ストリーミング SIMD 拡張命令 (Streaming SIMD Extensions)の ソフトウェア規則

バージョン 2.1

1999 年 1 月

注文番号: 243873J-002

【輸出規制に関する告知と注意事項】

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

【資料内容に関する注意事項】

● 本ドキュメントの内容を予告なしに変更することがあります。

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

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

●本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。

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

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

本資料の内容についてのお問い合わせは、下記までご連絡下さい。

インテル株式会社　資料販売センター

〒305-8603　筑波学園郵便局　私書箱 115 号

Fax: 0120-478832

「予約」もしくは「未定義」と記された機能や命令は、実際には存在しないものであるか、存在する場合でもその説明は暫定的なものになります。これらの定義は将来変更される可能性があり、また、将来変更されたことにより生じる矛盾や互換性の欠如などに対してはインテルはいっさい責任を負いません。

Pentium® II と Pentium III プロセッサにはエラッタと呼ばれる設計上の欠陥やエラーが含まれることがあり、公表された仕様どおりの製品となっていないことがあります。現時点で判明しているエラッタについては、ご要望があればいつでも入手することが可能です。

*サードパーティのブランドおよび商品名の所用権は、それぞれ各社に帰属します。

Copyright © Intel Corporation 1998, 1999

目次

1 はじめに 1
2 スタックのアライメント 1
2.1 アライメントの合ったespベースのスタック・フレーム 2
2.2 アライメントの合ったebpベースのスタック・フレーム 3
2.3 スタック・フレームの最適化 5
2.4 インライン・アセンブリとebx 5
3 レジスタ渡し規則 5
4 変数のアライメント 6

改訂履歴

改訂	改訂履歴	日付
2.1	リリース用の改訂	1999年1月

1 はじめに

ストリーミング SIMD 拡張命令が最高の性能を発揮するためには、メモリ・オペランドのアライメントを 16 バイトに合わせる必要がある。しかし、Microsoft Visual C++コンパイラまたは以前のバージョンの Intel® C/C++コンパイラで採用された、既存の IA-32 用ソフトウェア規則(stdcall, cdecl, fastcall)は、特定のローカル・データおよびパラメータの 16 バイト・アライメントを保証するための機構を提供しない。また、新しいレジスタ・ファイルが追加されたため、いくつかのパラメータがメモリ内ではなくレジスタ内で渡される可能性がある。したがって、Intel C/C++コンパイラでストリーミング SIMD 拡張命令および新しい`__m128` データタイプをサポートするために、IA-32 ソフトウェア規則が次のように拡張された。

- ストリーミング SIMD 拡張命令データを使用する関数は、16 バイトにアライメントの合ったスタック・フレームを提供する。
- 通常は、引数ブロック内でパディングを行うことで、`__m128` パラメータのアライメントが合わされる。
- `__m128` パラメータは、レジスタで渡すことができる。

この文書では、Intel C/C++コンパイラの現在のリリースに導入された、これらの新しい規則について説明する。

また、本書では、拡張された属性`__declspec(align())`についても説明する。この属性を使用して、データのアライメントを要求することができる、

2 スタックのアライメント

ここでは、Intel C/C++コンパイラがスタック・フレームのアライメントを 16 バイト境界に合わせる方法について説明する。ローカル変数に`__m128` データタイプを使用するルーチンをサポートするには、スタック・フレームのアライメントを 16 バイトに合わせる必要がある(ローカル変数が xmm レジスタからスタック・フレームに割り当てられる可能性がある場合)。ストリーミング SIMD 拡張命令のメモリ参照のアライメント要件をサポートするためには、この必要条件を満たさなければならない。また、この方法を使用して、`__m64` および `double` タイプのデータのアライメントを改善することもできる。

コンパイラは、スタック・フレームそれ自体の 16 バイト・アライメントを保証できない限り、スタック・フレームに割り当てられる変数について、変数のベースのアライメントを保証することができない。Microsoft Visual C++コンパイラおよび旧バージョンの Intel C/C++コンパイラでサポートしていた規則のように、以前の IA-32 ソフトウェア規則は、個々のスタック・フレームの 4 バイト・アライメントだけを保証している。したがって、例えば Microsoft Visual C++でコンパイルされた関数から呼び出される関数は、使用されたフレーム・ポインタのアライメントが 4 バイトに合っていると見なせるだけである。

以前のバージョンの Intel C/C++コンパイラは、メイン・ルーチンの始まりの部分でスタック・フレーム・ポインタを動的に調整し、コンパイルされる関数の 8 バイト・アライメントを保つことによって、8 バイトにアライメントの合ったスタック・フレームを提供しようと試みる。しかし、明らかに、この手法は次の理由で適用範囲が制限される。まず、メイン・ルーチンは、Intel C/C++コンパイラでコンパイルしなければならない。また、他のコンパイラでコンパイルされた呼び出しツリー内に関数が存在しない場合がある(コールバックとして登録されたルーチンでは、このような可能性がある)。さらに、パラメータの適切なアライメントがサポートされていない。

この問題の解決策は、関数のエントリ・ポイントを 4 バイト・アライメントと見なすことである。関数が 8 バイトまたは 16 バイト・アライメントを必要とする場合は、動的にスタックのアライメントを合わせるコードが挿入され、図 1 に示す 2 つのスタック・フレームのうちいずれかが得られる。

最適化のために、別のエントリ・ポイントが作成され、次の処理が行われる。このエントリ・ポイントは、呼び出し元がスタックの適切なアライメントを保証している場合に呼び出される。スタックの適切なアライメントが保証されている場合は、コール・グラフの分析によって、通常の(アライメントの合っていない)エントリ・ポイントの呼び出しが最適化され、アライメントの合った別のエントリ・ポイントの呼び出しで置き換えられる。さらに、コール・グラフ全体を通じて、関数のアライメントが修正され、アライメントの合っていないエントリ・ポイントの呼び出しの数が最小限に抑えられる。この処理の例として、スタックのアライメントが 4 である関数 F が、多くの呼び出しサイトおよびループ内で、アライメントが 16 である関数 G を呼び出す場合を考える。この場合、コンパイラは、F のアライメントを 16 に変更し、G に対するすべての呼び出しを、アライメントの合ったエントリ・ポイントに移動する。これにより、制御がアライメントの合っていないエントリ・ポイントを通る回数が最小限に抑えられる。

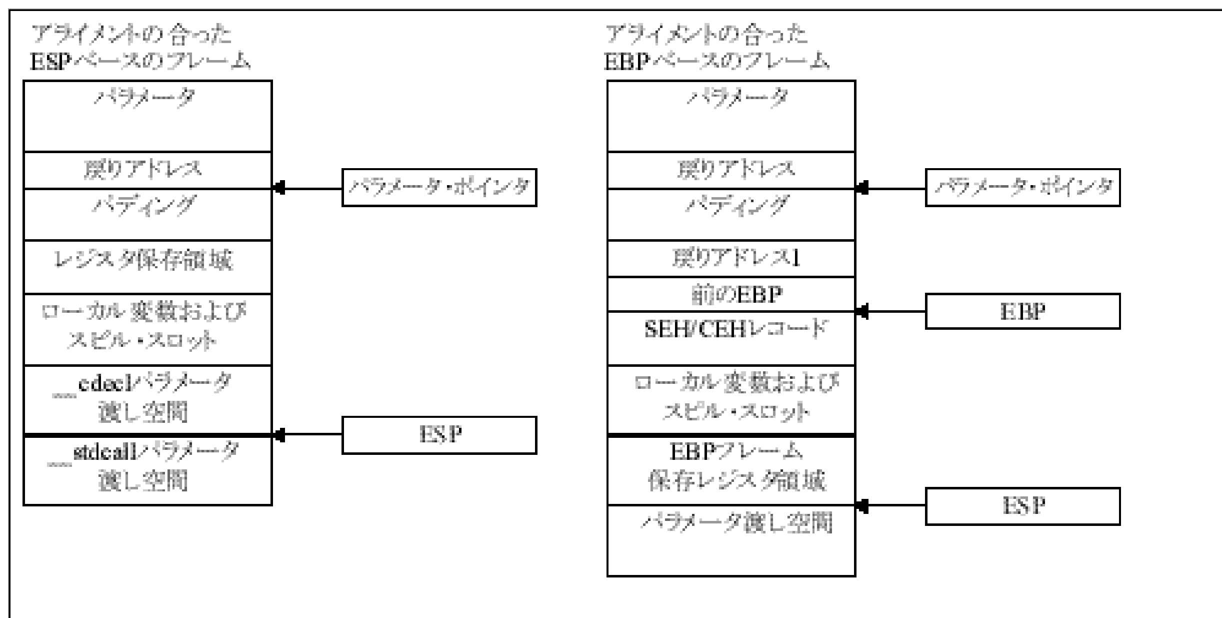

図 1

2.1 アライメントの合ったespベースのスタック・フレーム

コンパイラは、esp ベースのスタック・フレームを作成するとき、図 2 の例に示すように、戻りアドレスとレジスタ保存領域の間にパディングを行う。アライメントの合った esp ベースのフレームを使用できるのは、デバッグ情報が要求されない、例外処理をサポートする必要がない、インライン・アセンブリが使用されない、および関数内に `alloca` への呼び出しがないという条件を満たす場合に限られる。これらの条件が満たされない場合は、アライメントの合った ebp ベースのフレームを使用しなければならない。esp ベースのフレームを使用する場合は、戻りアドレス、保存レジスタ、ローカル変数、レジスタ・スピル領域、およびパラメータ空間の各サイズの合計が、16 バイトの倍数でなければならない。これにより、パラメータ空間のベースは、16 バイトにアライメントを合わせられる。また、stdcall 関数のパラメータを渡すために予約されている空間も、すべて 16 バイトの倍数になっていなければならない。つまり、stdcall 関数の呼び出しのためにプッシュされるパラメータのサイズが 16 の倍数でない場合は、呼び出し元がスタック空間の一部を調整する必要がある。呼び出し元がこの処理を実行しないと、スタック・ポインタが呼び出し前の値に戻らない。

上の例では、呼び出し元から送られるアライメントの点(戻りポインタ ebx および edx)の後のスタック上に 12 バイトがある。したがって、スタック・ポインタに 4 バイトを追加して、アライメントを合わせる必要がある。ローカル変数用に 16 バイトのスタック空間が必要であるとすると、コンパイラは 16 + 4 = 20 バイトを esp に追加し、esp のアライメントを 0 mod 16 のアドレスに合わせる。

注 A：アライメントの合ったエントリ・ポイントは、パラメータ・ブロックの始まりのアライメントが合っていると見なす。戻りポインタがプッシュされているため、スタック・ポインタは 12 mod 16 の境界に置かれる。したがって、アライメントの合っていないエントリ・ポイントは、スタック・ポインタを強制的にこの境界に合わせる必要がある。

注 B：common の部分のコードは、スタックが 8 mod 16 の境界上にあると見なし、スタック・ポインタのアライメントが 0 mod 16 の境界に合わせられるように、必要な空間をスタックに追加する。

```
void _cdecl foo(int k) {
  int j;
foo:                                  // see note A
        push  ebx
        mov   ebx, esp
        sub   esp, 0x00000008
        and   esp, 0xfffffff0
        add   esp, 0x00000008
        jmp   common
foo.aligned:
        push  ebx
        mov   ebx, esp
common:                               // see note B
        push  edx
        sub   esp, 20

 j = k;
        mov   edx, [ebx+8]
        mov   [esp+16], edx

  foo(5);
        mov   [esp], 5
        call  foo.aligned


  return j;
        mov   eax, [esp+16]
        add   esp, 20
        pop   edx
        mov   esp, ebx
        pop   ebx
        ret
}
```

図 2

2.2 アライメントの合ったebpベースのスタック・フレーム

ebp ベースのフレームも、戻りアドレスの直前にパディングが行われる。ただし、このフレームの特徴として、戻りアドレスが実際にはスタック内の 2 箇所に存在する場合がある。パディングする必要があり、関数に対する例外処理が有効になる場合は、必ず 2 箇所に戻りアドレスが発生する。図 3 は、このタイプのフレーム用に生成されるコードの例を示している。スタック内の戻りアドレスの位置は、12 mod 16 に合わせられる。これは、ebp の値が常に条件(ebp & 0x0f) == 0x08 を満たすという意味である。この場合は、戻りアドレス、前の ebp、例外処理レコード、ローカル変数、およびスピル領域の各サイズの合計が、16 バイトの倍数にならなければならない。また、パラメータ渡し空間も、常に 16 バイトの倍数でなければならない。stdcall 関数の呼び出しで、プッシュされているパラメータ・ブロックのサイズが 16 の倍数でない場合は、呼び出し元が多少のスタック空間を確保する必要がある。

```
void _stdcall foo(int k) {
   int j;
foo:
        push  ebx
        mov   ebx,esp
        sub   esp,0x00000008
        and   esp,0xfffffff0
        add   esp,0x00000008     // esp is (8 mod 16) after add
        jmp   common
foo.aligned:
        push  ebx                // esp is (8 mod 16) after push
        mov   ebx,esp
common:
        push  ebp                // this slot will be used for duplicate return ptr
        push  ebp                // esp is (0 mod 16) after push (rtn,ebx,ebp,ebp)
        mov   ebp,[ebx+4]        // fetch return pointer
        mov   [esp+4],ebp        // and store relative to ebp    (rtn,ebx,rtn,ebp)
        mov   ebp,esp            // ebp is (0 mod 16)
        sub   esp,28             // esp is (4 mod 16)   // see note C
        push  edx                // esp is (0 mod 16) after push
                                 // the goal is to make esp and ebp (0 mod 16) here
 j = k;
        mov   edx,[ebx+8]        // k is (0 mod 16) if caller aligned his stack
        mov   [ebp-16],edx       // J is (0 mod 16)

  foo(5);
        add   esp,-4             // normal call sequence to unaligned entry
        mov   [esp],5
        call  foo                // for stdcall, callee cleans up stack

  foo.aligned(5);
        add   esp,-16            // aligned entry, this should be a multiple of 16
        mov   [esp],5
        call  foo.aligned
        add   esp,12             // see note D

  return j;
        mov   eax,[ebp-16]
        pop   edx
        mov   esp,ebp
        pop   ebp
        mov   esp,ebx
        pop   ebx
        ret   4
}
```

図 3

注C：ここでローカル変数を割り当てることができる。ただし、この値は、保存されたレジスタをプッシュした後、espが 0 mod 6 になるように調整する必要がある。

注D：呼び出しの直前に、espは 0 mod 16 になる。アライメントを保つため、espは 16 バイトずつ調整しなければならない。呼び出し先がstdcall呼び出しシーケンスを使用する場合は、スタック・ポインタは呼び出し先によって元に戻される。実際には 16 バイトでなく 4 バイトだけが渡されているので、それを補正するために、最後に 12 バイトが加算される。したがって、残った調整については、呼び出し元が責任を負わなければならない。

2.3 スタック・フレームの最適化

Intel C/C++コンパイラは、アライメントの合ったフレームのセットアップと使用効率を改善する最適化機能を持っている。最適化は、次のように行われる。

- プロシージャがスタック・フレームの 16 バイト・アライメントを変更しないことが確実であり、16 バイト・アライメントを必要とする他のプロシージャを呼び出す場合は、アライメントを合わせるための不要なコードはすべて省略され、呼び出し先のアライメントの合ったエントリ・ポイントが呼び出される。
- 静的な関数が 16 バイト・アライメントを要求しており、その関数を呼び出す他の関数も確実に 16 バイト・アライメントを要求する場合は、その関数内のアライメント・コードは省略される。アライメントの合っていないフレームが、この関数のエントリ・ポイントになることはあり得ない。したがって、コンパイラは引数ブロックを指示する ebx を使用せず、別のエントリ・ポイントも作成しない。

2.4 インライン・アセンブリとebx

アライメントの合ったフレームを使用する場合は、ebx レジスタを使用して引数ブロックの状態を監視するため、通常はインライン・アセンブリ・ブロック内で ebx レジスタを修正してはならない。プログラマが ebx を修正できるのは、引数にアクセスする必要がなく、元の ebx を保存し、関数の終了前に復元する場合に限られる(esp は関数の終わりの部分で ebx を基準として復元されるため、この処理が必要になる)。

3 レジスタ渡し規則

3 つの一般的な IA-32 呼び出し規則が次のように拡張され、新しいレジスタ・セットをサポートするようになった。

- 最初の 3 つの`__m128` パラメータは、レジスタ xmm0, xmm1, および xmm2 内で渡される(「レジスタ内の引数」 "args in regs")。その他の`__m128` パラメータは、通常どおりスタック上で渡される。
- `__m128` の戻り値は、レジスタ xmm0 内で渡される。
- レジスタ xmm0～xmm7 は呼び出し元保存である。

呼び出し元は、通常であれば最初の 3 つの`__m128` パラメータが現れるはずの引数ブロック内の位置に、スペースを確保しなければならない。通常は、呼び出し元はこれらの位置を空白のままにしておき、必要に応じて呼び出し先が xmm0, xmm1, および xmm2 レジスタの「ホーム」として使用する。

Intel C/C++コンパイラには、stdarg.h および varargs.h ヘッダの新しいバージョンが付属している。これらの新しいヘッダは、__m128 データを含む可変引数のリストをサポートする(既に説明したように、これらの位置には、アライメントの合ったパラメータに合わせて、必要に応じてパディングされる)。新しい規則は、可変引数のリストを持つ関数を、その関数の呼び出しが行われる前にプロトタイピングするように要求している。この場合に限り、呼び出し元は、レジスタ xmm0, xmm1, および xmm2 内のデータに対応するスタック上の位置を充填しなければならない。__m128 データを含む可変変数のリストを持つ非プロトタイピング関数の呼び出し元は、スタック上およびレジスタ内の両方でパラメータを渡さなければならない。

4　変数のアライメント

__declspec(align())属性を使用して、通常より厳密に変数のアライメントを設定することができる。例えば、ストリーミング SIMD 拡張命令で浮動小数点の配列を使用する場合、その配列に__m128 データが入っているかのように見なすには、この方法が効果的である。また、この方法は、キャッシュラインのアライメントを強制的に設定する場合にも効果的である。

拡張された属性 align の構文は次のとおりである。

```
__declspec(align(integer-constant))
```

integer-constant は、2 の整数乗の値(最大 32 まで)である。

次のように、変数のアライメントを大きくすることができる。

```
__declspec(align(16)) float buffer[400];
```

変数のアドレス(buffer)は、強制的に 0 mod 16 に設定される。これで、変数バッファを使用する際に、変数バッファが__m128 タイプのオブジェクトを 100 個含んでいるかのように見なすことができる。便利なように、xmmintrin.h インクルード・ファイルは、_MM_ALIGN16 を__declspec(align(16))として定義する。

別の方法として、次のような宣言を使用することもできる。

```
union {
        float f[400];
        __m128 m[100];
} buffer;
```

この場合は、共用体の中に__m128 データがあるため、デフォルトにより 16 バイト・アライメントが使用される。__declspec(align())を使用して強制的に 16 バイト・アライメントに設定する必要はない。このような場合、通常は、単に__declspec(align())を使用してアライメントを強制的に設定するよりも、できるだけ上記のような共用体を使用する方がよい。

ただし、任意のデータ集合のキャッシュラインのアライメントを強制的に設定することが有益な場合もある。例えば、3 つのローカル変数 i, j、および k がある場合、それらの宣言を次のように変更することで、3 つの変数を強制的に 1 つのキャッシュライン上に置くことができる。

```
__declspec(align(16)) struct {
        int i, j, k;
} x;
```

もちろん、これらの変数への参照は、それぞれ x.i, x.j、および x.k として書き直さなければならない。

C++では、次のように、クラス/構造体/共用体タイプのアライメントを強制的に設定することもできる(C ではこの処理はできない)。

```
struct __declspec(align(16)) my_m128 {
        float f[4];
};
```

しかし、このようなクラスのデータをコンパイラのストリーミング SIMD 拡張命令内部関数で使用する場合は、通常は共用体を使用して、このような「タイプ・パニング(type punning)」を明示的に指定する方がよい。C++では、無名の共用体を使用して、この処理をより簡単に実行できる。

```
class my_m128 _
        union {
                __m128 m;
                 float f[4];
        };
};
```

この例では、共用体が無名なので、my_m128 の臨時のメンバ名として名前 m と f を使用できる。

ただし、C または C++でクラス、構造体、または共用体のメンバに__declspec(align())を適用した場合は、__declspec(align())は無効になる。