Using Streaming SIMD Extensions 2 (SSE2) to Implement an Inverse Discrete Cosine Transform

Version 2.0

7/00

Order Number: 248670-001

Table of Contents

Revision History

Revision	Revision History	Date
2.0	Pentium® 4 processor update	7/00
1.0	Original publication of document	9/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. Pennebaker and Mitchell, *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1993, pp. 29 – 64.

2. Rao and Yip, *Discrete Cosine Transform Algorithms, Advantages, Applications*, Academic Press, Inc., Boston, 1990, Appendix A.2

3. *A Fast Precise Implementation of 8x8 Discrete Cosine Transform Using the Streaming SIMD Extensions and MMX™ Instructions*, Intel Application Note, AP-922, Copyright 1999

4. IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform, IEEE Std. 1180-1990

1 Introduction

The Streaming SIMD Extensions 2 (SSE2) instructions introduces new Single Instruction Multiple Data (SIMD) double-precision floating-point instructions and new SIMD integer instructions into the IA-32 Intel® architecture. The double-precision SIMD instructions extend functionality in a manner analogous to the single-precision instructions introduced with the Streaming SIMD Extensions (SSE) instructions. The 128-bit SIMD integer extensions are a full superset of the 64-bit integer SIMD instructions, with additional instructions to support more integer data types, conversion between integer and floating-point data types, and efficient operations between the caches and system memory. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, spatial (3D) audio, video encoding/decoding, encryption, and scientific application. The 128-bit integer SIMD extensions in SSE2 technology can process data 128 bits at a time using the XMM registers. This enables the implementation of important algorithms, such as the Inverse Discrete Cosine Transform (IDCT) algorithm, to be improved further than previous implementation using SSE instructions. This application note contains both the code and a description of how the SSE2 instructions can be used to implement an IDCT.

2 Discrete Cosine Transforms

This section describes how the discrete cosine transform (DCT) converts the spatial data of an image into the frequency domain. Section 2.1 describes how the DCT is used for image compression. Section 2.2 describes the IDCT implementation used in this application note. The following discussion is borrowed from Pennebaker and Mitchell [1], pp. 29 – 64.

The one-dimensional DCT transforms eight pixel values of an image into eight coefficients. The coefficients represent amplitudes of eight reference cosine waves. These reference cosine waves are orthogonal waveforms that are also known as cosine basis functions (see Figure 1). Each cosine basis function has a different spatial frequency. The coefficient that scales the constant basis function shown in Fig. 1(0) is known as the Direct Current (DC) coefficient. The other seven coefficients are called the Alternating Current (AC) coefficients.

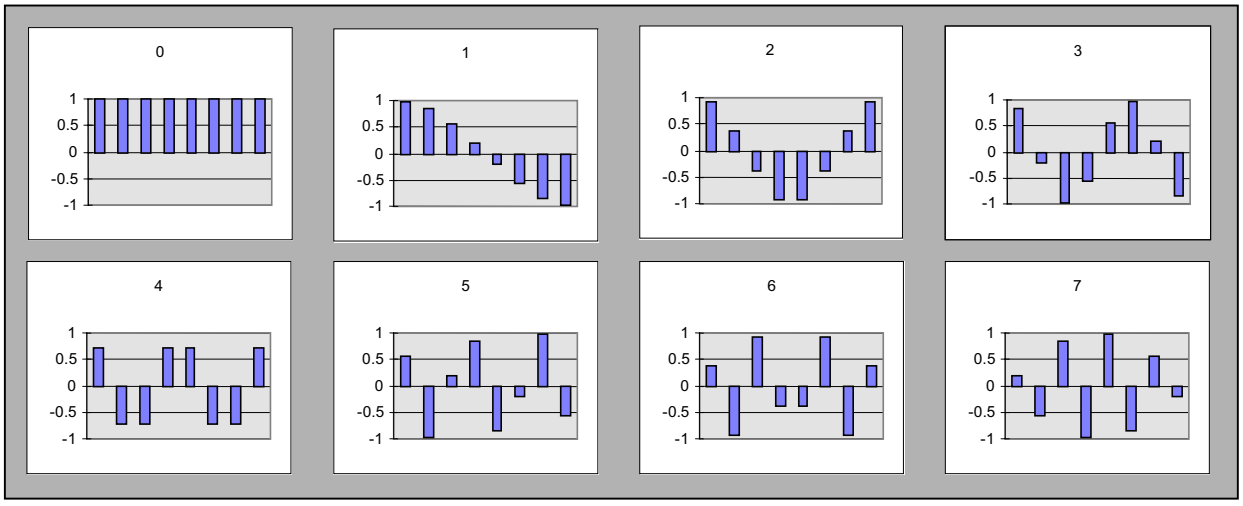

Figure 1: The Eight Cosine Basis Functions used in a 1D DCT

The two-dimensional DCT operates on 64 pixel values or an 8x8 block of pixels to generate 64 coefficients. The 2D DCT uses 64 basis functions that are created by multiplying 8 horizontal cosine basis functions with 8 vertical cosine basis functions. Figure 2 shows the 64 basis functions. The coefficient for the constant basis function shown in the upper left corner is known as the DC coefficient. The other 63 coefficients are known as the AC coefficients.

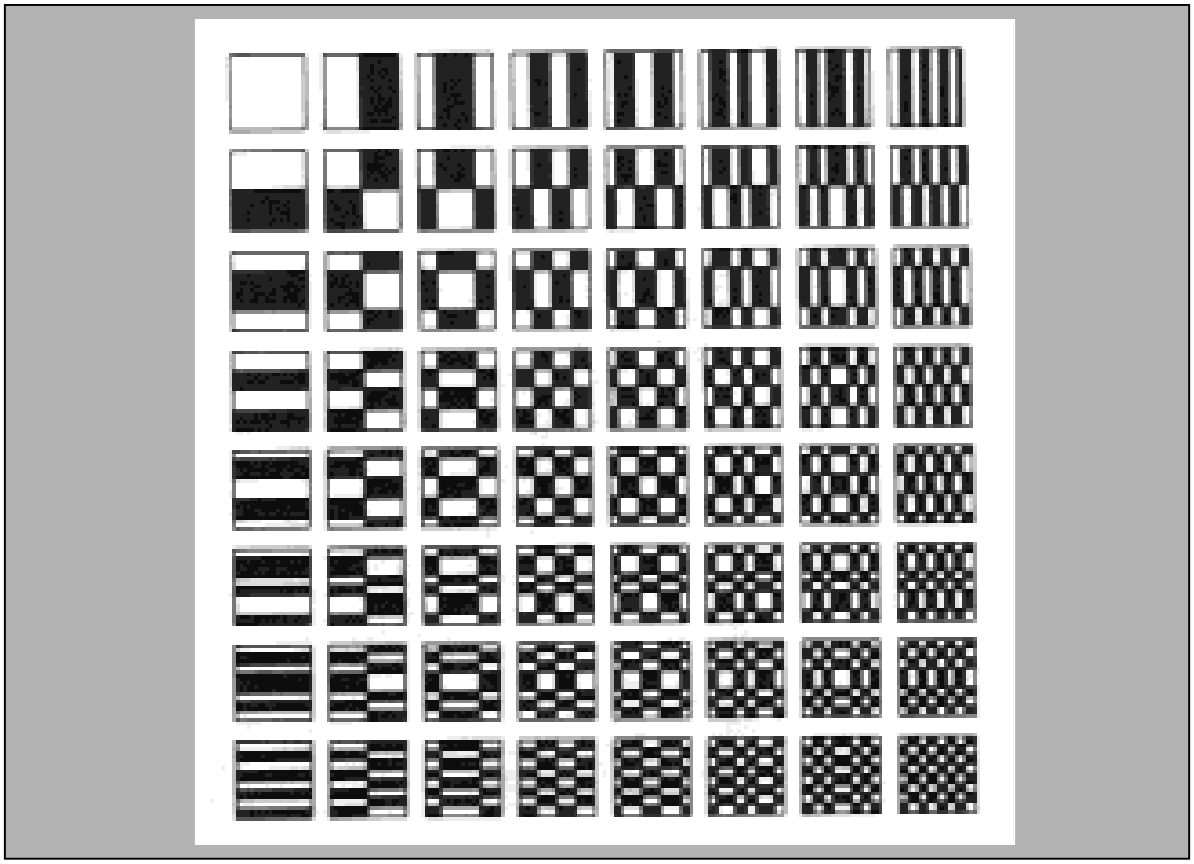

Figure 2: The 64 cosine basis functions for a 2D DCT. Gray represents zero, white represents positive amplitudes, and black represents negative amplitudes.

As explained in the preceding paragraph, the DCT is used to transform the pixel values of an image into coefficients (amplitudes of the cosine basis functions). The Inverse Discrete Cosine Transform (IDCT) converts the generated coefficients back to the original pixel values. The equations used in both the DCT and IDCT are as follows:

1D DCT:

$$F(u) = \frac{1}{2} C(u) \left[\sum_{x=0}^{7} f(x) * \cos \frac{(2x+1)u\pi}{16} \right] \quad (1)$$

2D DCT:

$$F(v,u) = \frac{1}{4} C(v) C(u) \left[\sum_{y=0}^{7} \sum_{x=0}^{7} f(y,x) * \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \right] \quad (2)$$

1D IDCT:

$$f(x) = \sum_{u=0}^{7} \frac{C(u)}{2} F(u) \cos \frac{(2x+1)u\pi}{16} \quad (3)$$

2D IDCT:

$$f(y,x) = \sum_{v=0}^{7} \frac{C(v)}{2} \sum_{u=0}^{7} \frac{C(u)}{2} F(v,u) \left[\cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \right] \quad (4)$$

where :

C(u) = $1/\sqrt{2}$ for u = 0, C(v) = $1/\sqrt{2}$ for v = 0,

C(u) = 1, for u > 0, C(v) = 1, for v >0,
f(x) = 1D sample value, f(y,x) = 2D sample value
F(u) = 1D DCT coefficient, F(v,u) = 2D DCT coefficient

The preceding equations show that the IDCT and DCT equations have the same number of operations. Using equation (2), a 2D DCT involves 64 multiplications and 63 additions per coefficient and transforming an 8x8 block involves 4096 multiplications and 4032 additions. To reduce the number of operations, you can substitute the 2D DCT with 16 1D DCTs (8 1D DCTs for the 8 rows and 8 1D DCTs for the 8 columns). Using equation (1), the 1D DCT requires 64 multiplications and 56 additions to generate the 8 coefficients. This means, transforming an 8x8 block involves 1024 multiplications and 896 additions. This number of operations is the upper bound. There are many other implementations of the DCT and IDCT that further reduce the number of operations. See Pennebaker-Mitchell, 1993 [1] and Rao-Yip, 1990 [2], for a discussion of other DCT and IDCT implementations. Section 2.2 discusses the implementation used in this application note.

2.1 Applications for the DCT/IDCT

The DCT can be used as part of an MPEG or JPEG encoder. As previously discussed, the DCT transforms the spatial data of an image into coefficients in the frequency domain. These coefficients are independent of each other. This means each coefficient can be treated separately without affecting the other coefficients. This independence is important because the human visual system is very dependent on spatial frequency and the eye perceives low frequency changes more readily than high frequency changes. Therefore, setting the coefficients associated with the high frequency basis functions to zero may be imperceptible to the human eye. To achieve good compression, encoders typically set the coefficients associated with the high frequency basis functions to zero.

Figure 3(a) describes a possible implementation of a variable length JPEG encoder. This implementation describes how the DCT can be used to achieve image compression. The DCT by itself does not compress the image. Instead, the DCT converts the image into the frequency domain to facilitate compression. In the encoder implementation described in Figure 3(a), compression is achieved by the Run Length Encoding and Huffman Encoding step. For more information on JPEG compression, see Pennebaker-Mitchell, 1993 [1].

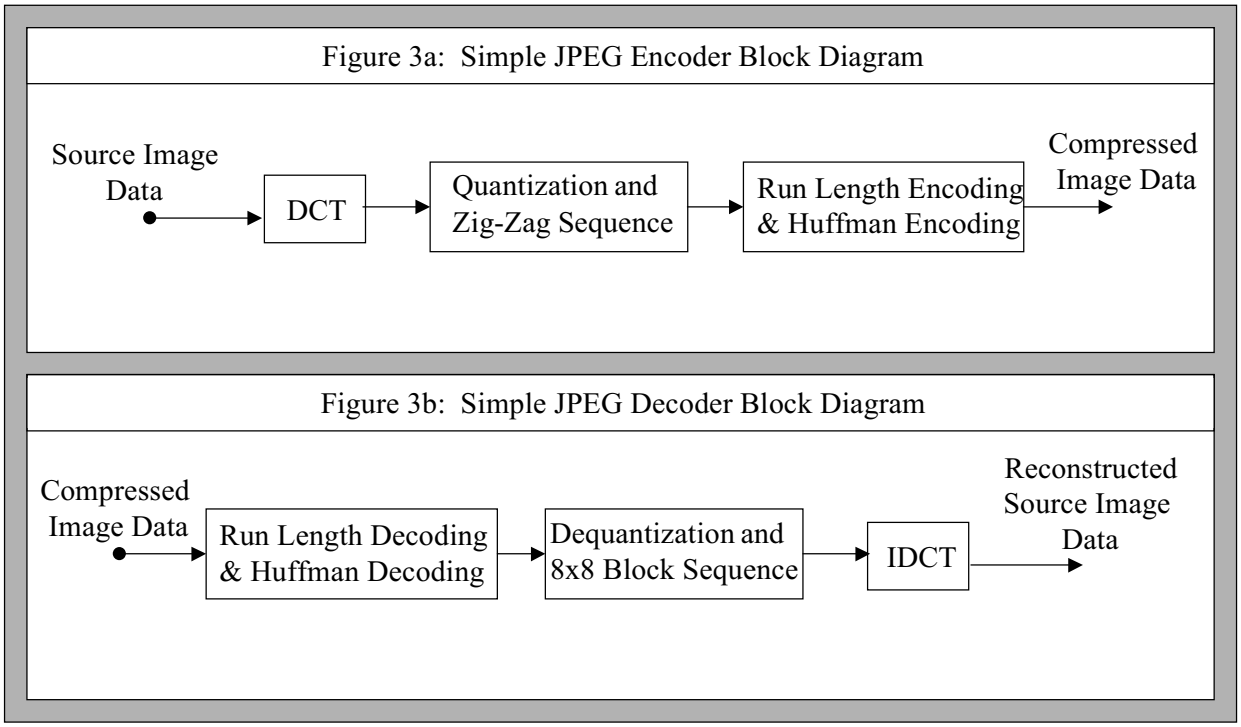

Figure 3: An implementation of a JPEG Encoder and Decoder

Just as the DCT is used for image compression in an encoder, the IDCT is used for image decompression in the decoder (see Figure 3(b)). Although algorithms vary, the DCT/IDCT steps generally take about 20% of the time in an encoder or decoder.

2.2 Implementing the IDCT Algorithm

As discussed earlier in this application note, the upper bound is 1024 multiplications and 896 additions. Many implementations have been created to reduce the number of operations. This app note uses the IDCT implementation that is presented in a previous Intel app note, *A Fast Precise Implementation of 8x8 Discrete Cosine Transform Using the Streaming SIMD Extensions and MMX™ Instructions,* AP-922 [3]. The reader is referred to this application note for a thorough discussion on this IDCT implementation. This implementation needs 320 multiplications, 464 additions, and 128 shifts. This implementation of the 2D IDCT was chosen for the following reasons:

1. The ability to use SIMD in both the row and column transformations. Using the SSE2 instructions, the number of operations needed in the 2D IDCT was reduced to 8 SIMD multiplications, 78 SIMD additions, 32 PMADD instructions, and 88 SIMD shift/shuffle instructions (where the SIMD width is eight shorts or eight 2-byte data values).
2. This implementation does not require a transpose before the columns are processed. Other SIMD implementations require a transpose, but this implementation uses two different 1D IDCT implementations to avoid the transpose.
3. No post-scaling/pre-scaling operations are done for this IDCT implementation. That is, this implementation did not distribute some of the multiplication operations to other steps in the encoder or decoder.
4. This implementation satisfies the precision requirements of the IEEE standard 1180-1900.

3 Performance

The IDCT implementation described in this application note was previously optimized for the Intel Pentium® III processor by using SSE instructions, see AP-922. The next section discusses how the SSE2 instructions are used to further optimize this implementation.

3.1 Gains/Improvements

The SSE2 instructions were used to optimize the 2D IDCT implementation. These instructions provided the following optimizations:

- **Larger SIMD width.** The integer SSE2 instructions use the 128-bit XMM registers instead of the 64-bit MMX technology registers. The increased SIMD width decreased register pressure and doubled the amount of data processed per instruction.
- **Enabled unrolling of the row 1D IDCT.** The decreased register pressure allowed the row 1D IDCT to be unrolled to operate on two rows at a time. The unrolling was done to execute more instructions in parallel. Specifically, the unrolling allowed the `pmaddwd` instructions to be processed sooner and in parallel with other instructions.
- **Decreased register pressure.** The decreased register pressure eliminated eight instructions in the assembly language (ASM) implementation. The ASM implementation removed two stores and two loads by keeping the values in the registers. Also, since the unrolling of the row IDCT operated on two rows simultaneously, pairing the rows properly resulted in eliminating four load effective address (`lea`) instructions. The eight eliminated instructions are commented out and labeled in the ASM implementation.

3.2 Considerations

The ASM implementation is slightly faster than the class library (IVEC) implementation. Both implementations use SSE2 instructions; the difference is that the IVEC implementation uses the Intel® C++ Class Libraries for SIMD Operations, which are a C++ wrapper to the Intel® C\C++ Compiler intrinsics.

The intrinsics are instructions that use a C-function call syntax and usually have a one-to-one mapping to the SSE2 instructions. The advantage of using the intrinsics or C++ SIMD classes is in being able to use the SSE2 instructions without having to resort to tedious assembly language programming. The disadvantage is that certain optimizations can only be achieved using assembly language. In this case, the ASM implementation removed two stores and two loads by keeping the values in the registers. The IVEC implementation does not provide this optimization because the compiler controls when or how the registers are spilled. For this reason, this ASM implementation is slightly faster than the IVEC implementation.

However, the reader should be aware that inline ASM might turn off some inter-procedural and intra-procedural compiler optimizations. Thus, for most cases, the C++ SIMD classes or intrinsics provide better application speedups than inline ASM. For more information on the C++ Class Libraries and intrinsics, please see the Intel C\C++ Class Libraries for SIMD Operations: With Support for the SSE2 instructions order number 749100-001 and the Intel C/C++ Compiler Intrinsics Reference Manual order number 748639-001.

4 Conclusion

The integer SSE2 instructions provided a speedup to a 2D IDCT implementation optimized with the SSE instructions. Because the SSE2 instructions use the XMM registers, the SSE2 instructions provided a larger SIMD width. The increased SIMD width reduced register pressure, reduced loads and stores, and enabled the row 1D IDCT to be unrolled. The unrolling resulted in better utilization of the Pentium 4 processor hardware resources and executed more instructions in parallel.

5 Baseline Coding Example

The code below uses the SSE instructions to perform a 2D IDCT. The following provides a summary of this 2-D IDCT implementation:

1. Perform a 1-D inverse DCT on each of the eight rows. A macro called `DCT_8_INV_ROW` performs the 1-D inverse DCT on one row. The macro expects the eight values of the row to be previously loaded into the `mm0` and `mm1` MMX technology registers. The macro also expects the `esi` register to point to the corresponding constant multiplier table. The macro needs to be called eight times to complete the 1-D inverse DCT on each of the eight rows. A description of the `DCT_8_INV_ROW` macro is provided in the comments belowin the example code.
2. Perform a 1-D inverse DCT on each of the eight columns. A macro called `DCT_8_INV_COL_4` performs the 1-D inverse DCT on four columns. The macro expects four values of the sixth row to be placed in the `mm0` MMX technology register. The macro also expects the `edx` register to point to the four columns to be processed. This macro needs to be called twice to complete the 1-D inverse DCT on each of the eight columns. A description of the `DCT_8_INV_COL_4` macro is provided in the comments example code.

```
#include "idct_kernel.h"

#define BITS_INV_ACC    4                               // 4 or 5 for IEEE
#define SHIFT_INV_ROW   16 - BITS_INV_ACC
#define SHIFT_INV_COL   1 + BITS_INV_ACC
const short RND_INV_ROW   = 1024 * (6 - BITS_INV_ACC);        //1 << (SHIFT_INV_ROW-1)
const short RND_INV_COL   = 16 * (BITS_INV_ACC - 3);          // 1 << (SHIFT_INV_COL-1)
const short RND_INV_CORR  = RND_INV_COL - 1;                  // correction -1.0 and round

static short one_corr[4] = {1,            1,            1,            1};
static short round_inv_row[4] = {RND_INV_ROW, 0, RND_INV_ROW, 0};
static short round_inv_col[4] = {RND_INV_COL,  RND_INV_COL,  RND_INV_COL,  RND_INV_COL};
static short round_inv_corr[4] = {RND_INV_CORR, RND_INV_CORR, RND_INV_CORR, RND_INV_CORR};

static short  tg_1_16[4] = {13036,  13036,  13036,  13036};       //  tg * (2<<16) + 0.5
static short  tg_2_16[4] = {27146,  27146,  27146,  27146};       //  tg * (2<<16) + 0.5
static short  tg_3_16[4] = {-21746, -21746, -21746, -21746};      //  tg * (2<<16) + 0.5
static short  cos_4_16[4] = {-19195, -19195, -19195, -19195};     //  cos * (2<<16) + 0.5


//--------------------------------------------------------------------------

// Table for rows 0,4 - constants are multiplied on cos_4_16

static short tab_i_04[] = {16384,21407,16384,8867,        //movq ->      w05 w04 w01 w00
              16384,   8867, -16384,  -21407,     //                   w07 w06 w03 w02
              16384,   -8867,  16384, -21407,     //                   w13 w12 w09 w08
```

```
                -16384,  21407,  16384,  -8867,     //                    w15 w14 w11 w10
                22725,   19266,  19266,  -4520,     //                    w21 w20 w17 w16
                12873,   4520, -22725, -12873,      //                    w23 w22 w19 w18
                12873,   -22725,  4520, -12873,     //                    w29 w28 w25 w24
                4520, 19266,  19266, -22725};       //                    w31 w30 w27 w26

// Table for rows 1,7 - constants are multiplied on cos_1_16

static short tab_i_17[] = {22725,  29692,  22725,  12299, //movq ->      w05 w04 w01 w00
                22725,  12299, -22725, -29692,      //                    w07 w06 w03 w02
                22725,   -12299, 22725, -29692,     //                    w13 w12 w09 w08
                -22725,  29692,  22725, -12299,     //                    w15 w14 w11 w10
                31521,  26722,  26722,  -6270,      //                    w21 w20 w17 w16
                17855,  6270,  -31521, -17855,      //                    w23 w22 w19 w18
                17855,   -31521, 6270,  -17855,     //                    w29 w28 w25 w24
                6270, 26722,  26722, -31521};       //                    w31 w30 w27 w26

// Table for rows 2,6 - constants are multiplied on cos_2_16

static short tab_i_26[] = {21407,  27969,  21407,  11585, /movq ->       w05 w04 w01 w00
                21407,  11585, -21407, -27969,      //                    w07 w06 w03 w02
                21407,   -11585, 21407, -27969,     //                    w13 w12 w09 w08
                -21407, 27969,  21407,  -11585,     //                    w15 w14 w11 w10
                29692,  25172,  25172,  -5906,      //                    w21 w20 w17 w16
                16819,  5906,  -29692, -16819,      //                    w23 w22 w19 w18
                16819,   -29692, 5906,  -16819,     //                    w29 w28 w25 w24
                5906, 25172,  25172, -29692};       //                    w31 w30 w27 w26

// Table for rows 3,5 - constants are multiplied on cos_3_16

static short tab_i_35[] = {19266,  25172,  19266,  10426, //movq ->      w05 w04 w01 w00
                19266,  10426, -19266, -25172,      //                    w07 w06 w03 w02
                19266,   -10426, 19266, -25172,     //                    w13 w12 w09 w08
                -19266, 25172,  19266,  -10426,     //                    w15 w14 w11 w10
                26722,  22654,  22654,  -5315,      //                    w21 w20 w17 w16
                15137,  5315,  -26722, -15137,      //                    w23 w22 w19 w18
                15137,   -26722, 5315,  -15137,     //                    w29 w28 w25 w24
                5315, 22654,  22654, -26722};       //                    w31 w30 w27 w26


//---------------------------------------------------------------------------

/*
```

```
;============================================================================
;============================================================================
;============================================================================
;
; Inverse DCT
;
;----------------------------------------------------------------------------
;
; This implementation calculates iDCT-2D by a row-column method.
; On the first stage the iDCT-1D is calculated for each row with use
; direct algorithm, on the second stage the calculation is executed
; at once for four columns with use of scaled iDCT-1D algorithm.
; Base R&Y algorithm for iDCT-1D is modified for second stage.
;
;============================================================================

;----------------------------------------------------------------------------
;
; The first stage - inverse DCTs of rows
;
;----------------------------------------------------------------------------
; The 8-point inverse DCT direct algorithm
;----------------------------------------------------------------------------
;
; static const short w[32] = {
;    FIX(cos_4_16),  FIX(cos_2_16),  FIX(cos_4_16),  FIX(cos_6_16),
;    FIX(cos_4_16),  FIX(cos_6_16), -FIX(cos_4_16), -FIX(cos_2_16),
;    FIX(cos_4_16), -FIX(cos_6_16), -FIX(cos_4_16),  FIX(cos_2_16),
;    FIX(cos_4_16), -FIX(cos_2_16),  FIX(cos_4_16), -FIX(cos_6_16),
;    FIX(cos_1_16),  FIX(cos_3_16),  FIX(cos_5_16),  FIX(cos_7_16),
;    FIX(cos_3_16), -FIX(cos_7_16), -FIX(cos_1_16), -FIX(cos_5_16),
;    FIX(cos_5_16), -FIX(cos_1_16),  FIX(cos_7_16),  FIX(cos_3_16),
;    FIX(cos_7_16), -FIX(cos_5_16),  FIX(cos_3_16), -FIX(cos_1_16) };
;
; #define DCT_8_INV_ROW(x, y)
; {
;    int a0, a1, a2, a3, b0, b1, b2, b3;
;
;    a0   = x[0] * w[ 0] + x[2] * w[ 1] + x[4] * w[ 2] + x[6] * w[ 3];
;    a1   = x[0] * w[ 4] + x[2] * w[ 5] + x[4] * w[ 6] + x[6] * w[ 7];
;    a2   = x[0] * w[ 8] + x[2] * w[ 9] + x[4] * w[10] + x[6] * w[11];
;    a3   = x[0] * w[12] + x[2] * w[13] + x[4] * w[14] + x[6] * w[15];
;    b0   = x[1] * w[16] + x[3] * w[17] + x[5] * w[18] + x[7] * w[19];
```

```
;    b1   = x[1] * w[20] + x[3] * w[21] + x[5] * w[22] + x[7] * w[23];
;    b2   = x[1] * w[24] + x[3] * w[25] + x[5] * w[26] + x[7] * w[27];
;    b3   = x[1] * w[28] + x[3] * w[29] + x[5] * w[30] + x[7] * w[31];
;
;    y[0] = SHIFT_ROUND ( a0 + b0 );
;    y[1] = SHIFT_ROUND ( a1 + b1 );
;    y[2] = SHIFT_ROUND ( a2 + b2 );
;    y[3] = SHIFT_ROUND ( a3 + b3 );
;    y[4] = SHIFT_ROUND ( a3 - b3 );
;    y[5] = SHIFT_ROUND ( a2 - b2 );
;    y[6] = SHIFT_ROUND ( a1 - b1 );
;    y[7] = SHIFT_ROUND ( a0 - b0 );
; }
;
;-------------------------------------------------------------------------
;
; In this implementation the outputs of the iDCT-1D are multiplied
;    for rows 0,4 - on cos_4_16,
;    for rows 1,7 - on cos_1_16,
;    for rows 2,6 - on cos_2_16,
;    for rows 3,5 - on cos_3_16
; and are shifted to the left for rise of accuracy
;
; For used constants
;    FIX(float_const) = (short) (float_const * (1<<15) + 0.5)
;
;-------------------------------------------------------------------------

;-------------------------------------------------------------------------
;
; The second stage - inverse DCTs of columns
;
; The inputs are multiplied
;    for rows 0,4 - on cos_4_16,
;    for rows 1,7 - on cos_1_16,
;    for rows 2,6 - on cos_2_16,
;    for rows 3,5 - on cos_3_16
; and are shifted to the left for rise of accuracy
;
;-------------------------------------------------------------------------
;
; The 8-point scaled inverse DCT algorithm (26a8m)
;
```

```
;-----------------------------------------------------------------------------
;
; #define DCT_8_INV_COL(x, y)
; {
;    short t0, t1, t2, t3, t4, t5, t6, t7;
;    short tp03, tm03, tp12, tm12, tp65, tm65;
;    short tp465, tm465, tp765, tm765;
;
;    tp765 = x[1]           + x[7] * tg_1_16;
;    tp465 = x[1] * tg_1_16 - x[7];
;    tm765 = x[5] * tg_3_16 + x[3];
;    tm465 = x[5]           - x[3] * tg_3_16;
;
;    t7    = tp765 + tm765;
;    tp65  = tp765 - tm765;
;    t4    = tp465 + tm465;
;    tm65  = tp465 - tm465;
;
;    t6    = ( tp65 + tm65 ) * cos_4_16;
;    t5    = ( tp65 - tm65 ) * cos_4_16;
;
;    tp03  = x[0] + x[4];
;    tp12  = x[0] - x[4];
;
;    tm03  = x[2]           + x[6] * tg_2_16;
;    tm12  = x[2] * tg_2_16 - x[6];
;
;    t0    = tp03 + tm03;
;    t3    = tp03 - tm03;
;    t1    = tp12 + tm12;
;    t2    = tp12 - tm12;
;
;    y[0]  = SHIFT_ROUND ( t0 + t7 );
;    y[7]  = SHIFT_ROUND ( t0 - t7 );
;    y[1]  = SHIFT_ROUND ( t1 + t6 );
;    y[6]  = SHIFT_ROUND ( t1 - t6 );
;    y[2]  = SHIFT_ROUND ( t2 + t5 );
;    y[5]  = SHIFT_ROUND ( t2 - t5 );
;    y[3]  = SHIFT_ROUND ( t3 + t4 );
;    y[4]  = SHIFT_ROUND ( t3 - t4 );
; }
;
;-----------------------------------------------------------------------------
```

```
*/

#define DCT_8_INV_ROW   __asm{                                                  \
      __asm movq     mm2, mm0                /* 2         ; x3 x2 x1 x0*/       \
      __asm movq     mm3, qword ptr [esi]    /* 3         ; w05 w04 w01 w00*/   \
      __asm pshufw   mm0, mm0, 10001000b                 /* x2 x0 x2 x0*/       \
      __asm movq     mm4, qword ptr [esi+8]  /* 4         ; w07 w06 w03 w02*/   \
      __asm movq     mm5, mm1                /* 5         ; x7 x6 x5 x4*/       \
      __asm pmaddwd  mm3, mm0                     /* x2*w05+x0*w04 x2*w01+x0*w00*/ \
      __asm movq     mm6, qword ptr [esi+32] /* 6         ; w21 w20 w17 w16*/   \
      __asm pshufw   mm1, mm1, 10001000b                 /* x6 x4 x6 x4*/       \
      __asm pmaddwd  mm4, mm1                     /* x6*w07+x4*w06 x6*w03+x4*w02*/ \
      __asm movq     mm7, qword ptr [esi+40] /* 7         ; w23 w22 w19 w18*/   \
      __asm pshufw   mm2, mm2, 11011101b                 /* x3 x1 x3 x1*/       \
      __asm pmaddwd  mm6, mm2                     /* x3*w21+x1*w20 x3*w17+x1*w16*/ \
      __asm pshufw   mm5, mm5, 11011101b                 /* x7 x5 x7 x5*/       \
      __asm pmaddwd  mm7, mm5                     /* x7*w23+x5*w22 x7*w19+x5*w18*/ \
      __asm paddd    mm3, qword ptr round_inv_row         /* +rounder    */      \
      __asm pmaddwd  mm0, qword ptr [esi+16]      /* x2*w13+x0*w12 x2*w09+x0*w08*/ \
      __asm paddd    mm3, mm4                /* 4 ; a1=sum(even1) a0=sum(even0)*/ \
      __asm pmaddwd  mm1, qword ptr [esi+24]      /* x6*w15+x4*w14 x6*w11+x4*w10*/ \
      __asm movq     mm4, mm3                /* 4         ; a1 a0               */ \
      __asm pmaddwd  mm2, qword ptr [esi+48]      /* x3*w29+x1*w28 x3*w25+x1*w24*/ \
      __asm paddd    mm6, mm7                /* 7  ; b1=sum(odd1) b0=sum(odd0)*/ \
      __asm pmaddwd  mm5, qword ptr [esi+56]      /* x7*w31+x5*w30 x7*w27+x5*w26*/ \
      __asm paddd    mm3, mm6                         /* a1+b1 a0+b0*/          \
      __asm paddd    mm0, qword ptr round_inv_row         /* +rounder*/         \
      __asm psrad    mm3, SHIFT_INV_ROW               /* y1=a1+b1 y0=a0+b0*/    \
      __asm paddd    mm0, mm1                /* 1  ; a3=sum(even3) a2=sum(even2)*/ \
      __asm psubd    mm4, mm6                /* 6         ; a1-b1 a0-b0  */      \
      __asm movq     mm7, mm0                /* 7         ; a3 a2 */             \
      __asm paddd    mm2, mm5                /* 5  ; b3=sum(odd3) b2=sum(odd2)*/ \
      __asm paddd    mm0, mm2                   /* a3+b3 a2+b2*/               \
      __asm psrad    mm4, SHIFT_INV_ROW            /* y6=a1-b1 y7=a0-b0*/     \
      __asm psubd    mm7, mm2                /* 2         ; a3-b3 a2-b2*/       \
      __asm psrad    mm0, SHIFT_INV_ROW               /* y3=a3+b3 y2=a2+b2*/    \
      __asm psrad    mm7, SHIFT_INV_ROW               /* y4=a3-b3 y5=a2-b2*/    \
      __asm packssdw mm3, mm0                /* 0         ; y3 y2 y1 y0*/       \
      __asm packssdw mm7, mm4                /* 4         ; y6 y7 y4 y5*/       \
      __asm pshufw   mm7, mm7, 10110001b                 /* y7 y6 y5 y4 */      \
   }
```

```
#define DCT_8_INV_COL_4 __asm{                                                  \
      __asm movq     mm1, qword ptr tg_3_16  /* 1         ; tg_3_16 */          \
      __asm movq     mm2, mm0                /* 2         ; x5 */               \
      __asm movq     mm3, qword ptr [edx+3*16]  /* 3      ; x3 */               \
      __asm pmulhw   mm0, mm1                           /* x5*tg_3_16 */        \
      __asm movq     mm4, qword ptr [edx+7*16]  /* 4      ; x7 */               \
      __asm pmulhw   mm1, mm3                           /* x3*tg_3_16 */        \
      __asm movq     mm5, qword ptr tg_1_16  /* 5         ; tg_1_16 */          \
      __asm movq     mm6, mm4                /*; 6        ; x7 */               \
      __asm pmulhw   mm4, mm5                           /* x7*tg_1_16 */        \
      __asm paddsw   mm0, mm2                           /* x5*tg_3_16 */        \
      __asm pmulhw   mm5, [edx+1*16]                    /* x1*tg_1_16 */        \
      __asm paddsw   mm1, mm3                           /* x3*tg_3_16 */        \
      __asm movq     mm7, qword ptr [edx+6*16]  /* 7             ; x6 */        \
      __asm paddsw   mm0, mm3                /* 3         ; tm765 = x5*tg_3_16+x3 */ \
      __asm movq     mm3, qword ptr tg_2_16  /* 3         ; tg_2_16 */          \
      __asm psubsw   mm2, mm1                /* 1         ; tm465 = x5-x3*tg_3_16 */ \
      __asm pmulhw   mm7, mm3                           /* x6*tg_2_16 */        \
      __asm movq     mm1, mm0                /* 1         ; tm765 */            \
      __asm pmulhw   mm3, [edx+2*16]                    /* x2*tg_2_16 */        \
      __asm psubsw   mm5, mm6                /* 6         ; tp465 = x1*tg_1_16-x7 */ \
      __asm paddsw   mm4, [edx+1*16]                    /* tp765 = x1+x7*tg_1_16 */ \
      __asm paddsw   mm0, mm4                           /* t7   = tp765 + tm765 */ \
      __asm paddsw   mm0, qword ptr one_corr            /* correction t7 +1.0 */  \
      __asm psubsw   mm4, mm1                /* 1         ; tp65 = tp765 - tm765 */ \
      __asm paddsw   mm7, [edx+2*16]                    /* tm03 = x2+x6*tg_2_16 */ \
      __asm movq     mm6, mm5                /* 6         ; tp465 */            \
      __asm psubsw   mm3, [edx+6*16]                    /* tm12 = x2*tg_2_16-x6 */ \
      __asm psubsw   mm5, mm2                           /* tm65 = tp465 - tm465 */ \
      __asm paddsw   mm5, qword ptr one_corr            /* correction tm65 +1.0 */ \
      __asm paddsw   mm6, mm2                /* 2         ; t4   = tp465 + tm465 */ \
      __asm movq     [edx+7*16],  mm0        /* 0         ; save t7 in y7 (tmp) */ \
      __asm movq     mm1, mm4                /* 1         ; tp65 */             \
      __asm movq     mm2, qword ptr cos_4_16 /* 2         ; cos_4_16 */         \
      __asm paddsw   mm4, mm5                           /* tp65 + tm65 */       \
      __asm movq     mm0, qword ptr cos_4_16 /* 0         ; cos_4_16 */         \
      __asm pmulhw   mm2, mm4                           /* (tp65 + tm65)*cos_4_16 */ \
      __asm movq     [edx+3*16],  mm6        /* 6         ; save t4 in y3 (tmp) */ \
      __asm psubsw   mm1, mm5                /* 5         ; tp65 - tm65 */      \
      __asm movq     mm6, [edx]              /* 6         ; x0 */               \
      __asm pmulhw   mm0, mm1                           /* (tp65 - tm65)*cos_4_16 */ \
      __asm movq     mm5, [edx+4*16]         /* 5         ; x4 */               \
      __asm paddsw   mm4, mm2                /* 2  ; t6 = (tp65 + tm65)*cos_4_16 */ \
```

```
__asm por      mm4, qword ptr one_corr              /* correction t6 +0.5 */      \
__asm paddsw   mm5, mm6                             /* tp03 = x0 + x4 */          \
__asm psubsw   mm6, [edx+4*16]                      /* tp12 = x0 - x4 */          \
__asm paddsw   mm0, mm1               /* 1  ; t5 = (tp65 - tm65)*cos_4_16 */      \
__asm por      mm0, qword ptr one_corr              /* correction t5 +0.5 */      \
__asm movq     mm2, mm5               /* 2         ; tp03 */                     \
__asm paddsw   mm5, mm7                             /* t0 = tp03 + tm03 */        \
__asm movq     mm1, mm6               /* 1         ; tp12 */                     \
__asm paddsw   mm5, qword ptr round_inv_col         /* t0 + rounder */            \
__asm psubsw   mm2, mm7               /* 7         ; t3 = tp03 - tm03 */         \
__asm movq     mm7, [edx+7*16]                      /* t7 */                      \
__asm paddsw   mm6, mm3                             /* t1 = tp12 + tm12 */        \
__asm paddsw   mm6, qword ptr round_inv_col         /* t1 + rounder */            \
__asm paddsw   mm7, mm5                             /* t0 + t7 */                 \
__asm psraw    mm7, SHIFT_INV_COL                   /* y0 = t0 + t7 */            \
__asm psubsw   mm1, mm3               /* 3         ; t2 = tp12 - tm12 */         \
__asm paddsw   mm2, qword ptr round_inv_corr /* correction t3 -1.0 +rounder */    \
__asm movq     mm3, mm6               /* 3         ; t1 */                       \
__asm paddsw   mm1, qword ptr round_inv_corr /* correction t2 -1.0 +rounder */    \
__asm paddsw   mm6, mm4                             /*       * t1 + t6 */        \
__asm movq     [edx], mm7             /* 7         ; save y0 */                  \
__asm psraw    mm6, SHIFT_INV_COL                   /* y1 = t1 + t6 */            \
__asm movq     mm7, mm1               /* 7         ; t2 */                       \
__asm paddsw   mm1, mm0                             /* t2 + t5 */                 \
__asm movq     [edx+1*16], mm6        /* 6         ; save y1 */                  \
__asm psraw    mm1, SHIFT_INV_COL                   /* y2 = t2 + t5 */            \
__asm movq     mm6, [edx+3*16]        /* 6         ; t4 */                       \
__asm psubsw   mm7, mm0               /* 0         ; t2 - t5 */                  \
__asm paddsw   mm6, mm2                             /* t3 + t4 */                 \
__asm psubsw   mm2, [edx+3*16]                      /* t3 - t4 */                 \
__asm psraw    mm7, SHIFT_INV_COL                   /* y5 = t2 - t5 */            \
__asm movq     [edx+2*16],  mm1       /* 1         ; save y2 */                  \
__asm psraw    mm6, SHIFT_INV_COL                   /* y3 = t3 + t4 */            \
__asm psubsw   mm5, [edx+7*16]                      /* t0 - t7 */                 \
__asm psraw    mm2, SHIFT_INV_COL                   /* y4 = t3 - t4 */            \
__asm movq     [edx+3*16],  mm6       /* 6         ; save y3 */                  \
__asm psubsw   mm3, mm4               /* 4         ; t1 - t6 */                  \
__asm movq     [edx+4*16],  mm2       /* 2         ; save y4 */                  \
__asm psraw    mm3, SHIFT_INV_COL                   /* y6 = t1 - t6 */            \
__asm movq     [edx+5*16],  mm7       /* 7         ; save y5 */                  \
__asm psraw    mm5, SHIFT_INV_COL                   /* y7 = t0 - t7 */            \
__asm movq     [edx+6*16],  mm3       /* 3         ; save y6 */                  \
__asm movq     [edx+7*16], mm5        /* 5         ; save y7 */                  \
```

```
   }


int idct_P3ASM::TheCode()
{
      short* src = dctcoeffshort;
      short* dst = tst;
      __asm mov      ecx, src
      __asm mov      edx, dst
      __asm movq     mm0, [ecx]
      __asm movq     mm1, [ecx+8]
      __asm lea      esi, tab_i_04
      DCT_8_INV_ROW; //Row 1, tab_i_04
      __asm movq     mm0, [ecx+16]
      __asm movq     qword ptr [edx],  mm3      /* 3       /* save y3 y2 y1 y0    */
      __asm movq     mm1, [ecx+24]
      __asm movq     qword ptr [edx+8],   mm7   /* 7       /* save y7 y6 y5 y4    */
      __asm lea      esi, tab_i_17
      DCT_8_INV_ROW; //Row 2, tab_i_17
      __asm movq     mm0, [ecx+32]
      __asm movq     qword ptr [edx+16],  mm3   /* 3       /* save y3 y2 y1 y0    */
      __asm movq     mm1, [ecx+40]
      __asm movq     qword ptr [edx+24],  mm7   /* 7       /* save y7 y6 y5 y4    */
      __asm lea      esi, tab_i_26
      DCT_8_INV_ROW; //Row 3, tab_i_26
      __asm movq     mm0, [ecx+48]
      __asm movq     qword ptr [edx+32],  mm3   /* 3       /* save y3 y2 y1 y0    */
      __asm movq     mm1, [ecx+56]
      __asm movq     qword ptr [edx+40],  mm7   /* 7       /* save y7 y6 y5 y4    */
      __asm lea      esi, tab_i_35
      DCT_8_INV_ROW; //Row 4, tab_i_35
      __asm movq     mm0, [ecx+64]
      __asm movq     qword ptr [edx+48],  mm3   /* 3       /* save y3 y2 y1 y0    */
      __asm movq     mm1, [ecx+72]
      __asm movq     qword ptr [edx+56],  mm7   /* 7       /* save y7 y6 y5 y4    */
      __asm lea      esi, tab_i_04
      DCT_8_INV_ROW; //Row 5, tab_i_04
      __asm movq     mm0, [ecx+80]
      __asm movq     qword ptr [edx+64],  mm3   /* 3       /* save y3 y2 y1 y0    */
      __asm movq     mm1, [ecx+88]
      __asm movq     qword ptr [edx+72],  mm7   /* 7       /* save y7 y6 y5 y4
      __asm lea      esi, tab_i_35
      DCT_8_INV_ROW; //Row 6, tab_i_35
```

```
        __asm movq      mm0, [ecx+96]
        __asm movq      qword ptr [edx+80],  mm3   /* 3       /* save y3 y2 y1 y0
        __asm movq      mm1, [ecx+104]
        __asm movq      qword ptr [edx+88],  mm7   /* 7       /* save y7 y6 y5 y4    */
        __asm lea       esi, tab_i_26
        DCT_8_INV_ROW; //Row 7, tab_i_26
        __asm movq      mm0, [ecx+112]
        __asm movq      qword ptr [edx+96],  mm3   /* 3       /* save y3 y2 y1 y0    */
        __asm movq      mm1, [ecx+120]
        __asm movq      qword ptr [edx+104],mm7 /* 7         /* save y7 y6 y5 y4    */
        __asm lea       esi, tab_i_17
        DCT_8_INV_ROW; //Row 8, tab_i_17
        __asm movq      qword ptr [edx+112],mm3 /* 3         /* save y3 y2 y1 y0    */
        __asm movq      mm0, qword ptr [edx+80] /* 0         /* x5 */
        __asm movq      qword ptr [edx+120],mm7 /* 7         /* save y7 y6 y5 y4    */

        DCT_8_INV_COL_4
        __asm movq      mm0, qword ptr [edx+88] /* 0         /* x5 */
        __asm add       edx, 8
        DCT_8_INV_COL_4
        //__asm  emms
        return(EXIT_SUCCESS);
}
```

6 SSE2 Instructions Assembly Code Example

The code in this section and in section 7 uses the SSE2 instructions to perform a 2D IDCT. The following is a summary of this 2-D IDCT implementation:

1. Perform a 1-D inverse DCT on each of the eight rows. A macro called `DCT_8_INV_ROW` is used to perform the 1-D inverse DCT on two rows. The macro expects the rows to be previously loaded into the `xmm0` and `xmm4` registers. The macro also expects the esi and `ecx` registers to point to the corresponding constant multiplier table. The macro needs to be called four times to complete the 1-D inverse DCT on each of the eight rows. A description of the `DCT_8_INV_ROW` macro is provided in the example code that follows.
2. Perform a 1-D inverse DCT on each of the eight columns. A macro called `DCT_8_INV_COL_8` performs the 1-D inverse DCT on eight columns. The macro expects the values of the sixth row to be placed in the `xmm0` register. The macro also expects the `edx` register to point to the eight columns that will be processed. This macro needs to be called once to complete the 1-D inverse DCT on each of the eight columns. A description of the DCT_8_INV_COL_8 macro is provided in the example code that follows.

```
#include <dvec.h>
#include "idct_kernel.h"



#define BITS_INV_ACC    4                                // 4 or 5 for IEEE
#define SHIFT_INV_ROW   16 - BITS_INV_ACC
#define SHIFT_INV_COL   1 + BITS_INV_ACC
const short RND_INV_ROW   = 1024 * (6 - BITS_INV_ACC);       //1 << (SHIFT_INV_ROW-1)
const short RND_INV_COL   = 16 * (BITS_INV_ACC - 3);         // 1 << (SHIFT_INV_COL-1)
const short RND_INV_CORR  = RND_INV_COL - 1;                 // correction -1.0 and round



__declspec(align(16)) short M128_one_corr[8] = {1,1,       1,1,   1,     1,     1,
   1};
__declspec(align(16)) short M128_round_inv_row[8] = {RND_INV_ROW, 0, RND_INV_ROW, 0,
RND_INV_ROW, 0, RND_INV_ROW, 0};


__declspec(align(16)) short M128_round_inv_col[8] = {RND_INV_COL,  RND_INV_COL,
RND_INV_COL,  RND_INV_COL, RND_INV_COL,  RND_INV_COL,  RND_INV_COL,  RND_INV_COL};
__declspec(align(16)) short M128_round_inv_corr[8]= {RND_INV_CORR, RND_INV_CORR,
RND_INV_CORR, RND_INV_CORR, RND_INV_CORR, RND_INV_CORR, RND_INV_CORR, RND_INV_CORR};
__declspec(align(16)) short  M128_tg_1_16[8] = {13036,  13036,  13036,  13036,
   13036,  13036,  13036,  13036}; //  tg * (2<<16) + 0.5
__declspec(align(16)) short  M128_tg_2_16[8] = {27146,  27146,  27146,  27146,
   27146,  27146,  27146,  27146}; //  tg * (2<<16) + 0.5
__declspec(align(16)) short  M128_tg_3_16[8] = {-21746, -21746, -21746, -21746,
-21746, -21746, -21746, -21746};   //  tg * (2<<16) + 0.5
__declspec(align(16)) short  M128_cos_4_16[8] = {-19195, -19195, -19195, -19195,
```

```
-19195, -19195, -19195, -19195};//  cos * (2<<16) + 0.5


//---------------------------------------------------------------------------

// Table for rows 0,4 - constants are multiplied on cos_4_16


//movq ->  w13 w12 w09 w08 w05 w04 w01 w00
//      w15 w14 w11 w10 w07 w06 w03 w02
//      w29 w28 w25 w24 w21 w20 w17 w16
//      w31 w30 w27 w26 w23 w22 w19 w18
                                                  //movq ->    w05 w04 w01 w00
__declspec(align(16)) short M128_tab_i_04[] = {16384,      21407, 16384, 8867,
             16384,  -8867,  16384, -21407,  //                  w13 w12 w09 w08
             16384,  8867, -16384, -21407,   //                  w07 w06 w03 w02
             -16384, 21407,  16384,  -8867,  //                  w15 w14 w11 w10
             22725,  19266,  19266,  -4520,  //                  w21 w20 w17 w16
             12873,  -22725, 4520, -12873,   //                  w29 w28 w25 w24
             12873,  4520, -22725, -12873,   //                  w23 w22 w19 w18
             4520, 19266,  19266,  -22725};  //                  w31 w30 w27 w26

// Table for rows 1,7 - constants are multiplied on cos_1_16
                                                  //movq ->    w05 w04 w01 w00
__declspec(align(16)) short M128_tab_i_17[] = {22725,  29692,  22725,  12299,
             22725,  -12299, 22725,  -29692, //                  w13 w12 w09 w08
             22725,  12299,  -22725, -29692, //                  w07 w06 w03 w02
             -22725, 29692,  22725,  -12299, //                  w15 w14 w11 w10
             31521,  26722,  26722,  -6270,  //                  w21 w20 w17 w16
             17855,  -31521, 6270, -17855,   //                  w29 w28 w25 w24
             17855,  6270, -31521, -17855,   //                  w23 w22 w19 w18
             6270, 26722,  26722,  -31521};  //                  w31 w30 w27 w26

// Table for rows 2,6 - constants are multiplied on cos_2_16
                                                  //movq ->    w05 w04 w01 w00
__declspec(align(16)) short M128_tab_i_26[] = {21407,  27969,  21407,  11585,
             21407,  -11585, 21407,  -27969, //                  w13 w12 w09 w08
             21407,  11585,  -21407, -27969, //                  w07 w06 w03 w02
             -21407, 27969,  21407,  -11585, //                  w15 w14 w11 w10
             29692,  25172,  25172,  -5906,  //                  w21 w20 w17 w16
```

```
             16819,  -29692, 5906, -16819,    //                 w29 w28 w25 w24
             16819,  5906, -29692, -16819,    //                 w23 w22 w19 w18
             5906, 25172,  25172,  -29692};   //                 w31 w30 w27 w26

// Table for rows 3,5 - constants are multiplied on cos_3_16
                                                 //movq ->     w05 w04 w01 w00
__declspec(align(16)) short M128_tab_i_35[] = {19266,  25172,  19266,  10426,
             19266,  -10426, 19266,  -25172, //                 w13 w12 w09 w08
             19266,  10426,  -19266, -25172, //                 w07 w06 w03 w02
             -19266, 25172,  19266,  -10426, //                 w15 w14 w11 w10
             26722,  22654,  22654,  -5315,  //                 w21 w20 w17 w16
             15137,  -26722, 5315, -15137,    //                 w29 w28 w25 w24
             15137,  5315, -26722, -15137,    //                 w23 w22 w19 w18
             5315, 22654,  22654,  -26722};   //                 w31 w30 w27 w26


//-----------------------------------------------------------------------

/*
;=======================================================================
;=======================================================================
;=======================================================================
;
; Inverse DCT
;
;-----------------------------------------------------------------------
;
; This implementation calculates iDCT-2D by a row-column method.
; On the first stage the iDCT-1D is calculated for each row with use
; direct algorithm, on the second stage the calculation is executed
; at once for four columns with use of scaled iDCT-1D algorithm.
; Base R&Y algorithm for iDCT-1D is modified for second stage.
;
;=======================================================================

;-----------------------------------------------------------------------
;
; The first stage - inverse DCTs of rows
;
;-----------------------------------------------------------------------
```

```
; The 8-point inverse DCT direct algorithm
;-----------------------------------------------------------------------
;
; static const short w[32] = {
;    FIX(cos_4_16),  FIX(cos_2_16),  FIX(cos_4_16),  FIX(cos_6_16),
;    FIX(cos_4_16),  FIX(cos_6_16), -FIX(cos_4_16), -FIX(cos_2_16),
;    FIX(cos_4_16), -FIX(cos_6_16), -FIX(cos_4_16),  FIX(cos_2_16),
;    FIX(cos_4_16), -FIX(cos_2_16),  FIX(cos_4_16), -FIX(cos_6_16),
;    FIX(cos_1_16),  FIX(cos_3_16),  FIX(cos_5_16),  FIX(cos_7_16),
;    FIX(cos_3_16), -FIX(cos_7_16), -FIX(cos_1_16), -FIX(cos_5_16),
;    FIX(cos_5_16), -FIX(cos_1_16),  FIX(cos_7_16),  FIX(cos_3_16),
;    FIX(cos_7_16), -FIX(cos_5_16),  FIX(cos_3_16), -FIX(cos_1_16) };
;
; #define DCT_8_INV_ROW(x, y)
; {
;    int a0, a1, a2, a3, b0, b1, b2, b3;
;
;    a0   = x[0] * w[ 0] + x[2] * w[ 1] + x[4] * w[ 2] + x[6] * w[ 3];
;    a1   = x[0] * w[ 4] + x[2] * w[ 5] + x[4] * w[ 6] + x[6] * w[ 7];
;    a2   = x[0] * w[ 8] + x[2] * w[ 9] + x[4] * w[10] + x[6] * w[11];
;    a3   = x[0] * w[12] + x[2] * w[13] + x[4] * w[14] + x[6] * w[15];
;    b0   = x[1] * w[16] + x[3] * w[17] + x[5] * w[18] + x[7] * w[19];
;    b1   = x[1] * w[20] + x[3] * w[21] + x[5] * w[22] + x[7] * w[23];
;    b2   = x[1] * w[24] + x[3] * w[25] + x[5] * w[26] + x[7] * w[27];
;    b3   = x[1] * w[28] + x[3] * w[29] + x[5] * w[30] + x[7] * w[31];
;
;    y[0] = SHIFT_ROUND ( a0 + b0 );
;    y[1] = SHIFT_ROUND ( a1 + b1 );
;    y[2] = SHIFT_ROUND ( a2 + b2 );
;    y[3] = SHIFT_ROUND ( a3 + b3 );
;    y[4] = SHIFT_ROUND ( a3 - b3 );
;    y[5] = SHIFT_ROUND ( a2 - b2 );
;    y[6] = SHIFT_ROUND ( a1 - b1 );
;    y[7] = SHIFT_ROUND ( a0 - b0 );
; }
;
;-----------------------------------------------------------------------
;
; In this implementation the outputs of the iDCT-1D are multiplied
;    for rows 0,4 - on cos_4_16,
```

```
;    for rows 1,7 - on cos_1_16,
;    for rows 2,6 - on cos_2_16,
;    for rows 3,5 - on cos_3_16
; and are shifted to the left for rise of accuracy
;
; For used constants
;    FIX(float_const) = (short) (float_const * (1<<15) + 0.5)
;
;-----------------------------------------------------------------------------

;-----------------------------------------------------------------------------
;
; The second stage - inverse DCTs of columns
;
; The inputs are multiplied
;    for rows 0,4 - on cos_4_16,
;    for rows 1,7 - on cos_1_16,
;    for rows 2,6 - on cos_2_16,
;    for rows 3,5 - on cos_3_16
; and are shifted to the left for rise of accuracy
;
;-----------------------------------------------------------------------------
;
; The 8-point scaled inverse DCT algorithm (26a8m)
;
;-----------------------------------------------------------------------------
;
; #define DCT_8_INV_COL(x, y)
; {
;    short t0, t1, t2, t3, t4, t5, t6, t7;
;    short tp03, tm03, tp12, tm12, tp65, tm65;
;    short tp465, tm465, tp765, tm765;
;
;    tp765 = x[1]           + x[7] * tg_1_16;
;    tp465 = x[1] * tg_1_16 - x[7];
;    tm765 = x[5] * tg_3_16 + x[3];
;    tm465 = x[5]           - x[3] * tg_3_16;
;
;    t7    = tp765 + tm765;
;    tp65  = tp765 - tm765;
```

```
;    t4    = tp465 + tm465;
;    tm65  = tp465 - tm465;
;
;    t6    = ( tp65 + tm65 ) * cos_4_16;
;    t5    = ( tp65 - tm65 ) * cos_4_16;
;
;    tp03  = x[0] + x[4];
;    tp12  = x[0] - x[4];
;
;    tm03  = x[2]          + x[6] * tg_2_16;
;    tm12  = x[2] * tg_2_16 - x[6];
;
;    t0    = tp03 + tm03;
;    t3    = tp03 - tm03;
;    t1    = tp12 + tm12;
;    t2    = tp12 - tm12;
;
;    y[0]  = SHIFT_ROUND ( t0 + t7 );
;    y[7]  = SHIFT_ROUND ( t0 - t7 );
;    y[1]  = SHIFT_ROUND ( t1 + t6 );
;    y[6]  = SHIFT_ROUND ( t1 - t6 );
;    y[2]  = SHIFT_ROUND ( t2 + t5 );
;    y[5]  = SHIFT_ROUND ( t2 - t5 );
;    y[3]  = SHIFT_ROUND ( t3 + t4 );
;    y[4]  = SHIFT_ROUND ( t3 - t4 );
; }
;
;-----------------------------------------------------------------------------
*/
//xmm7 = round_inv_row
#define DCT_8_INV_ROW __asm{                 \
     __asm pshuflw    xmm0, xmm0, 0xD8       \
     __asm pshufd     xmm1, xmm0, 0          \
     __asm pmaddwd    xmm1, [esi]            \
     __asm pshufd     xmm3, xmm0, 0x55       \
     __asm pshufhw    xmm0, xmm0, 0xD8       \
     __asm pmaddwd    xmm3, [esi+32]         \
     __asm pshufd     xmm2, xmm0, 0xAA       \
     __asm pshufd     xmm0, xmm0, 0xFF       \
     __asm pmaddwd    xmm2, [esi+16]         \
```

```
      __asm pshufhw    xmm4, xmm4, 0xD8        \
      __asm paddd      xmm1, M128_round_inv_row \
      __asm pshuflw    xmm4, xmm4, 0xD8        \
      __asm pmaddwd    xmm0, [esi+48]          \
      __asm pshufd     xmm5, xmm4, 0           \
      __asm pshufd     xmm6, xmm4, 0xAA        \
      __asm pmaddwd    xmm5, [ecx]             \
      __asm paddd      xmm1, xmm2              \
      __asm movdqa     xmm2, xmm1              \
      __asm pshufd     xmm7, xmm4, 0x55        \
      __asm pmaddwd    xmm6, [ecx+16]          \
      __asm paddd      xmm0, xmm3              \
      __asm pshufd     xmm4, xmm4, 0xFF        \
      __asm psubd      xmm2, xmm0              \
      __asm pmaddwd    xmm7, [ecx+32]          \
      __asm paddd      xmm0, xmm1              \
      __asm psrad      xmm2, 12                \
      __asm paddd      xmm5, M128_round_inv_row \
      __asm pmaddwd    xmm4, [ecx+48]          \
      __asm paddd      xmm5, xmm6              \
      __asm movdqa     xmm6, xmm5              \
      __asm psrad      xmm0, 12                \
      __asm pshufd     xmm2, xmm2, 0x1B        \
      __asm packssdw   xmm0, xmm2              \
      __asm paddd      xmm4, xmm7              \
      __asm psubd      xmm6, xmm4              \
      __asm paddd      xmm4, xmm5              \
      __asm psrad      xmm6, 12             \
      __asm psrad      xmm4, 12             \
      __asm pshufd     xmm6, xmm6, 0x1B     \
      __asm packssdw   xmm4, xmm6           \
   }


#define DCT_8_INV_COL_8 __asm{                          \
      __asm movdqa     xmm1, XMMWORD PTR M128_tg_3_16    \
      __asm movdqa     xmm2, xmm0                        \
      __asm movdqa     xmm3, XMMWORD PTR [edx+3*16]      \
      __asm pmulhw     xmm0, xmm1                        \
      __asm pmulhw     xmm1, xmm3                        \
```

```
__asm movdqa    xmm5, XMMWORD PTR M128_tg_1_16       \
__asm movdqa    xmm6, xmm4                           \
__asm pmulhw    xmm4, xmm5                           \
__asm paddsw    xmm0, xmm2                           \
__asm pmulhw    xmm5, [edx+1*16]                     \
__asm paddsw    xmm1, xmm3                           \
__asm movdqa    xmm7, XMMWORD PTR [edx+6*16]         \
__asm paddsw    xmm0, xmm3                           \
__asm movdqa    xmm3, XMMWORD PTR M128_tg_2_16       \
__asm psubsw    xmm2, xmm1                           \
__asm pmulhw    xmm7, xmm3                           \
__asm movdqa    xmm1, xmm0                           \
__asm pmulhw    xmm3, [edx+2*16]                     \
__asm psubsw    xmm5, xmm6                           \
__asm paddsw    xmm4, [edx+1*16]                     \
__asm paddsw    xmm0, xmm4                           \
__asm paddsw    xmm0, XMMWORD PTR M128_one_corr      \
__asm psubsw    xmm4, xmm1                           \
__asm movdqa    xmm6, xmm5                           \
__asm psubsw    xmm5, xmm2                           \
__asm paddsw    xmm5, XMMWORD PTR M128_one_corr      \
__asm paddsw    xmm6, xmm2                           \
__asm movdqa    [edx+7*16],  xmm0                    \
__asm movdqa    xmm1, xmm4                           \
__asm movdqa    xmm0, XMMWORD PTR M128_cos_4_16      \
__asm paddsw    xmm4, xmm5                           \
__asm movdqa    xmm2, XMMWORD PTR M128_cos_4_16      \
__asm pmulhw    xmm2, xmm4                           \
__asm movdqa    [edx+3*16],  xmm6                    \
__asm psubsw    xmm1, xmm5                           \
__asm paddsw    xmm7, [edx+2*16]                     \
__asm psubsw    xmm3, [edx+6*16]                     \
__asm movdqa    xmm6, [edx]                          \
__asm pmulhw    xmm0, xmm1                           \
__asm movdqa    xmm5, [edx+4*16]                     \
__asm paddsw    xmm5, xmm6                           \
__asm psubsw    xmm6, [edx+4*16]                     \
__asm paddsw    xmm4, xmm2                           \
__asm por     xmm4, XMMWORD PTR M128_one_corr        \
__asm paddsw    xmm0, xmm1                           \
```

```
    __asm por     xmm0, XMMWORD PTR M128_one_corr         \
    __asm movdqa     xmm2, xmm5                           \
    __asm paddsw     xmm5, xmm7                           \
    __asm movdqa     xmm1, xmm6                           \
    __asm paddsw     xmm5, XMMWORD PTR M128_round_inv_col \
    __asm psubsw     xmm2, xmm7                           \
    __asm movdqa     xmm7, [edx+7*16]                     \
    __asm paddsw     xmm6, xmm3                           \
    __asm paddsw     xmm6, XMMWORD PTR M128_round_inv_col \
    __asm paddsw     xmm7, xmm5                           \
    __asm psraw   xmm7, SHIFT_INV_COL                     \
    __asm psubsw     xmm1, xmm3                           \
    __asm paddsw     xmm1, XMMWORD PTR M128_round_inv_corr\
    __asm movdqa     xmm3, xmm6                           \
    __asm paddsw     xmm2, XMMWORD PTR M128_round_inv_corr\
    __asm paddsw     xmm6, xmm4                           \
    __asm movdqa     [edx], xmm7                          \
    __asm psraw   xmm6, SHIFT_INV_COL                     \
    __asm movdqa     xmm7, xmm1                           \
    __asm paddsw     xmm1, xmm0                           \
    __asm movdqa     [edx+1*16], xmm6                     \
    __asm psraw   xmm1, SHIFT_INV_COL                     \
    __asm movdqa     xmm6, [edx+3*16]                     \
    __asm psubsw     xmm7, xmm0                           \
    __asm psraw   xmm7, SHIFT_INV_COL                     \
    __asm movdqa     [edx+2*16],  xmm1                    \
    __asm psubsw     xmm5, [edx+7*16]                     \
    __asm psraw   xmm5, SHIFT_INV_COL                     \
    __asm movdqa     [edx+7*16], xmm5                     \
    __asm psubsw     xmm3, xmm4                           \
    __asm paddsw     xmm6, xmm2                           \
    __asm psubsw     xmm2, [edx+3*16]                     \
    __asm psraw   xmm6, SHIFT_INV_COL                     \
    __asm psraw   xmm2, SHIFT_INV_COL                     \
    __asm movdqa     [edx+3*16],  xmm6                    \
    __asm psraw   xmm3, SHIFT_INV_COL                     \
    __asm movdqa     [edx+4*16],  xmm2                    \
    __asm movdqa     [edx+5*16],  xmm7                    \
    __asm movdqa     [edx+6*16],  xmm3                    \
}
```

```
//assumes src and destination are aligned on a 16-byte boundary

int idct_M128ASM::TheCode()
{
// assert(((src & 0xf) == 0)&&((dst & 0xf) == 0))   //aligned on 16-byte boundary
      short* src = dctcoeffshort;
      short* dst = tst;
      __asm mov       eax, src
      __asm mov       edx, dst
      //.....................................................................//
      __asm movdqa    xmm0, XMMWORD PTR[eax]  //row 1
      __asm lea       esi, M128_tab_i_04
      __asm movdqa    xmm4, XMMWORD PTR[eax+16*2] //row 3

      __asm lea       ecx, M128_tab_i_26
      DCT_8_INV_ROW;  //Row 1, tab_i_04 and Row 3, tab_i_26
      __asm movdqa    XMMWORD PTR[edx],    xmm0
      __asm movdqa    XMMWORD PTR[edx+16*2],  xmm4
      //.....................................................................//
      __asm movdqa    xmm0, XMMWORD PTR[eax+16*4]   //row 5
      //__asm lea       esi, M128_tab_i_04
      __asm movdqa    xmm4, XMMWORD PTR[eax+16*6] //row 7

      //__asm lea       ecx, M128_tab_i_26
      DCT_8_INV_ROW;  //Row 5, tab_i_04 and Row 7, tab_i_26
      __asm movdqa    XMMWORD PTR[edx+16*4],  xmm0
      __asm movdqa    XMMWORD PTR[edx+16*6],  xmm4
      //.....................................................................//
      __asm movdqa    xmm0, XMMWORD PTR[eax+16*3]   //row 4
      __asm lea       esi, M128_tab_i_35
      __asm movdqa    xmm4, XMMWORD PTR[eax+16*1] //row 2

      __asm lea       ecx, M128_tab_i_17
      DCT_8_INV_ROW;  //Row 4, tab_i_35 and Row 2, tab_i_17
      __asm movdqa    XMMWORD PTR[edx+16*3],  xmm0
      __asm movdqa    XMMWORD PTR[edx+16*1],  xmm4
      //.....................................................................//
```

```
    __asm movdqa     xmm0, XMMWORD PTR[eax+16*5]    //row 6
    //__asm lea        esi, M128_tab_i_35
    __asm movdqa     xmm4, XMMWORD PTR[eax+16*7] //row 8

    //__asm lea        ecx, M128_tab_i_17
    DCT_8_INV_ROW;  //Row 6, tab_i_35 and Row 8, tab_i_17
    //__asm movdqa     XMMWORD PTR[edx+80],        xmm0
    //__asm movdqa     xmm0, XMMWORD PTR [edx+80]  /* 0         /* x5 */
    //__asm movdqa     XMMWORD PTR[edx+16*7],      xmm4
    //__asm movdqa     xmm4, XMMWORD PTR [edx+7*16]/* 4         ; x7 */

    DCT_8_INV_COL_8
    // __asm emms
    return(EXIT_SUCCESS);

}
```

7 SSE2 Instructions IVEC Coding Example

The code in this section uses the SIMD Classes implementation of the SSE2 instructions to perform a 2D IDCT. The following provides a summary of this 2-D IDCT implementation:

1. Perform a 1-D inverse DCT on each of the eight rows. A macro called DCT_8_INV_ROW is used to perform the 1-D inverse DCT on two rows. The macro expects the rows to have been previously loaded into the row and r2ow Is16vec8 variables. The macro also expects the table1 and table2 Is16vec8 pointers to point to the corresponding constant multiplier tables. The macro needs to be called four times to complete the 1-D inverse DCT on each of the eight rows. A description of the DCT_8_INV_ROW macro can be found in the comments below.
2. Perform a 1-D inverse DCT on each of the eight columns. This implementation does not use the macro: DCT_8_INV_COL_8 to perform the 1-D inverse DCT on eight columns. Instead the 1-D inverse DCT code has been included in the function. A description of this 1-D inverse DCT implementation can be found in the comments below.

```
#include <dvec.h>
#include "idct_kernel.h"

#define BITS_INV_ACC    4                               // 4 or 5 for IEEE
#define SHIFT_INV_ROW   16 - BITS_INV_ACC
#define SHIFT_INV_COL   1 + BITS_INV_ACC
const short RND_INV_ROW   = 1024 * (6 - BITS_INV_ACC);        //1 << (SHIFT_INV_ROW-1)
const short RND_INV_COL   = 16 * (BITS_INV_ACC - 3);          // 1 << (SHIFT_INV_COL-1)
const short RND_INV_CORR  = RND_INV_COL - 1;                  // correction -1.0 and round

Is16vec8 ivec_one_corr(1,  1, 1, 1, 1, 1, 1, 1);
Is16vec8 ivec_round_inv_row(0,RND_INV_ROW,0,RND_INV_ROW,0,RND_INV_ROW,0,
RND_INV_ROW);
Is16vec8 ivec_round_inv_col(RND_INV_COL,  RND_INV_COL,  RND_INV_COL,  RND_INV_COL,
   RND_INV_COL,  RND_INV_COL,  RND_INV_COL,  RND_INV_COL);
Is16vec8 ivec_round_inv_corr(RND_INV_CORR, RND_INV_CORR, RND_INV_CORR, RND_INV_CORR,
RND_INV_CORR, RND_INV_CORR, RND_INV_CORR, RND_INV_CORR);

Is16vec8 ivec_tg_1_16(13036,  13036,  13036,  13036,13036,  13036,  13036,  13036);
Is16vec8 ivec_tg_2_16(27146,  27146,  27146,  27146,27146,  27146,  27146,  27146);
Is16vec8 ivec_tg_3_16(-21746, -21746, -21746, -21746, -21746, -21746,-21746,-21746);
Is16vec8 ivec_cos_4_16(-19195, -19195, -19195, -19195,-19195, -19195,-19195,-19195);


//---------------------------------------------------------------------------

// Table for rows 0,4 - constants are multiplied on cos_4_16
```

```
//movq ->  w13 w12 w09 w08 w05 w04 w01 w00
//       w15 w14 w11 w10 w07 w06 w03 w02
//       w29 w28 w25 w24 w21 w20 w17 w16
//       w31 w30 w27 w26 w23 w22 w19 w18
                                                  //movq ->     w05 w04 w01 w00
__declspec(align(16)) short M128tab_i_04[] = {16384,       21407, 16384, 8867,
             16384,  -8867,  16384, -21407,  //                  w13 w12 w09 w08
             16384,  8867, -16384, -21407,   //                  w07 w06 w03 w02
             -16384, 21407,  16384,  -8867,  //                  w15 w14 w11 w10
             22725,  19266,  19266,  -4520,  //                  w21 w20 w17 w16
             12873,  -22725, 4520, -12873,   //                  w29 w28 w25 w24
             12873,  4520, -22725, -12873,   //                  w23 w22 w19 w18
             4520, 19266,  19266,  -22725};  //                  w31 w30 w27 w26

// Table for rows 1,7 - constants are multiplied on cos_1_16
                                                  //movq ->     w05 w04 w01 w00
__declspec(align(16)) short M128tab_i_17[] = {22725,  29692,  22725,  12299,
             22725,  -12299, 22725,  -29692, //                  w13 w12 w09 w08
             22725,  12299,  -22725, -29692, //                  w07 w06 w03 w02
             -22725, 29692,  22725,  -12299, //                  w15 w14 w11 w10
             31521,  26722,  26722,  -6270,  //                  w21 w20 w17 w16
             17855,  -31521, 6270, -17855,   //                  w29 w28 w25 w24
             17855,  6270, -31521, -17855,   //                  w23 w22 w19 w18
             6270, 26722,  26722,  -31521};  //                  w31 w30 w27 w26

// Table for rows 2,6 - constants are multiplied on cos_2_16
                                                  //movq ->     w05 w04 w01 w00
__declspec(align(16)) short M128tab_i_26[] = {21407,  27969,  21407,  11585,
             21407,  -11585, 21407,  -27969, //                  w13 w12 w09 w08
             21407,  11585,  -21407, -27969, //                  w07 w06 w03 w02
             -21407, 27969,  21407,  -11585, //                  w15 w14 w11 w10
             29692,  25172,  25172,  -5906,  //                  w21 w20 w17 w16
             16819,  -29692, 5906, -16819,   //                  w29 w28 w25 w24
             16819,  5906, -29692, -16819,   //                  w23 w22 w19 w18
             5906, 25172,  25172,  -29692};  //                  w31 w30 w27 w26

// Table for rows 3,5 - constants are multiplied on cos_3_16
```

```
                                                    //movq ->    w05 w04 w01 w00
__declspec(align(16)) short M128tab_i_35[] = {19266,  25172,  19266,  10426,
             19266,  -10426, 19266,  -25172, //                  w13 w12 w09 w08
             19266,  10426,  -19266, -25172, //                  w07 w06 w03 w02
             -19266, 25172,  19266,  -10426, //                  w15 w14 w11 w10
             26722,  22654,  22654,  -5315,  //                  w21 w20 w17 w16
             15137,  -26722, 5315, -15137,   //                  w29 w28 w25 w24
             15137,  5315, -26722, -15137,   //                  w23 w22 w19 w18
             5315, 22654,  22654,  -26722};  //                  w31 w30 w27 w26

Is16vec8 *ivec_tab_i_04 = (Is16vec8*)M128tab_i_04;
Is16vec8 *ivec_tab_i_17 = (Is16vec8*)M128tab_i_17;
Is16vec8 *ivec_tab_i_26 = (Is16vec8*)M128tab_i_26;
Is16vec8 *ivec_tab_i_35 = (Is16vec8*)M128tab_i_35;

//-----------------------------------------------------------------------------

/*
;=============================================================================
;=============================================================================
;=============================================================================
;
; Inverse DCT
;
;-----------------------------------------------------------------------------
;
; This implementation calculates iDCT-2D by a row-column method.
; On the first stage the iDCT-1D is calculated for each row with use
; direct algorithm, on the second stage the calculation is executed
; at once for four columns with use of scaled iDCT-1D algorithm.
; Base R&Y algorithm for iDCT-1D is modified for second stage.
;
;=============================================================================

;-----------------------------------------------------------------------------
;
; The first stage - inverse DCTs of rows
;
;-----------------------------------------------------------------------------
; The 8-point inverse DCT direct algorithm
```

```
;-----------------------------------------------------------------------------
;
; static const short w[32] = {
;    FIX(cos_4_16),  FIX(cos_2_16),  FIX(cos_4_16),  FIX(cos_6_16),
;    FIX(cos_4_16),  FIX(cos_6_16), -FIX(cos_4_16), -FIX(cos_2_16),
;    FIX(cos_4_16), -FIX(cos_6_16), -FIX(cos_4_16),  FIX(cos_2_16),
;    FIX(cos_4_16), -FIX(cos_2_16),  FIX(cos_4_16), -FIX(cos_6_16),
;    FIX(cos_1_16),  FIX(cos_3_16),  FIX(cos_5_16),  FIX(cos_7_16),
;    FIX(cos_3_16), -FIX(cos_7_16), -FIX(cos_1_16), -FIX(cos_5_16),
;    FIX(cos_5_16), -FIX(cos_1_16),  FIX(cos_7_16),  FIX(cos_3_16),
;    FIX(cos_7_16), -FIX(cos_5_16),  FIX(cos_3_16), -FIX(cos_1_16) };
;
; #define DCT_8_INV_ROW(x, y)
; {
;    int a0, a1, a2, a3, b0, b1, b2, b3;
;
;    a0   = x[0] * w[ 0] + x[2] * w[ 1] + x[4] * w[ 2] + x[6] * w[ 3];
;    a1   = x[0] * w[ 4] + x[2] * w[ 5] + x[4] * w[ 6] + x[6] * w[ 7];
;    a2   = x[0] * w[ 8] + x[2] * w[ 9] + x[4] * w[10] + x[6] * w[11];
;    a3   = x[0] * w[12] + x[2] * w[13] + x[4] * w[14] + x[6] * w[15];
;    b0   = x[1] * w[16] + x[3] * w[17] + x[5] * w[18] + x[7] * w[19];
;    b1   = x[1] * w[20] + x[3] * w[21] + x[5] * w[22] + x[7] * w[23];
;    b2   = x[1] * w[24] + x[3] * w[25] + x[5] * w[26] + x[7] * w[27];
;    b3   = x[1] * w[28] + x[3] * w[29] + x[5] * w[30] + x[7] * w[31];
;
;    y[0] = SHIFT_ROUND ( a0 + b0 );
;    y[1] = SHIFT_ROUND ( a1 + b1 );
;    y[2] = SHIFT_ROUND ( a2 + b2 );
;    y[3] = SHIFT_ROUND ( a3 + b3 );
;    y[4] = SHIFT_ROUND ( a3 - b3 );
;    y[5] = SHIFT_ROUND ( a2 - b2 );
;    y[6] = SHIFT_ROUND ( a1 - b1 );
;    y[7] = SHIFT_ROUND ( a0 - b0 );
; }
;
;-----------------------------------------------------------------------------
;
; In this implementation the outputs of the iDCT-1D are multiplied
;    for rows 0,4 - on cos_4_16,
;    for rows 1,7 - on cos_1_16,
```

```
;    for rows 2,6 - on cos_2_16,
;    for rows 3,5 - on cos_3_16
; and are shifted to the left for rise of accuracy
;
; For used constants
;    FIX(float_const) = (short) (float_const * (1<<15) + 0.5)
;
;-----------------------------------------------------------------------------

;-----------------------------------------------------------------------------
;
; The second stage - inverse DCTs of columns
;
; The inputs are multiplied
;    for rows 0,4 - on cos_4_16,
;    for rows 1,7 - on cos_1_16,
;    for rows 2,6 - on cos_2_16,
;    for rows 3,5 - on cos_3_16
; and are shifted to the left for rise of accuracy
;
;-----------------------------------------------------------------------------
;
; The 8-point scaled inverse DCT algorithm (26a8m)
;
;-----------------------------------------------------------------------------
; //Inverse 1-D DCT algorithm for the eight columns
;    short t0, t1, t2, t3, t4, t5, t6, t7;
;    short tp03, tm03, tp12, tm12, tp65, tm65;
;    short tp465, tm465, tp765, tm765;
;
;    tp765 = x[1]           + x[7] * tg_1_16;
;    tp465 = x[1] * tg_1_16 - x[7];
;    tm765 = x[5] * tg_3_16 + x[3];
;    tm465 = x[5]           - x[3] * tg_3_16;
;
;    t7    = tp765 + tm765;
;    tp65  = tp765 - tm765;
;    t4    = tp465 + tm465;
;    tm65  = tp465 - tm465;
;
```

```
;    t6    = ( tp65 + tm65 ) * cos_4_16;
;    t5    = ( tp65 - tm65 ) * cos_4_16;
;
;    tp03  = x[0] + x[4];
;    tp12  = x[0] - x[4];
;
;    tm03  = x[2]          + x[6] * tg_2_16;
;    tm12  = x[2] * tg_2_16 - x[6];
;
;    t0    = tp03 + tm03;
;    t3    = tp03 - tm03;
;    t1    = tp12 + tm12;
;    t2    = tp12 - tm12;
;
;    y[0]  = SHIFT_ROUND ( t0 + t7 );
;    y[7]  = SHIFT_ROUND ( t0 - t7 );
;    y[1]  = SHIFT_ROUND ( t1 + t6 );
;    y[6]  = SHIFT_ROUND ( t1 - t6 );
;    y[2]  = SHIFT_ROUND ( t2 + t5 );
;    y[5]  = SHIFT_ROUND ( t2 - t5 );
;    y[3]  = SHIFT_ROUND ( t3 + t4 );
;    y[4]  = SHIFT_ROUND ( t3 - t4 );
;
;------------------------------------------------------------------------
*/
//emm7 = round_inv_row
#define DCT_8_INV_ROWX2                                                    \
     row =   (Is16vec8)_mm_shufflelo_epi16(row,0xd8);                      \
     row20 = (Is16vec8)_mm_shuffle_epi32(row,0x00);                        \
     row20 = (Is16vec8)_mm_madd_epi16(row20,table1[0]);                    \
     row31 = (Is16vec8)_mm_shuffle_epi32(row,0x55);                        \
     row =   (Is16vec8)_mm_shufflehi_epi16(row,0xd8);                      \
     row31 = (Is16vec8)_mm_madd_epi16(row31,table1[2]);                    \
     /*-----------------------------------------------------------------*/  \
     row64 = (Is16vec8)_mm_shuffle_epi32(row,0xaa);                        \
     row75 = (Is16vec8)_mm_shuffle_epi32(row,0xff);                        \
     /*-----------------------------------------------------------------*/  \
     row64 = (Is16vec8)_mm_madd_epi16(row64,table1[1]);                    \
     r2ow =  (Is16vec8)_mm_shufflehi_epi16(r2ow,0xd8);                     \
     row20 = (Is32vec4)row20 + (Is32vec4)ivec_round_inv_row;               \
```

```
      r2ow =  (Is16vec8)_mm_shufflelo_epi16(r2ow,0xd8);                          \
      row75 = (Is16vec8)_mm_madd_epi16(row75,table1[3]);                         \
      r2ow20 = (Is16vec8)_mm_shuffle_epi32(r2ow,0x00);                           \
      r2ow64 = (Is16vec8)_mm_shuffle_epi32(r2ow,0xaa);                           \
      r2ow20 = (Is16vec8)_mm_madd_epi16(r2ow20,table2[0]);                       \
      /*-----------------------------------------------------------------*/     \
      a = (Is32vec4)row20 + (Is32vec4)row64;                                     \
      r2ow31 = (Is16vec8)_mm_shuffle_epi32(r2ow,0x55);                           \
      r2ow64 = (Is16vec8)_mm_madd_epi16(r2ow64,table2[1]);                       \
      b = (Is32vec4)row31 + (Is32vec4)row75;                                     \
      r2ow75 = (Is16vec8)_mm_shuffle_epi32(r2ow,0xff);                           \
      Yhigh = ((Is32vec4)((Is32vec4)a-(Is32vec4)b)) >> SHIFT_INV_ROW;            \
      r2ow31 = (Is16vec8)_mm_madd_epi16(r2ow31,table2[2]);                       \
      Ylow =  ((Is32vec4)((Is32vec4)a+(Is32vec4)b)) >> SHIFT_INV_ROW;            \
      r2ow20 = (Is32vec4)r2ow20 + (Is32vec4)ivec_round_inv_row;                  \
      r2ow75 = (Is16vec8)_mm_madd_epi16(r2ow75,table2[3]);                       \
      a2 = (Is32vec4)r2ow20 + (Is32vec4)r2ow64;                                  \
      Yhigh = (Is16vec8)_mm_shuffle_epi32(Yhigh, 0x1b);                          \
      b2 = (Is32vec4)r2ow31 + (Is32vec4)r2ow75;                                  \
      Y2high = ((Is32vec4)((Is32vec4)a2-(Is32vec4)b2)) >> SHIFT_INV_ROW;         \
      Y2low =  ((Is32vec4)((Is32vec4)a2+(Is32vec4)b2)) >> SHIFT_INV_ROW;         \
      Y2high = (Is16vec8)_mm_shuffle_epi32(Y2high, 0x1b);                        \

Is16vec8 tm765;
Is16vec8 tm465;
Is16vec8 tp765;
Is16vec8 tp465;
Is16vec8 tm65;
Is16vec8 tp65;
Is16vec8 tmp;
Is16vec8 tm03;
Is16vec8 tp03;
Is16vec8 tm12;
Is16vec8 tp12;
Is16vec8 t0;
Is16vec8 t1;
Is16vec8 t2;
Is16vec8 t3;
Is16vec8 t4;
Is16vec8 t5;
```

```
Is16vec8 t6;
Is16vec8 t7;

//assumes src and destination are aligned on a 16-byte boundary

int idct_M128IVEC::TheCode()
{
// assert(((src & 0xf) == 0)&&((dst & 0xf) == 0))   //aligned on 16-byte boundary
      Is16vec8* src = (Is16vec8*)dctcoeffshort;
      Is16vec8* dst = (Is16vec8*)tst;
      Is16vec8 row20, row64, row31, row75, r2ow20, r2ow64, r2ow31, r2ow75;
      Is16vec8 a, b, Ylow, Yhigh, a2, b2, Y2low, Y2high;

      Is16vec8 row = src[2], r2ow = src[3];
      Is16vec8 *table1 = ivec_tab_i_26, *table2 = ivec_tab_i_35;
      //
      DCT_8_INV_ROWX2;   //Row 3, tab_i_26  //Row 4, tab_i_35
      //
      dst[2] = (Is16vec8)_mm_packs_epi32(Ylow, Yhigh);
      dst[3] = (Is16vec8)_mm_packs_epi32(Y2low, Y2high);

      row = src[4];      r2ow = src[5];
      table1 = ivec_tab_i_04; //table2 = ivec_tab_i_35;
      //
      DCT_8_INV_ROWX2;   //Row 5, tab_i_04  //Row 6, tab_i_35
      //
      dst[4] = (Is16vec8)_mm_packs_epi32(Ylow, Yhigh);
      dst[5] = (Is16vec8)_mm_packs_epi32(Y2low, Y2high);

      row = src[0];      r2ow = src[1];
      /*table1 = ivec_tab_i_04;*/  table2 = ivec_tab_i_17;
      //
      DCT_8_INV_ROWX2;   //Row 1, tab_i_04  //Row 2, tab_i_17
      //
      dst[0] = (Is16vec8)_mm_packs_epi32(Ylow, Yhigh);

      dst[1] = (Is16vec8)_mm_packs_epi32(Y2low, Y2high);

      row = src[6];      r2ow = src[7];
      table1 = ivec_tab_i_26; //table2 = ivec_tab_i_17;
      //
```

```
DCT_8_INV_ROWX2;   //Row 7, tab_i_26  //Row 8, tab_i_17
//
dst[6] = (Is16vec8)_mm_packs_epi32(Ylow, Yhigh);
dst[7] = (Is16vec8)_mm_packs_epi32(Y2low, Y2high);    //store in Y2high
                                                      //so compiler can
                                                      //keep value in
                                                      //SIMD register


//
tm765 = mul_high(ivec_tg_3_16,dst[5]) + dst[5] + dst[3];
tm465 = dst[5] - (mul_high(ivec_tg_3_16,dst[3]) + dst[3]);
tp765 = mul_high(dst[7],ivec_tg_1_16) + dst[1];
tp465 = mul_high(dst[1],ivec_tg_1_16) - dst[7];
/*-------------------------------------------------------*/
tm65 = tp465 - tm465 + ivec_one_corr;
tp65 = tp765 - tm765;
t4 = tp465 + tm465;
t7 = tp765 + tm765 + ivec_one_corr;
/*-------------------------------------------------------*/
tmp = tp65 + tm65;
t6 = mul_high(tmp,ivec_cos_4_16) + tmp;
t6 |= ivec_one_corr;
tmp = tp65 - tm65;
t5 = mul_high(tmp,ivec_cos_4_16) + tmp;
t5 |= ivec_one_corr;
/*-------------------------------------------------------*/
tm03 = mul_high(dst[6],ivec_tg_2_16) + dst[2];
tm12 = mul_high(dst[2],ivec_tg_2_16) - dst[6];
tp03 = dst[0] + dst[4];
tp12 = dst[0] - dst[4];
/*-------------------------------------------------------*/
t0 = tp03 + tm03 + ivec_round_inv_col;
t3 = tp03 - tm03 + ivec_round_inv_corr;
t1 = tp12 + tm12 + ivec_round_inv_col;
t2 = tp12 - tm12 + ivec_round_inv_corr;
/*-------------------------------------------------------*/
dst[0] = (t0+t7) >> SHIFT_INV_COL;
dst[1] = (t1+t6) >> SHIFT_INV_COL;
dst[2] = (t2+t5) >> SHIFT_INV_COL;
```

```
    dst[3] = (t3+t4) >> SHIFT_INV_COL;
    dst[4] = (t3-t4) >> SHIFT_INV_COL;
    dst[5] = (t2-t5) >> SHIFT_INV_COL;
    dst[6] = (t1-t6) >> SHIFT_INV_COL;
    dst[7] = (t0-t7) >> SHIFT_INV_COL;

    return(EXIT_SUCCESS);

}
```

Appendix A - Performance Data

Performance Data Revision History

Revision	Revision History	Date
2.0	Update with 1.2 GHz Pentium® 4 processor performance data	7/00
1.0	Original publication of document	9/99

Table 1: Performance Data of IDCT Implementations

Performance Data in Microseconds		
Test Cases	**Pentium III Processor (733 MHz)**	**Pentium 4 Processor (1.2 GHz)**
Streaming SIMD Extensions (SSE) ASM	0.386	0.335
SSE2 ASM	-	0.255
SSE2 IVEC	-	0.277

Table 2: Speedups from Table 1 Performance Data

Implementations and Platforms	Speedup
Pentium 4 Processor (SSE2 ASM vs. SSE ASM)	1.31
SSE ASM (Pentium 4 Processor vs. PentiumIII Processor)	1.15
SSE2 ASM on Pentium 4 Processor vs. SSE on PentiumIII processor	1.51

The performance on a Pentium III processor was measured with a 733 MHz Pentium III. The performance on a Pentium 4 processor was measured with a 1.2 GHz Pentium 4 processor. Table 2 shows that the SSE2 instructions are 1.31 times faster than the SSE instructions when both implementations are executed on a Pentium 4 processor. This speedup is attributed to the following optimizations:

- **Larger SIMD width.** The integer SSE2 instructions use the 128-bit XMM registers instead of the 64-bit MMX™ technology registers. The increased SIMD width decreased register pressure and doubled the amount of data processed per instruction.
- **Unrolling of the row 1D IDCT.** The decreased register pressure allowed the row 1D IDCT to be unrolled to operate on two rows at a time. The unrolling was done to execute more instructions in parallel. Specifically, the unrolling enabled the `pmaddwd` instructions to be processed sooner and in parallel with other instructions.

- **Decreased register pressure.** The decreased register pressure eliminated eight instructions in the assembly (ASM) implementation. The ASM implementation removed two stores and two loads by keeping the values in the registers. Also, since the unrolling of the row IDCT operated on two rows at a time, pairing the rows properly resulted in eliminating four load effective address (`lea`) instructions. The eight eliminated instructions are commented out and labeled in the ASM implementation (see section 6).

The ASM implementation is slightly faster than the class library (IVEC) implementation. Both implementations use the SSE2 instructions; the difference is that the IVEC implementation uses the Intel® C++ Class Libraries for SIMD Operations, which are a C++ wrapper to the Intel® C\C++ Compiler intrinsics.

The intrinsics are instructions that use a C-function call syntax and usually have a one-to-one mapping to the SSE2 instructions. The advantage of using the intrinsics or C++ SIMD classes is in being able to use the SSE2 instructions without having to resort to tedious assembly language programming. The disadvantage is that certain optimizations can only be achieved using assembly language. In this case, the ASM implementation removed two stores and two loads by keeping the values in the registers. The IVEC implementation does not provide this optimization because the compiler controls when or how the registers are spilled. For this reason, this ASM implementation is slightly faster than the IVEC implementation.

However, the reader should be aware that inline ASM might turn off some inter-procedural and intra-procedural compiler optimizations. Thus, for most cases, the C++ SIMD classes or intrinsics provide better application speedups than inline ASM. For more information on the C++ Class Libraries and intrinsics, please see the Intel C\C++ Class Libraries for SIMD Operations: With Support for the SSE2 instructions, order number 749100-001 and the Intel C/C++ Compiler Intrinsics Reference Manual, order number 748639-001.

Test System Configuration

Table 3: Pentium III Configuration

Processor	Pentium III Processor at 733 MHz
System	Intel® Desktop Board VC820
Bios Version	VC82010A.86A.0028.P10
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster† Annihilator† Pro AGP nVidia GeForce256† DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows† 2000 Build 2195

Table 4: Pentium 4 Configuration

Processor	Pentium 4 Processor at 1.2 GHz
System	Intel Desktop Board D850GB
Bios Version	GB85010A.86A.0014.D.0007201756
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows 2000 Build 2195