは合っていなくてもかまいません。

```
void loadu(F64vec2 A, double *p)
```
対応する組込み関数: _mm_loadu_pd

A の 2 つの倍精度浮動小数点値をストアします。アライメントは合っていなくてもかまいません。

```
void storeu(float *p, F64vec2 A);
```
対応する組込み関数: _mm_storeu_pd

4 つの単精度浮動小数点値をロードし、それらを 4 つの浮動小数点値 A にコピーします。アライメントは合っていなくてもかまいません。

```
void loadu(F32vec4 A, double *p)
```
対応する組込み関数: _mm_loadu_ps

A の 4 つの単精度浮動小数点値をストアします。アライメントは合っていなくてもかまいません。

```
void storeu(float *p, F32vec4 A);
```
対応する組込み関数: `_mm_storeu_ps`

Fvec 演算子のアンパック演算子

A と B の中から、下位側にある倍精度浮動小数点値を取り出してインターリーブします。

```
F64vec2 R = unpack_low(F64vec2 A, F64vec2 B);
```
対応する組込み関数: `_mm_unpacklo_pd(a, b)`

A と B の中から、上位側にある倍精度浮動小数点値を取り出してインターリーブします。

```
F64vec2 R = unpack_high(F64vec2 A, F64vec2 B);
```
対応する組込み関数: `_mm_unpackhi_pd(a, b)`

A と B の中から、下位側にある 2 つの単精度浮動小数点値を取り出してインターリーブします。

```
F32vec4 R = unpack_low(F32vec4 A, F32vec4 B);
```
対応する組込み関数: `_mm_unpacklo_ps(a, b)`

A と B の中から、上位側にある 2 つの単精度浮動小数点値を取り出してインターリーブします。

```
F32vec4 R = unpack_high(F32vec4 A, F32vec4 B);
```
対応する組込み関数: `_mm_unpackhi_ps(a, b)`

マスク移動演算子

A の 2 つの倍精度浮動小数点値の最上位ビットから 2 ビットマスクを 1 つ作成します。次に例を示します:

```
int i = move_mask(F64vec2 A)
i := sign(a1)<<1 | sign(a0)<<0
```
対応する組込み関数: `_mm_movemask_pd`

A の 4 つの単精度浮動小数点値の最上位ビットから 4 ビットマスクを 1 つ作成します。次に例を示します:

```
int i = move_mask(F32vec4 A)
i := sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 |
sign(a0)<<0
```
対応する組込み関数: _mm_movemask_ps

各種クラスのクイック・リファレンス

この付録には、SIMD 演算用のインテル® C++ クラス・ライブラリのクラスごとに、クラス、機能、および対応する組込み関数を列挙した表がいくつか記載されています。次の表は、C++ の各 SIMD クラスには実装されていないインテル C++ コンパイラ組込み関数をすべてリストしたものです。

論理演算子: 対応する組込み関数とクラス

演算子	対応する組込み関数	I128vec1, I64vec2, I32vec4, I16vec8, I8vec16	I64vec, I32vec, I16vec, I8vec8	F64vec2	F32vec4	F32vec1
&, &=	_mm_and_[x]	si128	si64	pd	ps	ps
|, |=	_mm_or_[x]	si128	si64	pd	ps	ps
^, ^=	_mm_xor_[x]	si128	si64	pd	ps	ps
Andnot	_mm_andnot_[x]	si128	si64	pd	N/A	N/A

算術演算子: 対応する組込み関数とクラス

演算子	対応する組込み関数	I64vec2	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
+, +=	_mm_add_[x]	epi64	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	ss
-, -=	_mm_sub_[x]	epi64	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	ss
*, *=	_mm_mullo_[x]	N/A	N/A	epi16	N/A	N/A	pi16	N/A	pd	ps	ss
/, /=	_mm_div_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
mul_high	_mm_mulhi_[x]	N/A	N/A	epi16	N/A	N/A	pi16	N/A	N/A	N/A	N/A
mul_add	_mm_madd_[x]	N/A	N/A	epi16	N/A	N/A	pi16	N/A	N/A	N/A	N/A

sqrt	_mm_sqrt_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
rcp	_mm_rcp_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
rcp_nr	_mm_rcp_[x] _mm_add_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
rsqrt	_mm_rsqrt_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
rsqrt_nr	_mm_rsqrt_[x] _mm_sub_[x] _mm_mul_[x]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss

シフト演算子: 対応する組込み関数とクラス

演算子	対応する組込み関数	I128vec1	I64vec2	I32vec4	I16vec8	I8vec16	I64vec1	I32vec2	I16vec4	I8vec8
>>,>>=	_mm_srl_[x] _mm_srli_[x] _mm_sra_[x] _mm_srai_[x]	N/A N/A N/A N/A	epi64 epi64 N/A N/A	epi32 epi32 epi32 epi32	epi16 epi16 epi16 epi16	N/A N/A N/A N/A	si64 si64 N/A N/A	pi32 pi32 pi32 pi32	pi16 pi16 pi16 pi16	N/A N/A N/A N/A
<<, <<=	_mm_sll_[x] _mm_slli_[x]	N/A N/A	epi64 epi64	epi32 epi32	epi16 epi16	N/A N/A	si64 si64	pi32 pi32	pi16 pi16	N/A N/A

比較演算子: 対応する組込み関数とクラス

演算子	対応する組込み関数	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
cmpeq	_mm_cmpeq_[x]	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	ss
cmpneq	_mm_cmpeq_[x] _mm_andn	epi32 si128	epi16 si12	epi8 si128	pi32 si64	pi16 si64	pi8 si64	pd	ps	ss

	ot_[y]*		8							
cmpgt	_mm_cmpgt_[x]	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	ss
cmpge	_mm_cmpge_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64	pd	ps	ss
cmplt	_mm_cmplt_[x]	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	ss
cmple	_mm_cmple_[x] _mm_andnot_[y]*	epi32 si128	epi16 si128	epi8 si128	pi32 si64	pi16 si64	pi8 si64	pd	ps	ss
cmpngt	_mm_cmpngt_[x]	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	ss
cmpnge	_mm_cmpnge_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
cmnpnlt	_mm_cmpnlt_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
cmpnle	_mm_cmpnle_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss

* 組込み関数 _mm_andnot_[y] は fvec クラスには適用されない点に注意してください。

条件付き選択演算子: 対応する組込み関数とクラス

演算子	対応する組込み関数	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
select_eq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64	pd	ps	ss
select_neq	_mm_cmpeq_[x] _mm_and_[y] _mm_andnot_[y]* _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64	pd	ps	ss
select_gt	_mm_cmpgt_[x] _mm_and	epi32 si12	epi16 si12	epi8 si128	pi32 si64 si64	pi16 si64 si64	pi8 si64 si64	pd	ps	ss

	_[y] _mm_andnot_[y] * _mm_or_[y]	8 si128 si128	8 si128 8si128	si128 si128	si64	si64	si64			
select_ge	_mm_cmpge_[x] _mm_and_[y] _mm_andnot_[y] * _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64	pd	ps	ss
select_lt	_mm_cmplt_[x] _mm_and_[y] _mm_andnot_[y] * _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64	pd	ps	ss
select_le	_mm_cmple_[x] _mm_and_[y] _mm_andnot_[y] * _mm_or_[y]	epi32 si128 si128 si128	epi16 si128 si128 si128	epi8 si128 si128 si128	pi32 si64 si64 si64	pi16 si64 si64 si64	pi8 si64 si64 si64	pd	ps	ss
select_ngt	_mm_cmpgt_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
select_nge	_mm_cmpge_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
select_nlt	_mm_cmplt_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss
select_nle	_mm_cmple_[x]	N/A	N/A	N/A	N/A	N/A	N/A	pd	ps	ss

* 組込み関数 _mm_andnot_[y] は fvec クラスには適用されない点に注意してください。

パック演算子とアンパック演算子: 対応する組込み関数とクラス

演算子	対応する組込み関数	I64vec2	I32vec4	I16vec8	I8vec16	I32vec2	I16vec4	I8vec8	F64vec2	F32vec4	F32vec1
unpack_high	_mm_unpackhi_[x]	epi64	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	N/A

unpack_low	_mm_unpaklo_[x]	epi64	epi32	epi16	epi8	pi32	pi16	pi8	pd	ps	N/A
pack_sat	_mm_packs_[x]	N/A	epi32	epi16	N/A	pi32	pi16	N/A	N/A	N/A	N/A
packu_sat	_mm_packus_[x]	N/A	N/A	epi16	N/A	N/A	pu16	N/A	N/A	N/A	N/A
sat_add	_mm_adds_[x]	N/A	N/A	epi16	epi8	N/A	pi16	pi8	pd	ps	ss
sat_sub	_mm_subs_[x]	N/A	N/A	epi16	epi8	N/A	pi16	pi8	pi16	pi8	pd

変換演算子: 対応する組込み関数とクラス

演算子	対応する組込み関数
F64vec2ToInt	_mm_cvttsd_si32
F32vec4ToF64vec2	_mm_cvtps_pd
F64vec2ToF32vec4	_mm_cvtpd_ps
IntToF64vec2	_mm_cvtsi32_sd
F32vec4ToInt	_mm_cvtt_ss2si
F32vec4ToIs32vec2	_mm_cvttps_pi32
IntToF32vec4	_mm_cvtsi32_ss
Is32vec2ToF32vec4	_mm_cvtpi32_ps

プログラミング例

次のサンプル・プログラムは、F32vec4 クラスを使用して 20 の要素から成る浮動小数点配列のすべての要素の平均値を計算します。

```
// Include Streaming SIMD Extension Class Definitions
#include <fvec.h>

// Shuffle any 2 single precision floating point from a
// into low 2 SP FP and shuffle any 2 SP FP from b
// into high 2 SP FP of destination
#define SHUFFLE(a,b,i) (F32vec4) mm shuffle ps(a,b,i)
#include <stdio.h>
#define SIZE 20

// Global variables
float result;
 MM ALIGN 16 float array[SIZE];

//*************************************************
// Function: Add20ArrayElements
// Add all the elements of a 20 element array
```

```
//*****************************************************

void Add20ArrayElements (F32vec4 *array, float *result)
{
   F32vec4 vec0, vec1;
   vec0 =  mm load ps ((float *) array); // Load array's first 4
floats

   //*****************************************************
   // Add all elements of the array, 4 elements at a time
   //*****************************************************

   vec0 += array[1]; // Add elements 5-8
   vec0 += array[2]; // Add elements 9-12
   vec0 += array[3]; // Add elements 13-16
   vec0 += array[4]; // Add elements 17-20

   //*****************************************************
   // There are now 4 partial sums.
   // Add the 2 lowers to the 2 raises,
   // then add those 2 results together
   //*****************************************************

   vec1 = SHUFFLE(vec1, vec0, 0x40);
   vec0 += vec1;
   vec1 = SHUFFLE(vec1, vec0, 0x30);
   vec0 += vec1;
   vec0 = SHUFFLE(vec0, vec0, 2);
    mm store ss (result, vec0); // Store the final sum
}

void main(int argc, char *argv[])
{
   int i;
   // Initialize the array
   for (i=0; i < SIZE; i++)
   {
      array[i] = (float) i;
   }

   // Call function to add all array elements
   Add20ArrayElements (array, &result);

   // Print average array element value
   printf ("Average of all array values = %f\n", result/20.);
   printf ("The correct answer is %f\n\n\n", 9.5);
}
```

キーワード

#

#assert 45

#define 45

#pragma distribute point.....177

#pragma hdrstop 57

#pragma ivdep.....132, 143, 180

#pragma loop count.....177

#pragma noprefetch 179

#pragma noswp.....176

#pragma nounroll 178

#pragma novector 143, 180

#pragma omp 157, 160, 167

#pragma optimize.....85

#pragma prefetch 179

#pragma swp.....176

#pragma taskq.....170, 174

#pragma unroll 178

#pragma vector 143, 180

#undef 45

[

-[no]align オプション 13

-[no]restrict オプション 13

__

__cplusplus マクロ..... 81

__DATE__ マクロ 81

__declspec 94, 143

__ECC 事前定義済みマクロ 46

__EDG__ 事前定義済みマクロ 46

__ELF__ 事前定義済みマクロ 46

__FILE__ マクロ 81

__gnu_linux__ 事前定義済みマクロ 46

__GNUC__ 事前定義済みマクロ . 46, 76

__GNUC_MINOR__ 事前定義済みマクロ..... 46, 76

__GNUC_PATCHLEVEL__ 事前定義済みマクロ 46, 76

__GXX_ABI_VERSION 事前定義済みマクロ 46

__i386 事前定義済みマクロ 46

__i386__ 事前定義済みマクロ 46

__ia64 事前定義済みマクロ 46

__ia64__ 事前定義済みマクロ 46

__ICC 事前定義済みマクロ..... 46

__INTEL_COMPILER 事前定義済みマクロ 46

__LINE__ マクロ 81

__linux 事前定義済みマクロ 46

__linux__ 事前定義済みマクロ 46

__lp64 事前定義済みマクロ 46

__LP64__ 事前定義済みマクロ 46

__NO_INLINE__ 事前定義済みマクロ.....46

__NO_MATH_INLINES 事前定義済みマクロ 46

__NO_STRING_INLINES 事前定義済みマクロ 46

__OPTIMIZE__ 事前定義済みマクロ 46

__PTRDIFF_TYPE__ 事前定義済みマクロ 46

__SIZE_TYPE__ 事前定義済みマクロ.....46

__STDC__ マクロ 81

__TIME__ マクロ.....81

__unix 事前定義済みマクロ 46

__unix__ 事前定義済みマクロ 46

__USER_LABEL_PREFIX__ 事前定義済みマクロ.....46

_INTEGRAL_MAX_BITS 事前定義済みマクロ 46

_M_IA64 事前定義済みマクロ..... 46

_PGO_INSTRUMENT 事前定義済みマクロ..... 46

A

-A- オプション..... 13

acos ライブラリ関数 200

acosd ライブラリ関数 200

acosh ライブラリ関数 204

-alias_args[-] オプション 13

-align オプション..... 64

-Aname[(value)] オプション 13

annuity ライブラリ関数 211

-ansi オプション..... 13, 81

ANSI/ISO 81

-ansi_alias[-] オプション 13, 81

asin ライブラリ関数 200

asind ライブラリ関数 200

asinh ライブラリ関数 204

atan ライブラリ関数..... 200

atan2 ライブラリ関数..... 200

atand ライブラリ関数..... 200

atand2 ライブラリ関数..... 200

atanh ライブラリ関数..... 204

-auto_ilp32 オプション 13
-ax オプション 13, 93, 131, 135
B
bash_profile.....40
-Bdynamic リンカ・オプション.....60
-Bstatic リンカ・オプション.....60
C
-C オプション 13
C_INCLUDE_PATH 環境変数 49
-c99[-] オプション 13
cabs ライブラリ関数 224
cacos ライブラリ関数 224
cacosh ライブラリ関数 224
captureprivate.....170
carg ライブラリ関数.....224
casin ライブラリ関数 224
casinh ライブラリ関数 224
catan ライブラリ関数.....224
catanh ライブラリ関数.....224
cbrt ライブラリ関数.....206
ccos ライブラリ関数 224
ccosh ライブラリ関数 224
ceil ライブラリ関数..... 215
cexp ライブラリ関数 224
cexp10 ライブラリ関数 224
cimag ライブラリ関数 224
cis ライブラリ関数..... 224
clog ライブラリ関数 224
clog2 ライブラリ関数 224
-complex_limited_range オプション13
compound ライブラリ関数 211
conj ライブラリ関数 224
copysign ライブラリ関数..... 219
cos ライブラリ関数 200
cosd ライブラリ関数 200
cosh ライブラリ関数 204
cot ライブラリ関数 200
cotd ライブラリ関数 200
CPATH 環境変数 49
CPLUS_INCLUDE_PATH 環境変数 49
cpow ライブラリ関数..... 224
cproj ライブラリ関数 224
cpu ディスパッチ 94
creal ライブラリ関数 224

-create_pch オプション 13, 57
csin ライブラリ関数 224
csinh ライブラリ関数 224
csqrt ライブラリ関数.....224
ctan ライブラリ関数.....224
ctanh ライブラリ関数.....224
-cxxlib-gcc オプション 13, 76
-cxxlib-icc オプション.....13, 76

D

Dinkumware*
C++ ライブラリ 64
-dM オプション 13
-Dname[=value] オプション 13
-dryrun オプション 13
-dynamic-linker オプション 13

E

-E オプション.....13
ECCCFG 環境変数.....49
ECPCCFG 環境変数 49
EMMS 命令 243
-EP オプション 13
erf ライブラリ関数 211
erfc ライブラリ関数 211
exp ライブラリ関数 206
exp10 ライブラリ関数 206
exp2 ライブラリ関数 206
expm1 ライブラリ関数 206

F

-F オプション 13
-f[no]verbose-asm オプション..... 13
fabs ライブラリ関数 219
-falias オプション 13
-fast オプション..... 13
-fcode-asm オプション..... 13
fdim ライブラリ関数 219
-ffnalias オプション 13
finite ライブラリ関数 219
firstprivate 170
floor ライブラリ関数..... 215
fma ライブラリ関数..... 219
fmax ライブラリ関数 219
fmin ライブラリ関数 219
-fminshared オプション..... 13
fmod ライブラリ関数..... 218

-fno-alias オプション 13
-fno-common オプション 13
-fno-fnalias オプション.....13
-fno-rtti オプション 13
-fnsplit[-] オプション 13, 109
-fp オプション 13, 85
-fp_port オプション.....13, 85, 86
-fpic オプション 13
-fpstkchk オプション 13, 86
-fr32 オプション 13
frexp ライブラリ関数.....206
-fshort-enums オプション 13
-fsource-asm オプション 13
-fsyntax-only オプション 13
-ftz[-] オプション 13, 89
-funsigned-bitfields オプション 13
-funsigned-char オプション 13
fvec.....419
-fvisibility オプション.....13
-fvisibility-default= オプション.....13
-fvisibility-extern= オプション 13
-fvisibility-hidden= オプション.....13
-fvisibility-internal= オプション 13
-fvisibility-protected= オプション 13

G

-g オプション 13
gamma ライブラリ関数 211
gamma_r ライブラリ関数..... 211
gcc
  言語拡張機能..... 73
  互換..... 73
  相互運用性 76
gcc 関数属性..... 80
-gcc-name オプション..... 13, 76
-gcc-version= オプション 13, 76
GNU*
  C ライブラリ..... 64

H

-H オプション..... 13
-help オプション..... 13
hypot ライブラリ関数 206

I

-I オプション 13
-i_dynamic オプション 13

i386 事前定義済みマクロ 46
IA32ROOT 環境変数 49
ia64 事前定義済みマクロ 46
IA64ROOT 環境変数 49
ICCCFG 環境変数 49
iccvars.csh.....40
iccvars.sh 40
ICPCCFG 環境変数.....49
-idirafter オプション 13
ilogb ライブラリ関数 206
-inline_debug_info オプション ..13, 99
-ip オプション 13, 98, 99, 101
-ip_no_inlining オプション 13, 99, 101
-ip_no_pinlining オプション 13, 101
-IPF_flt_eval_method0 オプション 13, 85, 89
-IPF_fltacc[-] オプション 13, 85, 89
-IPF_fma[-] オプション 13, 85, 89
-IPF_fp_speculation オプション13, 85, 89
-ipo オプション ..13, 98, 99, 101, 102, 103, 104, 105, 106
-ipo_c オプション 13, 99
-ipo_obj オプション ..13, 99, 103, 135
-ipo_S オプション 13, 99
isnan ライブラリ関数 219
-isystem オプション 13
-ivdep_parallel オプション 13, 132
ivec
  整数..... 391
  浮動小数点 419
ivec 391

J

j0 ライブラリ関数 211
j1 ライブラリ関数 211
jn ライブラリ関数 211

K

-Kc++ オプション 13
KMP_LIBRARY 環境変数 164
KMP_STACKSIZE 環境変数..... 164
-Knopic オプション 13
-KPIC オプション 13

L

-L オプション 13
lastprivate 170
LD_LIBRARY_PATH 環境変数.... 66

ldexp ライブラリ関数 206
ld-linux.so.2 60
lgamma ライブラリ関数 211
lgamma_r ライブラリ関数 211
libimf.a 64
linux 事前定義済みマクロ 46
llrint ライブラリ関数 215
llround ライブラリ関数 215
log ライブラリ関数 206
log10 ライブラリ関数 206
log1p ライブラリ関数 206
log2 ライブラリ関数 206
logb ライブラリ関数 206
-long_double オプション 13, 86
lrint ライブラリ関数 215
lround ライブラリ関数 215

M

-M オプション 13
make 42
makefile 42
-march=cpu オプション 13, 92
-mcpu=cpu オプション 13, 90
-MD オプション 13
-MF オプション 13
-MG オプション 13
-MM オプション 13
-MMD オプション 13
-mno-relax オプション 13
-mno-serialize-volatile オプション .. 13
modf ライブラリ関数 215
-mp オプション 13, 86
-mp1 オプション 13, 85, 86
-mrelax オプション 13
-mserialize-volatile オプション 13
-MX オプション 13

N

nearbyint ライブラリ関数 215
nextafter ライブラリ関数 219
nexttoward ライブラリ関数 219
-no_cpprt オプション 13
-nobss_init オプション 13
-nodefaultlibs オプション 13
-no-gcc オプション 13, 76
-nolib_inline オプション 13, 86, 99

-nostartfiles オプション.....13

-nostdinc オプション.....13

-nostdlib オプション.....13

O

-o オプション 13, 105

-O0 オプション 13, 85

-O1 オプション 13, 84

-O2 オプション 13, 84

-O3 オプション 13, 84, 131

-Ob オプション 13

OMP_DYNAMIC 環境変数 164

OMP_NESTED 環境変数 164

OMP_NUM_THREADS 環境変数 154, 164

OMP_SCHEDULE 環境変数 154, 164

-openmp オプション 13, 134, 160

OpenMP*

ディレクティブ 162

節 162

OpenMP*...156, 157, 160, 162, 163, 164, 165, 167, 168, 174

-openmp_report オプション...13, 134, 160

-openmp_stubs オプション 13, 134

-opt_report オプション 13

-opt_report_file オプション..... 13, 182

-opt_report_help オプション .. 13, 182

-opt_report_level オプション.. 13, 182

-opt_report_phase オプション 13, 182

-opt_report_routine オプション 13, 182

P

-P オプション 13

-par_report オプション... 13, 134, 154

-par_threshold[n] オプション. 13, 134, 154

-parallel オプション 13, 134, 136, 154

PATH 環境変数..... 49

-pc32 オプション..... 13

-pc64 オプション..... 13

-pc80 オプション..... 13

-pch オプション..... 13, 57

-pch_dir オプション..... 13, 57

-pcn オプション..... 86

pgopti.dpi 112

pow ライブラリ関数 206

-prec_div オプション 13, 85, 86

-prof_dir オプション.....13, 111

PROF_DIR 環境変数.....112

PROF_DUMP_INTERVAL 環境変数 131

-prof_file オプション 13

-prof_format_32 オプション.....13

-prof_gen[x] オプション .13, 108, 109, 111

PROF_NO_CLOBBER 環境変数112

-prof_use オプション 13, 108, 109

profmerge.....112

Q

-Qinstall オプション 13

-Qlocation オプション.....13

-Qoption オプション 13, 99, 101

-Qoption 指定子.....99

-qp オプション 13

R

-rcd オプション 13, 86

remainder ライブラリ関数 218

remquo ライブラリ関数 218

-restrict オプション 81

rint ライブラリ関数..... 215

round ライブラリ関数..... 215

S

-S オプション 13

scalb ライブラリ関数..... 206

scalbln ライブラリ関数..... 206

scalbn ライブラリ関数..... 206

-shared オプション 13

-shared-libcxa オプション 13

sin ライブラリ関数 200

sincos ライブラリ関数..... 200

sincosd ライブラリ関数..... 200

sind ライブラリ関数 200

sinh ライブラリ関数 204

sinhcosh ライブラリ関数..... 204

SMP..... 156

-sox[-] オプション..... 13

sqrt ライブラリ関数..... 206

-static オプション 13

-static-libcxa オプション..... 13

-std=c99 オプション..... 13

-strict_ansi オプション 13, 81

-syntax オプション.....13

T

-T オプション.....13

tan ライブラリ関数.....200

tand ライブラリ関数.....200

tanh ライブラリ関数.....204

tgamma ライブラリ関数.....211

TMP 環境変数 49

-tpp1 オプション 13

-tpp2 オプション 13

-tpp5 オプション 13, 90

-tpp6 オプション 13, 90

-tpp7 オプション 13, 90

trunc ライブラリ関数.....215

U

-u オプション 13

-U オプション 13

unix 事前定義済みマクロ 46

-unroll[n] オプション 13

-unroll0 オプション 13

unwinder ライブラリ 64

-use_asm オプション 13

-use_msasm オプション 13

-use_pch オプション 13, 57

V

-v オプション..... 13

-vec_report[n] オプション..... 13, 135

W

-w オプション..... 13

-Wall オプション 13

-Wbrief オプション 13

-Wcheck オプション..... 13

-wd オプション..... 13

-we オプション..... 13

-Werror オプション 13

-Wl オプション 13

-wn オプション..... 13

-Wp64 オプション 13

-wr オプション..... 13

-ww オプション 13

X

-X オプション .. 13, 32, 52, 81, 86, 92, 131, 135, 136

-Xa オプション..... 13

-Xc オプション 13

xild.....102, 103, 105, 106

-Xlinker オプション.....13

Y

y0 ライブラリ関数.....211

y1 ライブラリ関数.....211

yn ライブラリ関数.....211

Z

-Zp オプション.....13

あ

アプリケーションの時間測定.....184

アライメント.....56

い

インクルード・ファイル

検索.....52

インクルード・ファイル.....52

インテル® 数値演算ライブラリ64, 200, 204, 206, 211, 215, 218, 219, 224

インテル拡張機能.....168

インライン展開.....101

お

オプション

クイック・リファレンス.....13

デフォルト.....37

新しいオプション.....10

対応表.....32

く

クラス・ライブラリ

概要

浮動小数点ベクトルクラス ... 419, 420, 421, 422, 423, 426, 427, 428, 431, 433, 435, 436, 437, 441

整数ベクトルクラス.. 391, 393, 394, 396, 397, 398, 400, 402, 403, 405, 407, 410, 416, 417, 418

クラス・ライブラリ . 384, 385, 386, 389

グローバル・シンボル.....67

こ

コード・カバレッジ・ツール.....113

コンパイル

make.....42

コマンドライン.....40

フェーズ.....39

リンク.....64

制御.....53

代替ツールと代替パス.....55

非共有ライブラリ.....67

コンパイル.....53

さ

サポート 4

し

しきい値制御 154

シンボル・プリエンプション 69

す

ストリップ・マイニング.....141

そ

ソフトウェアのパイプライン化 176

て

データのアライメント.....147

データ依存性.....138

テスト・プライオリタイゼーション・ ツール 121

デノーマル結果.....56

デノーマル結果のフラッシュ.....56

デフォルト

コンパイラ・オプション 37

コンパイラ動作 39

ふ

ファイル

インクルード 52

コンパイラ入力 42

プリコンパイル済みヘッダ 57

応答 51

設定 50

プリコンパイル済みヘッダ

ソースファイルの管理..... 57

プリコンパイル済みヘッダ 57

プリプロセッサ

オプション 44

プロファイル情報 . 128, 129, 130, 131

へ

ベクトライザ 134, 135, 136, 138, 139, 140, 141, 142, 147

ま

マトリックス乗算 150

ら

ライセンス 5

ライブラリ

スタティック 60

リンク 60

管理..... 66

共用..... 60

ライブラリ 64

り

リンク.....60

る

ループ変換.....132

漢字

応答ファイル.....51

環境

iccvars.sh での設定.....40

カスタマイズ.....49

変数.....49

関数分割.....109

共有ライブラリ.....66

言語の適合性.....81

構造体タグのアライメント.....56

最適化

インテル® プロセッサ.....90, 92, 93

プロシージャ間....98, 99, 101, 102, 103, 104, 105, 107

プロファイルに基づく108, 109, 111, 112, 113, 121, 128, 129, 130, 131

ベクトル化134, 135, 136, 138, 139, 140, 141, 142, 143, 147, 150

高水準言語.....131, 132

制限..... 85

浮動小数点の精度 86, 89

並列プログラミング.. 134, 151, 152, 154, 156, 157, 160, 162, 163, 164, 165, 167, 168, 174

最適化..... 84

自動並列化..... 151

数値演算ライブラリ..... 64

整数ベクトルクラス 391

静的リンク..... 60

設定ファイル 50

組込み関数

Itanium® プロセッサ341, 344, 347, 351, 354

MMX® テクノロジ ... 244, 246, 249, 252, 253, 254, 256

ストリーミング SIMD 拡張命令 257, 258, 262, 263, 270, 273, 274, 275, 276, 277, 280, 285, 287, 289, 290, 292

ストリーミング SIMD 拡張命令 2

整数 ... 312, 318, 319, 323, 325, 326, 327, 329, 333, 334, 337

浮動小数点. 293, 296, 297, 304, 307, 309, 310, 311

ストリーミング SIMD 拡張命令 2 292

データのアライメント. 362, 363, 365

概要 232
各種のプロセッサでの使用367, 368, 370, 372, 377
使用するメリット.....233
使用する構文.....236
新しいインテル® プロセッサ.....338, 340
組込み関数 79
対称型マルチプロセッシング.....156
代替ツールと代替パス 55
中間言語 103
動的リンク 60
特徴
新機能.....3
利点 4
配列 142
表記法
クラス・ライブラリ用 7
コンパイラ・オプション 10
組込み関数 7
本書 7
浮動小数点ベクトルクラス..... 419
変数
環境 49
環境設定 40
免責条項 1
要件
ソフトウェア 5
ハードウェア 5