TEXAS INSTRUMENTS
www.tij.co.jp

参 考 資 料

Application Report
JAJA022 WAS
SCJA002-Oct. 2002
Rev. 1.0

TMS320C6000 Optimization Technique

–Application note--

大下 和昭 / 関口 健雄　　日本テキサス・インスツルメンツ(株)

目次

注意：本レポートに含まれる総ての項目は最適化を助けるためのものであり、これらにより必ず生成効率が改善されることを保証するものではありません。

１．　概要

C6000 シリーズは VLIW アーキテクチャを採用しており、1 サイクルで最大 8 個の命令を実行します。
C62x／C67x コアは 2 つの乗算器、6 つの ALU を含む 8 個の機能ユニットを持っており、コンパイラやアセンブリ・オプティマイザにより最下層のループにソフトウエア・パイプラインを構成し、効率的な並列処理を行う事ができます。
また、C64x コアは SIMD 命令をサポートしており、次の例の様にさらに効率的な並列処理を実行できます。

［コンパイラの出力例］7 命令（内 3 個は SIMD 命令）を 1 サイクルで実行

```
           ADD2     .L2X    B5,A7,B5          ; |310|
|| [ B1]   BDEC     .S2     L9,B              ;
||         ADD2     .S1     A5,A16,A4         ; |309|
||         STW      .D2T1   A6,*++B25(8       ; |326|
||         MV       .L1     A4,A6             ; @ ^ |263|
||         MPYSU4   .M2     B23,B7,B5:B4      ; @ ^ |292|
|| [!B0]   LDW      .D1T2   *+A26[A24],B18    ; @|268|
```

信号処理では大量のデータを高速に処理する事が必要です。例えば、画像処理での 1 ラスタの画素数は A4，600dpi の場合、約 5000 となります。この回数の処理が連続して繰り返されますので、C6000 実行コードにおいて適切なソフトウエア・パイプラインが構成される事は極めて重要です。

本レポートでは、C6000 シリーズでの効率的なソフトウエア・パイプラインを作成するための注意事項と、C64x コアでの SIMD 命令の使用について説明します。

２． C6000 ソフトウェアパイプラインについて

C6000 は並列に動作する 8 個の演算ユニットから構成されているため、これらを効率よく並列動作させることにより、非常に高速な動作が可能になります（プロセッサ自体のフェッチ/デコード/実行から構成される、一般的にいうパイプラインとは別のものです）。
この効率的な動作を実現させるための手法として、繰り返しループに対するソフトウェアパイプラインという手法があります。これはループ内において、例えば、ループカウンタが１のときと２のときの処理を同時に行うことにより並列度をあげるためのものです。下記、積和演算のループを例にとってソフトウェアパイプラインの概要を説明します。C6000 C コンパイラは –o2 以上の最適化オプションにより自動的にソフトウェアパイプライン化された高速なコードを生成します。

[積和のループ]

```
loop: LDW    .D1    *A1, A4
      LDW    .D2    *B1, B4
      MPY    .M1    A4, B4, A6
      ADD    .L1    A6, A7, A7
      SUB    .S1    B0, 1, B0
      B      .S2    loop
```

[積和のループの実行例]

積和ループは下図(左)のように実行されますが、この方法ではループ処理に
3x(ループ回数)ステップが必要です(ディレイロット無視)。
一方で、下図(右)のように同じサイクルにおいてループの次の周回の処理を一緒に行うことにより、実行ステップ数を減らすことができます。このように各演算ユニットを効率よく使用する方法論をソフトウェアパイプラインと呼んでいます。
この例では、ソフトウェアパイプライン化されたコードは
ループ回数＋4 ステップ
でのループの実行が可能になります。

［ソフトウェアパイプライン化された積和のループ - ディレイスロットは無視］

```
        LDW   .D1    *A1, A4
||      LDW   .D2    *B1, B4

        LDW   .D1    *A1, A4
||      LDW   .D2    *B1, B4
||      MPY   .M1    A4, B4, A6

loop:   LDW   .D1    *A1, A4
||      LDW   .D2    *B1, B4
||      MPY   .M1     A4, B4, A6
||      ADD   .L1    A6, A7, A7
|| [B0] SUB   .S1    B0, 1, B0
|| [B0] B     .S2    loop

        MPY   .M1    A4, B4, A6
||      ADD   .L1    A6, A7, A7

        ADD   .L1    A6, A7, A7
```

３．　ソフトウェアパイプラインが生成されない条件について

それぞれのループにおいてソフトウェアパイプラインが生成されるためにはいくつかの条件を満たす必要があります。しかしながら、ソフトウェアパイプラインの生成には多岐の条件がからみあい複雑になりますので、ここでは簡単のためにソフトウェアパイプラインが不生成となる条件をあげます。
実際のコーディング時、および、コード最適化の際には下記の項目を避けるようにしてください。また、効率的なコードを記述するためのより具体的な方法論については 4 章をご覧ください。

① ループではない場合
　ソフトウェアパイプライン生成の定義から外れています。

② switch 文や break を含むループ

③ Intrinsics 関数(組込み関数)以外の関数コールを含むループ(-o2 以下の場合)
　プログラムフローの構造が崩れてしまうため生成されません。
　ただし、Intrinsics 関数では対応するアセンブラ命令に置き換わることからソフトウェアパイプラインの生成に影響を及ぼしません。

④ ループ処理を開始するために前のループ処理の結果を必要とするような場合
　同じサイクルにおいてループ内の異なる周回の処理を行いますので、前の周回における演算結果を次の周回で使用するような場合にはソフトウェアパイプラインが生成されません。

⑤ 複雑な条件分岐を含むループ
　C6000 では条件付命令実行をサポートしていることから条件分岐の効率的な実行が可能ですが、条件が複雑になってくるとソフトウェアパイプラインが生成されません。

⑥ ループ内のコードサイズが大きいとき
　あまりに大きいループではＣコンパイラがソフトウェアパイプライン生成を行いません。このような場合には、ループを複数に分割する事により解決できます。

⑦ ループ内で条件によりループカウンタの値が変更される、もしくは、変数による演算結果によりループカウンタが変更される場合。

⑧ レジスタの本数が足りない場合
　コンパイラからフィードバックが can not allocate registers となるのでわかります。

４．　Ｃソースコードでの効率的なコーディングのための注意事項

１）ループ内のコードサイズ

C6000 コンパイラは、最下層のループにソフトウエア・パイプラインを構成します。1 つのループ内のコードサイズが大き過ぎる場合、並列化の条件が複雑になりソフトウエア・パイプラインが作成できなかったり並列度が落ちます。
この場合、適切に機能を分解しそれぞれの機能の関数として実現する事が必要です。

[ソフトウエアパイプラインの作成が困難な例]

```
画像処理関数( )
{
        for( i = 0; i < iNumPixels; i++ ) {
                色変換処理 ;                    // 1 画素
                スケーリング処理 ;              // 1 画素
                ディザ処理 ;                    // 1 画素
        }
}
```

[ソフトウエアパイプラインの作成が可能な例]

```
画像処理関数( )
{
        色変換処理関数( ) ;                     // 1 ラスタ
        スケーリング処理関数( ) ;               // 1 ラスタ
        ディザ処理関数( ) ;                     // 1 ラスタ
}
```

C のソースコードでのサイズは一概に言えませんが、C62x/C67x コアの場合、数行～30 行程度、C64x の場合はレジスタ数が 2 倍でありコードサイズも C62x/C67x の場合の 2 倍程度までが比較的良好なコードを生成できます。

２）ループ内での if(else)のネスティング

ループ内に if(else)のネスティングがある場合、その多くはソフトウエア・パイプラインが作成されません。また、if～else if～else による条件分岐に対してはソフトウェアパイプラインが生成されませんので注意が必要です。

[ソフトウエアパイプラインの作成が困難な例]

```
      for( i = 0; i < iNumPixels; i++ ) {
        if( 条件 1 ) {
                処理 1 ;
                if( 条件 2 ) {
                  処理 2 ;
                }
        }
      }
```

[ソフトウエアパイプラインの作成が可能な例]

```
for( i = 0; i < iNumPixels; i++ ) {
   if( 条件 1 ) {
      処理 2 ;
   }
   if(( 条件 1 ) && ( 条件 2)) {
      処理 2 ;
   }
}
```

[ソフトウエアパイプラインの作成が困難な例] Run Length のカウント処理

```
for( i = 1; i < 64; i++ ) {
   ...
   if( sData == 0 ) {
       sRun++;
       if( sRun >= 16 ) {
          処理 ;
          sRun = 0;
       }
   }
   else {
       ...
   }
}
```

[ソフトウエアパイプラインの作成が可能な例]

```
for( i = 1; i < 64; i++ ) {
    if( sData == 0 ) {
        sRun++;
    }
    else {
        ...                                  // sRun は変化しない
     }
     if( sRun >= 16 ) {
        処理 ;
        sRun = 0;
     }
}
```

３）ループ内での Switch
ループ内に Switch 構造がある場合は、ソフトウエア・パイプラインが作成されません。プログラム自体を切り替える場合の対策は困難ですが、通常はデータやポインタを切り替える事で Switch 構造のない形にできます。

[例] データを切り替える。
```
iData = iArray[Switch 条件];
```
ポインタを切り替える。
```
piData = piPointerArray[Switch 条件]
iData = *piData;
```
シフトによりデータ（4 種類の係数）を切り替える。
```
bCoefficient = nPackedCoefficients >> (8*Switch 条件);
```

４）ループ内での Break
ループ内での Break も避ける必要があります。

Break を避ける方法は処理内容に依存しますが、予め Break 条件を高速にサーチしておいて、Break までの回数をソフトウエアパイプラインが構成された高速なループ処理とする方法が効果的です。

<注意> ワーストケースの処理時間があまり問題にならず、平均的な処理時間を早くする必要のある応用では、次の例の様にソフトウエアパイプラインを構成せずに Break させた方が早い事があります。

[例] JPEG ハフマンコードのデコード（Break タイプ）
```
JDEC_DecodeHuffman( )
{                                                   // 1 ブロック
      for( i = 0; i < 64; i++ ) {
          デコード処理 ;                            // 1 コード
          if( EOB を検出 ) {
              Break;
          }
      }
}
```

５）ループ内での関数呼び出し
最適化オプションが-o2 以下の場合、ループ内に関数呼び出しがあるとソフトウエア・パイプラインが構成されません。呼び出される関数が複雑な処理を行う場合は、1）項の様に適切に処理を分割する必要があります。軽い処理の場合は、関数のプロトタイピングにおいて inline キーワードを付け、inline 展開させる事ができます。

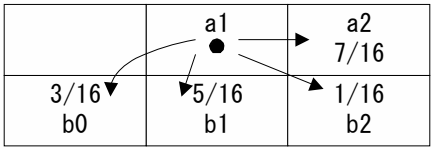
[例]
```
inline int Function( int iCoef, int iData );
```

<注意> -o3 オプションの場合は自動的に小さな関数のインライン展開を行います。

6）レジスタ・ベースの処理

C6000 は 32 本のレジスタ（C62x/C67x）または 64 本のレジスタ（C64x）を持つ、いわゆるレジスタ・マシンです。RISC CPU と同様、高速化のためにはメモリ・アクセスを少なくして、レジスタ間の演算で処理を行うことが有効です。
このため、極力データや係数をレジスタ（最下位の関数内で定義されるローカル変数）に置きます。

[例] 誤差拡散処理
下記 a1,a2,b0,b1,b2 は誤差拡散処理中はレジスタに置きます。



7）メモリ・アクセスを減らす

メモリ・アクセス、特に Read アクセスを減らす必要があります。具体的には、6）項の様にデータや係数をレジスタに置く事や、BYTE データを 4 バイトまとめて int でアクセスしたり、short データを 2 データまとめて int でアクセスする事は大きな効果があります。C64x の場合は、double アクセスも可能です。

<注意> この様な int/double アクセスのため BYTE や short データを、4 または 8 バイトの境界にアラインしておきます。

[例] データバッファ

```
#pragma DATA_ALIGN( JENC_sCbBuffer, 8 );
short JENC_sCbBuffer[16][16];
```

[例] 構造体

```
typedef struct _JPEG_ENC_DQT {
    WORD        wLength;            // Lq
    BYTE        bPrecisionY;        // Pq/Tq
    BYTE        bPrecisionCbCr;     // Pq/Tq
    double      dDummy;
    short       sQTY[64];
    short       sQTCbCr[64];
    } JPEG_ENC_DQT;
```

8）データ型の統一

unsigned 型と signed 型を混在させるとコンパイラは躍起になって型変換処理を挿入します。どちらかで良い場合は統一する事によりコードがシンプルになります。

９）多次元の配列
多次元配列のどれかの入力がループ内で一定の場合は、次元を減らして使用します。

[例] ディザ・マトリクス（2 次元）を 1 次元として使用

```
pbMatrix = &bDitherMatrix88[iLineNum%8][0];

for(j = 0; j < iWidth; j++) {
                              …
   bSlice = pbMatrix[iCtr8];
        …
}
```

10) 配列 or ポインタ
C62x/C67x の場合はレジスタが不足する事が多く、ほとんどの場合ポインタが有利です。C64x の場合でもポインタを基本とする事を推奨します。

11) 入力ポインタと出力ポインタの分離
入力と出力が同じメモリの場合、ポインタは 1 本で済みますが、これを入力専用と出力専用の 2 本にすることにより、並列度が上がる場合があります。

[例]

```
void JENC_Quantization( short* psData, short* psRevQT )
{
   double               dTemp;
   int* piDataIn = (int*)psData;
   int* piDataOut = (int*)psData;
   ...
```

12) データの先読み
データの先読みをプログラムで記述する事により、並列度が上がる事があります。

[例] Run Length のカウント処理

```
sNextData = psBlockData[JENC_wZigZag[1]];
for( i = 1; i < 64; i++ ) {
      Data = sNextData;
      sNextData = psBlockData[JENC_wZigZag[i+1]];
      sData の処理...
```

13) 2 重ループの単一ループ化

次のようなループ回数の少ない２重ループは、単一ループに展開する事により、処理速度が向上する事があります。

[2 重ループの例]

```
for( i = 0; i < 8; i++ ) {
    処理 1；
    for( j = 0; j < 8; j++ ) {
        処理 2；
    }
}
```

[単一ループ]

```
for( i = 0; i < 8*8; i++ ) {
    if((i%8) == 0) {
        処理 1；
    }
    処理 2；
}
```

５．　Ｃコンパイラについて

１）コンパイラ・オプション

次のオプションの使用がデバッグ上、比較的便利だと思われます。Code Composer Studio の Project-Options メニューから設定して下さい。

(モジュール・デバッグ時)

-g –k -q –ml3

ここで、

-g	Full symbolic debug
-k	ASM ファイルを残す。
-q	Suppresses progress messages
-ml3	Far Model

（モジュール・デバッグ終了後）

-k -q –mt -o2 –ml3

ここで、

-mt	Ignore alias disambiguation
-o2	最適化レベル 2、ソフトウエア・パイプライン作成を指示。

２）コンパイラからのフィード・バック

次の様に、コンパイラからのフィード・バックがＡＳＭファイルに記述されます。
特に、「*」のついた項目と、下から 9 行目の ii = 数字（ここでは 4）に注目して下さい。

```
;*----------------------------------------------------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count                 : 1
;*      Known Maximum Trip Count                 : 536870911
;*      Known Max Trip Count Factor              : 1
;*      Loop Carried Dependency Bound(^)         : 0
;*      Unpartitioned Resource Bound             : 4
;*      Partitioned Resource Bound(*)            : 4
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     3        4*
;*      .D units                     4*       3
;*      .M units                     0        0
;*      .X cross paths               0        1
;*      .T address paths             4*       3
;*      Long read paths              1        1
;*      Long write paths             0        0
;*      Logical  ops (.LS)           2        2     (.L or .S unit)
;*      Addition ops (.LSD)          1        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             3        3
;*      Bound(.L .S .D .LS .LSD)     4*       4*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 4  Schedule found with 5 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages       : 3
;*
;*      Prolog not entirely removed
;*      Collapsed prolog stages       : 2
;*
;*      Minimum required memory pad   : 0 bytes
;*
;*      For further improvement on this loop, try option -mh16
;*
;*      Minimum safe trip count       : 2
;*----------------------------------------------------------------------------------------------------------------------*
```

６．　C64x での SIMD 命令の使用について

SIMD 命令は、2 または 4 データを並列に処理します。ループ内で 4 データの並列処理

が可能な場合、SIMD 命令を適用する事で SIMD 命令を使用しない場合の 2～3 倍の高い処理性能を得る事が可能です。
SIMD 命令を記述する方法は、① マクロ関数（Intrinsics）を使用して C 言語で記述する ② Assembly Optimizer を使用する ③ アセンブラで記述するなどがあります。
本レポートでは、C 言語で記述が可能な、マクロ関数（Intrinsics）を使用する方法を説明します。

６．１．　**8** ビットデータの **4** 並列処理（**1**）

画像処理ではフィルタ処理やデータの変換処理など、8 ビットデータと 8 ビットの係数で乗算または積和演算を行う処理が多く見られます。これらの処理は乗算回数が多く SIMD 命令が有効です。

[例] RGB to YCC 変換

16 ビットの変換係数を使用した場合の RGB to YCC 変換は、レベルシフトを含めて次の処理となります。

```
Y  = (( 9798*R + 19235*G +  3736*B + 16384 ) >> 15) - 128;
Cb = ( -5528*R - 10856*G + 16384*B + 16384 ) >> 15;
Cr = ( 16384*R - 13720*G -  2664*B + 16384 ) >> 15;
```

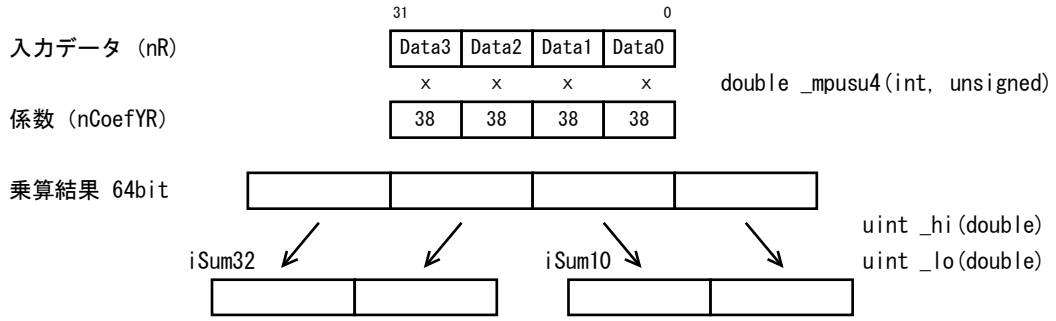
8 ビットの変換係数を使用した場合は次の処理となります。

```
Y  = (( 38*R + 75*G + 15*B + 64 ) >> 7) - 128;
Cb = ( -22*R - 42*G + 64*B + 64 ) >> 7;
Cr = (  64*R - 54*G - 10*B + 64 ) >> 7;
```

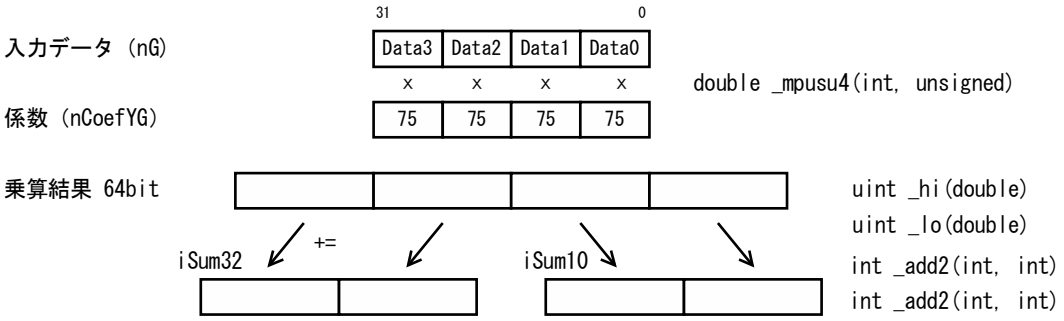
8 ビットの変換係数を使用した場合に、SIMD 命令を適用した処理は次のようになります。

① RGB 各 4 データをロード

② 4 データ(R)と 4 係数の乗算

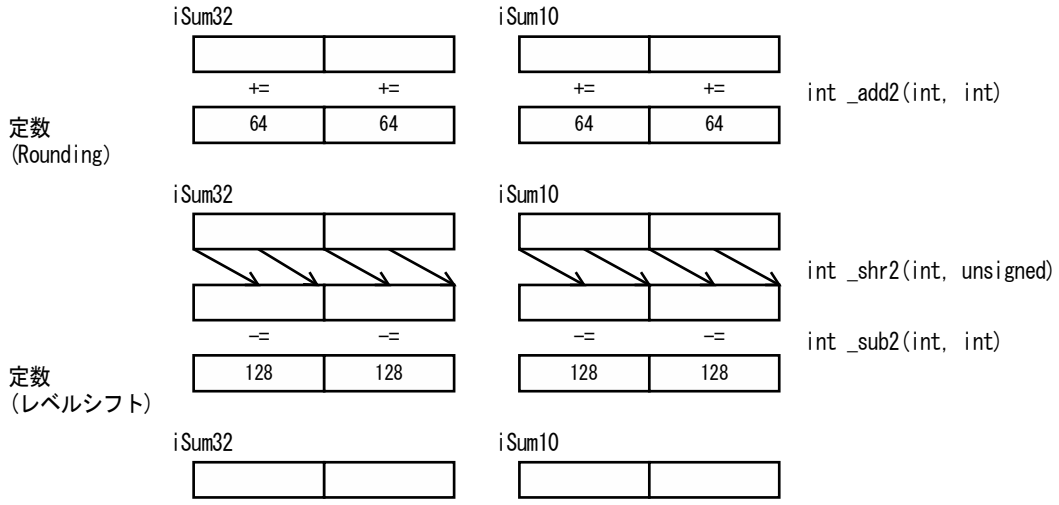


③ 4 データ(G)と 4 係数の乗算、及び Sum への加算



④ 4 データ(B)と 4 係数の乗算、及び Sum への加算
③と同様

⑤ Rounding 及び Sum の右 7 シフト及びレベルシフト



⑥ データをストア

⑦ Cb について、同様に②～⑥の処理を行う。但し、-128 のレベルシフト処理は必要なし。
⑧ Cr について、同様に②～⑥の処理を行う。但し、-128 のレベルシフト処理は必要なし。

②～⑥のプログラムは次になります。

```
// sY  = (( 38*sR + 75*sG + 15*sB + 64 ) >> 7) - 128;
 dTemp = _mpysu4( iCoefYR, nDataR );
 iSum32 = _hi(dTemp);
 iSum10 = _lo(dTemp);
 dTemp = _mpysu4( iCoefYG, nDataG );
 iSum32 = _add2( iSum32, _hi(dTemp) );
 iSum10 = _add2( iSum10, _lo(dTemp) );
 dTemp = _mpysu4( iCoefYB, nDataB );
 iSum32 = _add2( iSum32, _hi(dTemp) );
 iSum10 = _add2( iSum10, _lo(dTemp) );
 iSum32 = _add2( iSum32, iCoefRound );          // Rounding
 iSum10 = _add2( iSum10, iCoefRound );          // Rounding
 iSum32 = _shr2( iSum32, 7 );
 iSum10 = _shr2( iSum10, 7 );
 iSum32 = _sub2( iSum32, iCoef128 );            // Level shift
 iSum10 = _sub2( iSum10, iCoef128 );            // Level shift
 *piYBuffer++ = iSum32;
 *piYBuffer++ = iSum10;
```

６．２．　　8 ビットデータの 4 並列処理（２）

次の例は、乗算を伴わない場合です。

[例] 多値ディザ
ディザ処理の代表的な方式として次があります。

- 入力とスライスレベルを比較する。
- 入力とスライスレベルの比較結果を格納したテーブルを使用する方式。特に、入力データと X 方向の画素位置、Y 方向の画素位置を入力とする方式はテーブルが大きくなりますが高速です。
- 5 値、9 値など (2^N+1)値の場合は、入力の上位から多値出力の初期値を得て、入力の下位から初期値＋１を判定する事ができます。入力とスライスレベルとの比較が１回のため、比較的高速です。

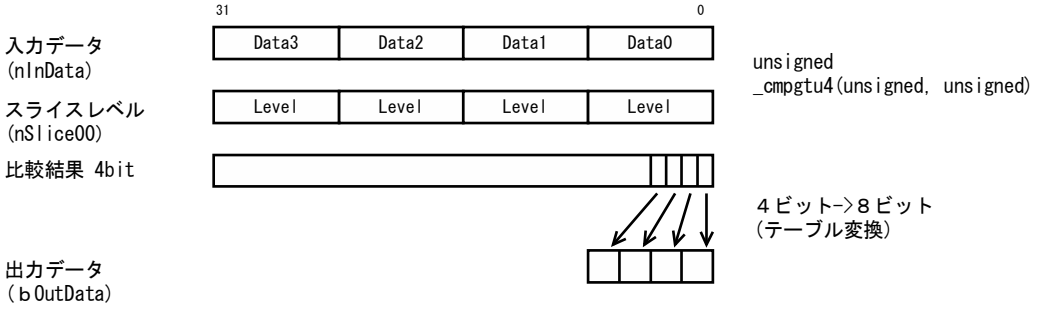
入力とスライスレベルを比較する方式で、4 値出力の場合は次の処理になります。

```
 bOutData = 0;
 if( bInData > bSlice0[(sXNum+0)%8] ) bOutData++;
 if( bInData > bSlice1[(sXNum+0)%8] ) bOutData++;
 if( bInData > bSlice2[(sXNum+0)%8] ) bOutData++;
```

SIMD 命令を適用した処理は次のようになります。

① 4 データをロード

② 各データとスライスを比較し、比較結果 4 ビットを 8 ビット(2 ビット/画素)に変換する。



③ ②と同様に各データと 2 番目のスライスを比較し、2 ビット/画素フォーマットに変換し、出力データに加算する。

④ ②と同様に各データと 3 番目のスライスを比較し、2 ビット/画素フォーマットに変換し、出力データに加算する。

⑤ データをストア

ループ内で 8 画素を処理する場合のプログラムは次になります。

```
  BYTE b4To8[16] = {
       0x00, 0x01, 0x04, 0x05, 0x10, 0x11, 0x14, 0x15,
       0x40, 0x41, 0x44, 0x45, 0x50, 0x51, 0x54, 0x55
 };

  for ( i = 0; i < sNumPixels/8; i++ ) {
       nInData = *pnInData++;
       nTemp = _cmpgtu4( nInData, nSlice00 );
       nOutData  = b4To8[nTemp];
       nTemp = _cmpgtu4( nInData, nSlice01 );
       nOutData += b4To8[nTemp];
       nTemp = _cmpgtu4( nInData, nSlice02 );
       nOutData += b4To8[nTemp];
       *pbOut++ = (BYTE)nOutData;

       nInData = *pnInData++;
       nTemp = _cmpgtu4( nInData, nSlice10 );
       nOutData  = b4To8[nTemp];
       nTemp = _cmpgtu4( nInData, nSlice11 );
       nOutData += b4To8[nTemp];
       nTemp = _cmpgtu4( nInData, nSlice12 );
       nOutData += b4To8[nTemp];
       *pbOut++ = (BYTE)nOutData;
   }
```

６．３．　　16 ビットデータの２並列処理

次の例は、16 ビットデータと 16 ビットの係数で乗算を行う場合です。

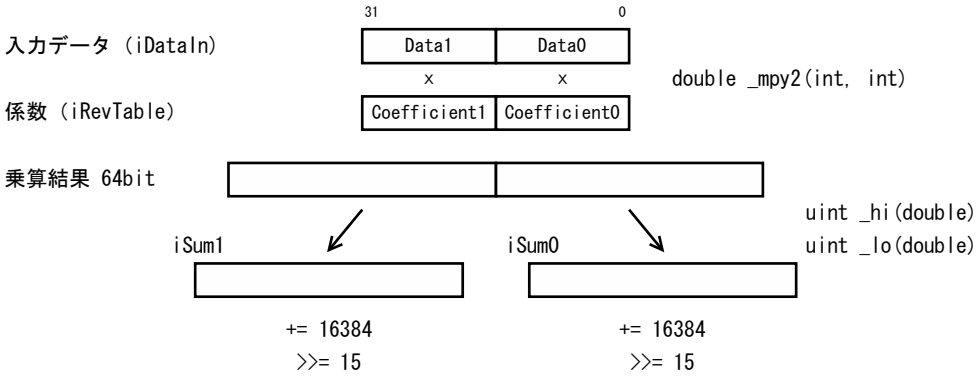
[例] JPEG 量子化処理

除算を逆数の乗算に置き換えて、次の処理になります。

```
sData = *psDataIn++;
sRevTable = *psRevQT++;                    // = 32768.0/sQT;
iDataOut = sRevTable*sData + 16384;
*psDataOut++ = iDataOut >> 15;
```

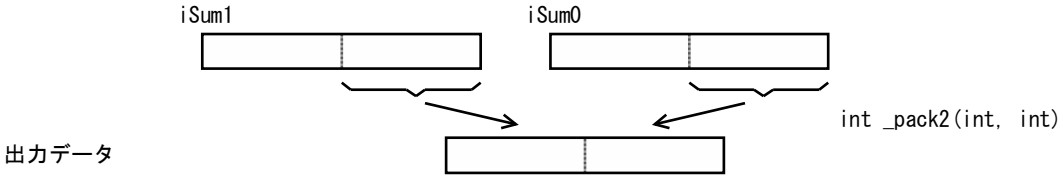
SIMD 命令を適用した処理は次のようになります。

① 2 データをロード



② 2 データと 2 係数の乗算、Rounding、右 15 シフトを行う。

③ 2 データをパックする。



④ データをストア

ループ内で 2 画素を処理する場合のプログラムは次になります。

```
for ( i = 0; i < 64/2; i++ ) {
    iDataIn = *piDataIn++;
    iRevTable = *piRevQT++;
    dTemp = _mpy2( iDataIn, iRevTable );
    iSum1 = _hi(dTemp);
    iSum0 = _lo(dTemp);
    iSum1 += 16384;
    iSum0 += 16384;
    iSum1 >>= 15;
    iSum0 >>= 15;
    iSum0 = _pack2( iSum1, iSum0 );
    *piDataOut++ = iSum0;
}
```

７．　参考資料

C6000 に関する資料の中で、次の資料は効率的なプログラムに関する解説書となっています。

SPRU198　　TMS320C6000 Programmer's Guide

Intrinsics については、TMS320C6000 Optimizing Compiler User’s Guide 及び TMS320C6000 CPU and Instruction Set Reference Guide を参照して下さい。