A Fast Precise Implementation of 8x8 Discrete Cosine Transform Using the Streaming SIMD Extensions and MMX™ Instructions

Version 1.0

4/99

Order Number: 742474-001

Table of Contents

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. N. Ahmed, T. Natarajan, and K.R. Rao. Discrete cosine transform. IEEE Trans. Comput., vol. C-23, pp. 90-93, Jan. 1974.

2. K.R. Rao and P. Yip. Discrete Cosine Transform. Algorithms, Advantages, Applications. Academic Press, Inc, London, 1990.

3. Z. Wang. Fast algorithms for the discrete W transform and for the discrete Fourier transform. IEEE Trans. Acoust., Speech and Signal Process., vol. ASSP-32, pp. 803-816, Aug. 1984.

4. N. Suehiro and M. Hatori. Fast algorithms for the DFT and other sinusoidal transforms. IEEE Trans. Acoust., Speech and Signal Processing, vol. ASSP-34, pp. 642-644, June 1986.

5. B.G. Lee. A new algorithm to compute the discrete cosine transform. IEEE Trans. Acoust., Speech and Signal Processing, vol. ASSP-32, no. 6, pp. 1243-1245, Dec. 1984.

6. N.I. Cho and S.U. Lee. Fast Algorithm and Implementation of 2-D DCT. IEEE Trans. Circuits and Systems, vol. 38, no. 3, pp. 297-305, Mar 1991.

7. E. Feig and S. Winograd. Fast Algorithms for the Discrete Cosine Transform, IEEE Trans. Signal Processing, vol. 40, no. 9, pp. 2174-2193, Sep. 1992.

8. C. Loeffler, A. Ligtenberg, and G.S. Moschytz. Practical fast 1-D DCT algorithms with 11 multiplication's. Proc. ICASSP 1989, pp. 988-991, 1989.

9. Y. Arai, T. Agui, and M. Nakajima. A fast DCT-SQ scheme for images. Trans. IEICE, vol. E-71, no. 11, pp. 1095-1097, Nov. 1988.

10. G. Wallace. The JPEG still picture compression standard. Communications of the ACM, vol. 34, no. 4, pp. 31-44, 1991.

11. D. LeGall. MPEG: A video compression standard for multimedia applications. Communications of the ACM, vol. 34, no. 4, pp. 46-58, 1991.

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

13. Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture. Intel Corp., 1997.

14. Intel Architecture Optimization Manual. Intel Corp. 1999. Order #730795.

15. Using MMX™ Instructions in a Fast iDCT Algorithm for MPEG Decoding. Application Note AP-528.

16. Using Streaming SIMD Extensions in a Fast DCT Algorithm for MPEG Encoding. Application Note AP-817.

Revision History

Revision	Revision History	Date
1.0	Original publication of the document	4/99

1 Introduction

The latest generations of 32-bit Intel processors support single instruction, multiple data (SIMD) data processing. Pentium® processors with MMX™ technology and later generations can execute integer SIMD instructions. In addition, Pentium III processors support single-precision floating-point SIMD instructions, as well as other new instructions, collectively referred to as *Streaming SIMD Extensions*.

This application note presents algorithms and code samples of a high-performance implementation of 8x8 Discrete Cosine Transform with precision satisfying IEEE standard 1180-1990. The new algorithm takes approximately 300 clock cycles per transform on processors with MMX™ technology or Pentium III processors.

2 Discrete Cosine Transform

Discrete Cosine Transform (DCT) is widely used in 1D and 2D signal processing. In particular, image processing applications often use the 8x8 2D DCT. There are many algorithms for the direct computation of the 8x8 2D DCT as well as algorithms for 8-element 1D DCTs, which you can use in the row-column method to effectively perform an 8x8 2D DCT [1-9].

2.1 Algorithmic optimization of DCT

Optimization usually focuses on reducing the number of DCT arithmetic operations, especially the number of multiplications. For example, Feig and Winograd proposed one of the fastest 2D 8x8 DCT algorithms [7]. The Loeffler-Ligtenberg-Moschytz (LLM) algorithm [8] and the Arai-Agui-Nakajima (AAN) algorithm [9] are among the fastest known 1D DCTs. Table 1 presents the characteristics of the above DCT algorithms.

Table 1: Characteristics of Existing DCT Algorithms

Algorithm	**Operations per 8-element 1D DCT**		**Operations per 8x8 2D DCT**	
	additions	***multiplications***	***additions***	***multiplications***
Feig & Winograd [7]	N/A (*direct 2D algorithm*)		454	94
LLM [8]	28	11	448	176
AAN [9]	29	5	464	144

All integer DCT implementations encounter serious precision problems. This application note is based on precision requirements for DCTs of 9-bit integer data, set forth in IEEE standard 1180-1990 [12]. Additional recommendations regarding the precision of scaling 2D DCTs (for both forward and inverse transforms) can be found in JPEG and MPEG specifications [10, 11].

2.2 Existing SIMD Implementations of DCT

On the latest Intel processors with the MMX technology [13] or Streaming SIMD extensions [14], DCT code using the new SIMD instructions can run much faster. For example, fast SIMD DCT implementations (based on the 1D AAN algorithm) are described in [15, 16]. However, these implementations are not compliant with IEEE standard 1180-1990.

In SIMD implementations of the DCT, non-arithmetic operations (copying and other auxiliary instructions) have more impact on the overall performance than in scalar implementations. For these reasons, SIMD code of a direct 2D DCT algorithms turns out to be less efficient than the row-column approach.

3 New Precise DCT Implementation Using SIMD Instructions

This application note presents a new fast algorithm for computing the 8x8 DCT with precision satisfying IEEE standard 1180-1990. The new algorithm processes 16-bit integer data and imposes the following restrictions on the input data range:

- input data for the forward DCT must have at most 9-bit values
- input data for the inverse DCT must have at most 12-bit values.

Although the number of *arithmetic* operations in the new algorithm is slightly more than that of [15, 16], its *overall performance* approximately the same as [15, 16], and precision is significantly better.

3.1 The 8x8 2D DCT: Formulas and Strategy

We used the forward 8x8 2D DCT defined by this equation:

$$f_{nm} = \frac{1}{4} c_n c_m \sum_{i=0}^{7} \sum_{j=0}^{7} \cos\frac{\pi n(2i+1)}{16} \cos\frac{\pi m(2j+1)}{16} x_{ij} \tag{1}$$

and the corresponding inverse 8x8 2D DCT, defined as follows:

$$x_{ij} = \frac{1}{4} \sum_{n=0}^{7} \sum_{m=0}^{7} \cos\frac{\pi n(2i+1)}{16} \cos\frac{\pi m(2j+1)}{16} c_n c_m f_{nm}\,, \tag{2}$$

where $c_0 = 1/\sqrt{2}$ and $c_n = 1$ for n = 1...7. (There exist alternative DCT definitions in which the output of forward DCTs is multiplied by certain coefficients; see [7] for more information.)

The new SIMD DCT implementation uses different computations for 1D transforms for rows and columns. The 1D transform for each row is computed directly, in 32-bit precision. On the other hand, each group of 4 columns is transformed in parallel using 16-bit precision (with additional scaling). Although the 32-bit transforms for rows are relatively slow, the above combination of 1D transforms for rows and columns allows us to *avoid the transposition overhead*.

The forward 2D DCT processes columns first, then rows; the inverse 2D DCT applies the 1D transform to rows first. The intermediate results are stored as 16-bit integer array; they are shifted to the left in order to reduce the subsequent computational errors. To increase the performance, we use a scaling algorithm in which intermediate results are multiplied by scaling coefficients during the 1D row transforms. Effectively, this does not require additional arithmetic operations.

3.2 The Underlying 1D DCT

The direct *N*-element 1D DCT is defined by the following equation:

$$y_n = c_n \sum_{k=0}^{N-1} \cos \frac{\pi n(2k+1)}{2N} x_k \tag{3}$$

and the corresponding inverse 1D DCT by this equation:

$$x_k = \sum_{n=0}^{N-1} \cos \frac{\pi n(2k+1)}{2N} c_n y_n\,, \tag{4}$$

where $c_0 = \frac{1}{\sqrt{N}}$ and $c_n = \sqrt{\frac{2}{N}}$ for $n = 1 \ldots N - 1$.

We use the same notation as in [7] and let $\gamma(k) = \cos(\pi k/16)$, which allows us to represent the matrix of 8-element 1D DCT as follows:

$$C_8 = \frac{1}{2}\begin{bmatrix} \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) \\ \gamma(1) & \gamma(3) & \gamma(5) & \gamma(7) & -\gamma(7) & -\gamma(5) & -\gamma(3) & -\gamma(1) \\ \gamma(2) & \gamma(6) & -\gamma(6) & -\gamma(2) & -\gamma(2) & -\gamma(6) & \gamma(6) & \gamma(2) \\ \gamma(3) & -\gamma(7) & -\gamma(1) & -\gamma(5) & \gamma(5) & \gamma(1) & \gamma(7) & -\gamma(3) \\ \gamma(4) & -\gamma(4) & -\gamma(4) & \gamma(4) & \gamma(4) & -\gamma(4) & -\gamma(4) & \gamma(4) \\ \gamma(5) & -\gamma(1) & \gamma(7) & \gamma(3) & -\gamma(3) & -\gamma(7) & \gamma(1) & \gamma(5) \\ \gamma(6) & -\gamma(2) & \gamma(2) & -\gamma(6) & -\gamma(6) & \gamma(2) & -\gamma(2) & \gamma(6) \\ \gamma(7) & -\gamma(5) & \gamma(3) & -\gamma(1) & \gamma(1) & -\gamma(3) & \gamma(5) & -\gamma(7) \end{bmatrix} \tag{5}$$

The operator C_8 is orthogonal, therefore we can determine the matrix of inverse 1D DCT by transposing the matrix of the forward 1D DCT: $C_8^{-1} = C_8^T$. (Here A^{-1} denotes the inverse of *A*, and A^T denotes the transpose of *A*.)

We now factorize the matrix C_8 as follows:

$$C_8 = \frac{1}{2} \cdot P_8 \cdot M_8 \cdot A_8 \tag{6}$$

where

$$A_8 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \end{bmatrix} \tag{7}$$

$$M_8 = \begin{bmatrix} \gamma(4) & \gamma(4) & \gamma(4) & \gamma(4) & & & & \\ \gamma(2) & \gamma(6) & -\gamma(6) & -\gamma(2) & & & & \\ \gamma(4) & -\gamma(4) & -\gamma(4) & \gamma(4) & & & & \\ \gamma(6) & -\gamma(2) & \gamma(2) & -\gamma(6) & & & & \\ & & & & \gamma(1) & \gamma(3) & \gamma(5) & \gamma(7) \\ & & & & \gamma(3) & -\gamma(7) & -\gamma(1) & -\gamma(5) \\ & & & & \gamma(5) & -\gamma(1) & \gamma(7) & \gamma(3) \\ & & & & \gamma(7) & -\gamma(5) & \gamma(3) & -\gamma(1) \end{bmatrix} \tag{8}$$

$$P_8 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

(White spaces in matrices denote zero entries.) The operator P_8 in (6) performs a permutation of elements in the data vector and does not require arithmetic operations. The symmetric properties of the operator A_8 also allow us to reduce the number of arithmetic operations. (All fast 1D DCTs use the factorization (6) or its analogs. Our new algorithm differs only in the subsequent representation of M_8.)

Since C_8 is orthogonal, the inverse DCT matrix is

$$C_8^{-1} = C_8^T = \frac{1}{2} \cdot A_8^T \cdot M_8^T \cdot P_8^T \qquad (10)$$

3.3 Computing the 8-element DCTs for rows

Our direct algorithm for computing forward and inverse DCTs for rows is based on factorizations (6) and (10). It performs all arithmetic operations involved in the operators M_8 and M_8^T – namely, 32 multiplications and 24 additions.

To compute the forward DCT, the algorithm uses the following operations:

Operation	MMX™ instructions used
Operator A_8 on 16-bit integers	***PADDSW*** and ***PSUBSW***
Operator M_8 (16-bit input, 32-bit results)	***PMADDWD*** and ***PADDD***
Rounding and packing the results to 16-bit packed word format.	***PADDD***, ***PSRAD*** and ***PACKSSDW***

To compute the inverse DCT, we use the operator M_8^T (similar to M_8, it produces 32-bit intermediate results). Then, the operator A_8^T (implemented using the MMX™ instructions ***PADDD*** and ***PSUBD***) acts on the 32-bit data. Finally, the 16-bit output data are obtained by rounding and packing, similar to the forward DCT.

The algorithm uses real signed constants $\gamma(k)$, as defined in the operator M_8; see (8). The constants are scaled as follows:

$$\lfloor \gamma(k) \cdot 2^{15} + 0.5 \rfloor \quad \text{or} \quad \lfloor -\gamma(k) \cdot 2^{15} + 0.5 \rfloor,$$

where $\lfloor x \rfloor$ denotes the integer part of *x*. The scaled constants stored in 16-bit integer format in an order suitable for applying the ***PMADDWD*** instruction (constants for the forward and inverse transform are stored in different order).

Note that the column DCTs (described in the next section) use a scaling algorithm, which requires additional multipliers to be used during the row DCTs. Effectively, the operators M_8 and M_8^T are multiplied by a constant which depends on the row number. By properly modifying the tabulated constants once, we avoid additional arithmetic operations in the row DCT.

In addition to arithmetic operations, DCT requires data reordering. This is because the “natural” data storage order differs from the optimal order for computing the operators A_8 and A_8^T. For data reordering, our algorithm uses the MMX instructions ***PUNPCKLWD*** and ***PUNPCKHWD*** or, on Pentium® III processors, the ***PSHUFW*** instructions.

The auxiliary data reordering operations are the only part of the algorithm that uses new instructions of Pentium III processors (if available).

All the above operations are performed for each row in the data matrix. The total numbers of arithmetic operations are 256 additions and 256 multiplications; these numbers do not include auxiliary operations.

The forward DCT for rows takes 96 MMX instructions; the inverse transform takes 128 MMX instructions for arithmetic operations. In addition, it takes approximately 240 instructions to perform auxiliary operations.

3.4 Computing the 8-element DCTs for columns

To compute the 8-element DCTs for columns, we modified an algorithm published in [2] based on original works [3,4]. Originally, the algorithm [2] used 26 multiplications and 16 additions; we transformed it into a scaling algorithm which uses 26 additions and 8 multiplications. (Additional multiplications are now incorporated in the row DCT).

The DCT for columns uses the following factorization of the operator M_8:

$$M_8 = D_8 \cdot B_8 \cdot E_8 \cdot F_8 \qquad (11)$$

where

$$D_8 = \begin{bmatrix} \gamma(4) & 0 & 0 & 0 & & & & \\ 0 & 0 & \gamma(2) & 0 & & & & \\ 0 & \gamma(4) & 0 & 0 & & & & \\ 0 & 0 & 0 & \gamma(2) & & & & \\ & & & & \gamma(1) & 0 & 0 & 0 \\ & & & & 0 & 0 & \gamma(3) & 0 \\ & & & & 0 & 0 & 0 & \gamma(3) \\ & & & & 0 & \gamma(1) & 0 & 0 \end{bmatrix} \qquad (12)$$

$$B_8 = \begin{bmatrix} 1 & 1 & & & & & & \\ 1 & -1 & & & & & & \\ & & 1 & \frac{\gamma(6)}{\gamma(2)} & & & & \\ & & \frac{\gamma(6)}{\gamma(2)} & -1 & & & & \\ & & & & 1 & \frac{\gamma(7)}{\gamma(1)} & & \\ & & & & \frac{\gamma(7)}{\gamma(1)} & -1 & & \\ & & & & & & 1 & \frac{\gamma(5)}{\gamma(3)} \\ & & & & & & \frac{\gamma(5)}{\gamma(3)} & -1 \end{bmatrix} \tag{13}$$

$$E_8 = \begin{bmatrix} 1 & 0 & 0 & 1 & & & & \\ 0 & 1 & 1 & 0 & & & & \\ 1 & 0 & 0 & -1 & & & & \\ 0 & 1 & -1 & 0 & & & & \\ & & & & 1 & 1 & 0 & 0 \\ & & & & 0 & 0 & 1 & 1 \\ & & & & 1 & -1 & 0 & 0 \\ & & & & 0 & 0 & 1 & -1 \end{bmatrix} \tag{14}$$

$$F_8 = \begin{bmatrix} 1 & 0 & 0 & 0 & & & & \\ 0 & 1 & 0 & 0 & & & & \\ 0 & 0 & 1 & 0 & & & & \\ 0 & 0 & 0 & 1 & & & & \\ & & & & 1 & 0 & 0 & 0 \\ & & & & 0 & \lambda(4) & \lambda(4) & 0 \\ & & & & 0 & \lambda(4) & -\lambda(4) & 0 \\ & & & & 0 & 0 & 0 & 1 \end{bmatrix} \tag{15}$$

In the original DCT algorithm, all non-zero elements in the matrix D_8 are 1, and all the multiplications by scaling constants are incorporated in the operator B_8. In our algorithm, on the other hand, the 1D column DCT stage performs only computations corresponding to the operators B_8, E_8 and F_8. Multiplications corresponding to D_8 are actually performed during the row-by-row DCT stage. The following multipliers are used:

- rows 0 and 4 are multiplied by $\gamma(4)$,
- rows 1 and 7 are multiplied by $\gamma(1)$,
- rows 2 and 6 are multiplied by $\gamma(2)$,
- rows 3 and 5 are multiplied by $\gamma(3)$.

The new constants (fractions) in the operator B_8 can be interpreted as tangents of the corresponding angles, for example:

$$\frac{\gamma(5)}{\gamma(3)} = \frac{\cos(5\pi/16)}{\cos(3\pi/16)} = \tan(3\pi/16)$$

The inverse 1D DCT operator M_8^T is factorized as

$$M_8^T = F_8^T \cdot E_8^T \cdot B_8^T \cdot D_8^T \qquad (16)$$

All computations in (11) and (16) are performed with 16-bit precision by using MMX instructions. The algorithm uses scaled real coefficients:

$$ICONST = \lfloor const \cdot 2^{16} + 0.5 \rfloor \qquad (17)$$

they are stored as 16-bit *integers*.

To perform multiplications, we use the MMX™ instruction ***PMULHW*** which computes the higher 16 bits of the signed product. If a real constant *const* is greater than 0.5, the corresponding *ICONST* after scaling (17) would contain 1 in the sign bit, which means ICONST would be a negative integer. To compute products $const \cdot x$ correctly, we use the following expression:

$$const \cdot x = PMULHW(ICONST, x) + x$$

For better accuracy of the forward DCT, the algorithm shifts the input data bits to the left. During the row-by-row transform, the intermediate results are shifted to the right. The data for inverse column DCT are already shifted to the left during the row transform. The results are rounded and shifted back to the right.

The DCT of 8 columns requires 208 additions and 64 multiplications. The algorithm processes 4 columns in parallel. The whole DCT takes 70 MMX instructions for arithmetic operations. In addition, auxiliary operations take 80-100 instructions.

4 Accuracy: Compliance with IEEE Standard 1180-1990

The IEEE Standard 1180-1990 [12] sets accuracy requirements for the 8x8 inverse DCT in terms of permissible differences between the actual transform results and the results computed in double-precision floating-point arithmetic:

- mean absolute difference
- standard deviation (or root-mean-square difference)
- other integral characteristics of differences for a large number of input data matrices.

Compliance tests are performed for a large input data set generated with a random number generator (which is also specified in the standard).

Our original implementation of the inverse DCT already had relatively high accuracy; however, it did not comply with the IEEE standard yet. The mean error is 0.004817, whereas the maximum permissible mean error is 0.0015 (see Table 2).

Table 2: Errors in the original DCT implementation for sample data

-0.0078	-0.0094	-0.0083	-0.0090	-0.0096	-0.0084	-0.0095	-0.0092
-0.0112	-0.0141	-0.0104	-0.0117	-0.0105	-0.0135	-0.0126	-0.0122
0.0113	0.0105	0.0115	0.0101	0.0103	0.0123	0.0139	0.0115
0.0079	0.0100	0.0095	0.0093	0.0105	0.0107	0.0084	0.0097
0.0082	0.0118	0.0108	0.0100	0.0090	0.0100	0.0086	0.0118
0.0079	0.0083	0.0079	0.0071	0.0079	0.0075	0.0089	0.0061
0.0128	0.0108	0.0108	0.0113	0.0094	0.0104	0.0121	0.0116
0.0101	0.0103	0.0091	0.0104	0.0098	0.0084	0.0084	0.0108

As seen from Table 2, the computed transform results are biased (the computed results in the first two rows are less than the corresponding exact results; the computed results in the other rows are greater than the corresponding exact results). This bias is because the column transforms used the MMX instruction ***PMULHW***, which effectively computed all products

$$x \cdot const \tag{18}$$

by truncating the scaled products as follows:

$$\left\lfloor \frac{x \cdot ICONST}{2^{16}} \right\rfloor \tag{19}$$

where $ICONST$ denotes the integer constant defined in (17). Therefore, in the original implementation of the algorithm the product (18) was actually replaced by $\lfloor x \cdot const \rfloor$.

This is a biased result; in most cases it is less than the exact result. This appears to be the main cause for errors in the original version of the algorithm.

To reduce the errors in the computed product (18), we could simply use the following formula with round-off correction:

$$\lfloor x \cdot const + 0.5 \rfloor,$$

However, the direct implementation of this formula with MMX instructions is inefficient. Below we describe a correction of the computed results in the improved version of our algorithm.

Suppose that we use ***PMULHW*** instructions to compute a sum of two products

$$A \cdot x + B \cdot y .$$

Without corrections, the actual computed result would be

$$\lfloor A \cdot x \rfloor + \lfloor B \cdot y \rfloor.$$

The natural correction for the above sum is the following:

$$\lfloor A \cdot x + B \cdot y + 0.5 \rfloor.$$

If the input data x and y have independent uniform distributions, then

$$\lfloor A \cdot x + B \cdot y + 0.5 \rfloor = \lfloor A \cdot x \rfloor + \lfloor B \cdot y \rfloor \qquad \text{with probability 1/8,}$$

$$\lfloor A \cdot x + B \cdot y + 0.5 \rfloor = \lfloor A \cdot x \rfloor + \lfloor B \cdot y \rfloor + 1 \qquad \text{with probability 3/4,}$$

$$\lfloor A \cdot x + B \cdot y + 0.5 \rfloor = \lfloor A \cdot x \rfloor + \lfloor B \cdot y \rfloor + 2 \qquad \text{with probability 1/8.}$$

If we just add 1 to computed sums for all data x and y:

$$\lfloor A \cdot x \rfloor + \lfloor B \cdot y \rfloor + 1 ,$$

this would considerably reduce the mean error of the sums.

Another correction technique is to increase the computed product (19) by setting the lowest bit of the product to 1.

We found an optimal combination of the above techniques by computational experiment. As a result, our algorithm met and even exceeded the precision requirements of IEEE standard 1180-1990. For the same sample data as in Table 2, our improved algorithm computed the DCT with errors shown in Table 3.

Table 3: Errors in the improved DCT implementation

0.0016	-0.0007	0.0008	-0.0004	0.0005	0.0014	0.0016	0.0006
-0.0004	-0.0014	0.0007	-0.0008	0.0019	0.0010	0.0007	-0.0001
-0.0020	0.0001	-0.0003	-0.0007	-0.0011	-0.0007	0.0002	-0.0004
-0.0020	-0.0001	-0.0006	-0.0002	0.0008	-0.0003	-0.0008	0.0002
-0.0003	0.0012	-0.0001	0.0007	0.0002	-0.0001	-0.0005	0.0010
0.0008	0.0013	0.0011	-0.0015	-0.0002	-0.0006	0.0005	-0.0005
-0.0008	-0.0010	0.0001	0.0014	-0.0001	-0.0001	0.0003	0.0002
-0.0005	0.0001	0.0014	0.0019	-0.0008	0.0001	-0.0013	-0.0005

Now the mean error is 0.00039, which is much less than the permissible mean error 0.0015 specified in the standard.

5 Performance

The performance characteristics of the new 8x8 2D DCT algorithm for 16-bit signed integer data are shown in Table 4. The algorithm's implementation with MMX instructions and Streaming SIMD Extensions is 3 to 3.5 times faster than the implementation that does not use SIMD instructions of either type.

Table 4: 8x8 2D DCT Performance (in clock cycles)

Transform type	Integer implementation	MMX™ instructions	MMX™ instructions and Streaming SIMD Extensions
2D DCT 8x8	970	280	250
2D iDCT 8x8	970	320	290

6 Comparison to Other Fast DCT Algorithms

Other existing fast DCT algorithms optimized for the latest generations of Intel processors are based on the 1D AAN algorithm and take about 250 clock cycles per 8x8 2D DCT [15,16]. However, the results of transform algorithms [15, 16] are presented in *transposed* format. Note also that if the user needs an inverse DCT for the algorithms in [15, 16], this would require auxiliary operations on top of the 250 cycles per transform. These auxiliary operations include shifting the inverse DCT input data to the left, as well as scaling the results of the forward DCT. An estimated overhead for these auxiliary operations in transforms defined in (1) and (2) is 50 to 70 clocks.

Thus, the algorithm presented in this application note has approximately the same performance as those in [15, 16]. At the same time, the new algorithm is compliant with the precision requirements of IEEE standard 1180-1990.

Note that on older processors which do not support SIMD instructions the new algorithm is inferior to existing DCT algorithms: the number of additions is 464, and the number of multiplications is 320 per transform.

7 Code Examples

```
;=============================================================================
;
; These examples contain code fragments for first stage iDCT 8x8
; (for rows) and first stage DCT 8x8 (for columns)
;
;=============================================================================

mword    typedef  qword
mptr     equ      mword ptr

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

BITS_FRW_ACC  = 3                               ; 2 or 3 for accuracy
SHIFT_FRW_COL = BITS_FRW_ACC
SHIFT_FRW_ROW = BITS_FRW_ACC + 17
RND_FRW_ROW   = 262144 * (BITS_FRW_ACC - 1)     ; 1 << (SHIFT_FRW_ROW-1)

               .data
               Align 16

one_corr       sword  1,             1,             1,             1
round_inv_row  dword  RND_INV_ROW,                  RND_INV_ROW
round_inv_col  sword  RND_INV_COL,   RND_INV_COL,   RND_INV_COL,   RND_INV_COL
round_inv_corr sword  RND_INV_CORR,  RND_INV_CORR,  RND_INV_CORR,  RND_INV_CORR
round_frw_row  dword  RND_FRW_ROW,                  RND_FRW_ROW

  tg_1_16      sword   13036,  13036,  13036,  13036   ;  tg * (2<<16) + 0.5
  tg_2_16      sword   27146,  27146,  27146,  27146   ;  tg * (2<<16) + 0.5
  tg_3_16      sword  -21746, -21746, -21746, -21746   ;  tg * (2<<16) + 0.5
 cos_4_16      sword  -19195, -19195, -19195, -19195   ; cos * (2<<16) + 0.5
ocos_4_16      sword   23170,  23170,  23170,  23170   ; cos * (2<<15) + 0.5

;=============================================================================
;
; The first stage iDCT 8x8 - 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 - by cos_4_16,
;    for rows 1,7 - by cos_1_16,
;    for rows 2,6 - by cos_2_16,
;    for rows 3,5 - by cos_3_16
; and are shifted to the left for better accuracy
;
; For the constants used,
;    FIX(float_const) = (short) (float_const * (1<<15) + 0.5)
;
;=============================================================================

;=============================================================================
IF _MMX                                                 ; MMX code
;=============================================================================

; Table for rows 0,4 - constants are multiplied by cos_4_16

tab_i_04 sword   16384,  16384,  16384, -16384   ; movq-> w06 w04 w02 w00
         sword   21407,   8867,   8867, -21407   ;        w07 w05 w03 w01
         sword   16384, -16384,  16384,  16384   ;        w14 w12 w10 w08
         sword   -8867,  21407, -21407,  -8867   ;        w15 w13 w11 w09
         sword   22725,  12873,  19266, -22725   ;        w22 w20 w18 w16
         sword   19266,   4520,  -4520, -12873   ;        w23 w21 w19 w17
         sword   12873,   4520,   4520,  19266   ;        w30 w28 w26 w24
         sword  -22725,  19266, -12873, -22725   ;        w31 w29 w27 w25

; Table for rows 1,7 - constants are multiplied by cos_1_16

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

```
; Table for rows 2,6 - constants are multiplied by cos_2_16

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

; Table for rows 3,5 - constants are multiplied by cos_3_16

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

;-----------------------------------------------------------------------------

DCT_8_INV_ROW_1 MACRO INP:REQ, OUT:REQ, TABLE:REQ

         movq       mm0, mptr [INP]        ; 0   ; x3 x2 x1 x0

         movq       mm1, mptr [INP+8]      ; 1   ; x7 x6 x5 x4
         movq       mm2, mm0               ; 2   ; x3 x2 x1 x0

         movq       mm3, mptr [TABLE]      ; 3   ; w06 w04 w02 w00
         punpcklwd  mm0, mm1                     ; x5 x1 x4 x0

         movq       mm5, mm0               ; 5   ; x5 x1 x4 x0
         punpckldq  mm0, mm0                     ; x4 x0 x4 x0

         movq       mm4, mptr [TABLE+8]    ; 4   ; w07 w05 w03 w01
         punpckhwd  mm2, mm1               ;   1 ; x7 x3 x6 x2

         pmaddwd    mm3, mm0                     ; x4*w06+x0*w04 x4*w02+x0*w00
         movq       mm6, mm2               ; 6   ; x7 x3 x6 x2

         movq       mm1, mptr [TABLE+32]   ; 1   ; w22 w20 w18 w16
         punpckldq  mm2, mm2                     ; x6 x2 x6 x2

         pmaddwd    mm4, mm2                     ; x6*w07+x2*w05 x6*w03+x2*w01
         punpckhdq  mm5, mm5                     ; x5 x1 x5 x1

         pmaddwd    mm0, mptr [TABLE+16]         ; x4*w14+x0*w12 x4*w10+x0*w08
         punpckhdq  mm6, mm6                     ; x7 x3 x7 x3

         movq       mm7, mptr [TABLE+40]   ; 7   ; w23 w21 w19 w17
         pmaddwd    mm1, mm5                     ; x5*w22+x1*w20 x5*w18+x1*w16

         paddd      mm3, mptr round_inv_row      ; +rounder
         pmaddwd    mm7, mm6                     ; x7*w23+x3*w21 x7*w19+x3*w17

         pmaddwd    mm2, mptr [TABLE+24]         ; x6*w15+x2*w13 x6*w11+x2*w09
         paddd      mm3, mm4               ;   4 ; a1=sum(even1) a0=sum(even0)
```

```
        pmaddwd    mm5, mptr [TABLE+48]          ; x5*w30+x1*w28 x5*w26+x1*w24
        movq       mm4, mm3                ; 4   ; a1 a0

        pmaddwd    mm6, mptr [TABLE+56]          ; x7*w31+x3*w29 x7*w27+x3*w25
        paddd      mm1, mm7                ;   7 ; b1=sum(odd1) b0=sum(odd0)

        paddd      mm0, mptr round_inv_row       ; +rounder
        psubd      mm3, mm1                      ; a1-b1 a0-b0

        psrad      mm3, SHIFT_INV_ROW            ; y6=a1-b1 y7=a0-b0
        paddd      mm1, mm4                ;   4 ; a1+b1 a0+b0

        paddd      mm0, mm2                ;   2 ; a3=sum(even3) a2=sum(even2)
        psrad      mm1, SHIFT_INV_ROW            ; y1=a1+b1 y0=a0+b0

        paddd      mm5, mm6                ;   6 ; b3=sum(odd3) b2=sum(odd2)
        movq       mm4, mm0                ; 4   ; a3 a2

        paddd      mm0, mm5                      ; a3+b3 a2+b2
        psubd      mm4, mm5                ;   5 ; a3-b3 a2-b2

        psrad      mm0, SHIFT_INV_ROW            ; y3=a3+b3 y2=a2+b2

        psrad      mm4, SHIFT_INV_ROW            ; y4=a3-b3 y5=a2-b2

        packssdw   mm1, mm0                ;   0 ; y3 y2 y1 y0

        packssdw   mm4, mm3                ;   3 ; y6 y7 y4 y5

        movq       mm7, mm4                ; 7   ; y6 y7 y4 y5
        psrld      mm4, 16                       ;  0 y6  0 y4

        pslld      mm7, 16                       ; y7  0 y5  0

        movq       mptr [OUT], mm1         ;   1 ; save y3 y2 y1 y0
        por        mm7, mm4                ;   4 ; y7 y6 y5 y4

        movq       mptr [OUT+8], mm7       ;   7 ; save y7 y6 y5 y4

ENDM

;==============================================================================
ELSE                                             ; code for Pentium III
;==============================================================================

; Table for rows 0,4 - constants are multiplied by cos_4_16

tab_i_04 sword   16384,  21407,  16384,   8867   ; movq-> w05 w04 w01 w00
         sword   16384,   8867, -16384, -21407   ;        w07 w06 w03 w02
         sword   16384,  -8867,  16384, -21407   ;        w13 w12 w09 w08
         sword  -16384,  21407,  16384,  -8867   ;        w15 w14 w11 w10
         sword   22725,  19266,  19266,  -4520   ;        w21 w20 w17 w16
         sword   12873,   4520, -22725, -12873   ;        w23 w22 w19 w18
         sword   12873, -22725,   4520, -12873   ;        w29 w28 w25 w24
         sword    4520,  19266,  19266, -22725   ;        w31 w30 w27 w26

; Table for rows 1,7 - constants are multiplied by cos_1_16

tab_i_17 sword   22725,  29692,  22725,  12299   ; movq-> w05 w04 w01 w00
         sword   22725,  12299, -22725, -29692   ;        w07 w06 w03 w02
```

```
        sword   22725, -12299,  22725, -29692   ;        w13 w12 w09 w08
        sword  -22725,  29692,  22725, -12299   ;        w15 w14 w11 w10
        sword   31521,  26722,  26722,  -6270   ;        w21 w20 w17 w16
        sword   17855,   6270, -31521, -17855   ;        w23 w22 w19 w18
        sword   17855, -31521,   6270, -17855   ;        w29 w28 w25 w24
        sword    6270,  26722,  26722, -31521   ;        w31 w30 w27 w26

; Table for rows 2,6 - constants are multiplied by cos_2_16

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

; Table for rows 3,5 - constants are multiplied by cos_3_16

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

;-----------------------------------------------------------------------------

DCT_8_INV_ROW_1 MACRO INP:REQ, OUT:REQ, TABLE:REQ

        movq       mm0, mptr [INP]        ; 0   ; x3 x2 x1 x0

        movq       mm1, mptr [INP+8]      ; 1   ; x7 x6 x5 x4
        movq       mm2, mm0               ; 2   ; x3 x2 x1 x0

        movq       mm3, mptr [TABLE]      ; 3   ; w05 w04 w01 w00
        pshufw     mm0, mm0, 10001000b          ; x2 x0 x2 x0

        movq       mm4, mptr [TABLE+8]    ; 4   ; w07 w06 w03 w02
        movq       mm5, mm1               ; 5   ; x7 x6 x5 x4
        pmaddwd    mm3, mm0                     ; x2*w05+x0*w04 x2*w01+x0*w00

        movq       mm6, mptr [TABLE+32]   ; 6   ; w21 w20 w17 w16
        pshufw     mm1, mm1, 10001000b          ; x6 x4 x6 x4
        pmaddwd    mm4, mm1                     ; x6*w07+x4*w06 x6*w03+x4*w02

        movq       mm7, mptr [TABLE+40]   ; 7   ; w23 w22 w19 w18
        pshufw     mm2, mm2, 11011101b          ; x3 x1 x3 x1
        pmaddwd    mm6, mm2                     ; x3*w21+x1*w20 x3*w17+x1*w16

        pshufw     mm5, mm5, 11011101b          ; x7 x5 x7 x5
        pmaddwd    mm7, mm5                     ; x7*w23+x5*w22 x7*w19+x5*w18

        paddd      mm3, mptr round_inv_row      ; +rounder

        pmaddwd    mm0, mptr [TABLE+16]         ; x2*w13+x0*w12 x2*w09+x0*w08
        paddd      mm3, mm4               ;   4 ; a1=sum(even1) a0=sum(even0)
```

```
        pmaddwd    mm1, mptr [TABLE+24]         ; x6*w15+x4*w14 x6*w11+x4*w10
        movq       mm4, mm3               ; 4   ; a1 a0

        pmaddwd    mm2, mptr [TABLE+48]         ; x3*w29+x1*w28 x3*w25+x1*w24
        paddd      mm6, mm7               ;   7 ; b1=sum(odd1) b0=sum(odd0)

        pmaddwd    mm5, mptr [TABLE+56]         ; x7*w31+x5*w30 x7*w27+x5*w26
        paddd      mm3, mm6                     ; a1+b1 a0+b0

        paddd      mm0, mptr round_inv_row      ; +rounder
        psrad      mm3, SHIFT_INV_ROW           ; y1=a1+b1 y0=a0+b0

        paddd      mm0, mm1               ;   1 ; a3=sum(even3) a2=sum(even2)
        psubd      mm4, mm6               ;   6 ; a1-b1 a0-b0

        movq       mm7, mm0               ; 7   ; a3 a2
        paddd      mm2, mm5               ;   5 ; b3=sum(odd3) b2=sum(odd2)

        paddd      mm0, mm2                     ; a3+b3 a2+b2
        psrad      mm4, SHIFT_INV_ROW           ; y6=a1-b1 y7=a0-b0

        psubd      mm7, mm2               ;   2 ; a3-b3 a2-b2
        psrad      mm0, SHIFT_INV_ROW           ; y3=a3+b3 y2=a2+b2

        psrad      mm7, SHIFT_INV_ROW           ; y4=a3-b3 y5=a2-b2

        packssdw   mm3, mm0               ;   0 ; y3 y2 y1 y0

        packssdw   mm7, mm4               ;   4 ; y6 y7 y4 y5

        movq       mptr [OUT], mm3        ;   3 ; save y3 y2 y1 y0
        pshufw     mm7, mm7, 10110001b          ; y7 y6 y5 y4

        movq       mptr [OUT+8], mm7      ;   7 ; save y7 y6 y5 y4

ENDM

;=============================================================================
ENDIF                                           ;
;=============================================================================

;=============================================================================
;
; The first stage DCT 8x8 - forward DCTs of columns
;
; The outputs 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 forward DCT algorithm (26a8m)
;
;-----------------------------------------------------------------------------
;
; #define DCT_8_FRW_COL(x, y)
; {
;    short t0, t1, t2, t3, t4, t5, t6, t7;
```

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

DCT_8_FRW_COL_4 MACRO INP:REQ, OUT:REQ

LOCAL     x0, x1, x2, x3, x4, x5, x6, x7
LOCAL     y0, y1, y2, y3, y4, y5, y6, y7

x0        equ        [INP + 0*16]
x1        equ        [INP + 1*16]
x2        equ        [INP + 2*16]
x3        equ        [INP + 3*16]
x4        equ        [INP + 4*16]
x5        equ        [INP + 5*16]
x6        equ        [INP + 6*16]
x7        equ        [INP + 7*16]

y0        equ        [OUT + 0*16]
y1        equ        [OUT + 1*16]
y2        equ        [OUT + 2*16]
y3        equ        [OUT + 3*16]
y4        equ        [OUT + 4*16]
y5        equ        [OUT + 5*16]
y6        equ        [OUT + 6*16]
y7        equ        [OUT + 7*16]
```

```
movq      mm0, x1                ; 0   ; x1

movq      mm1, x6                ; 1   ; x6
movq      mm2, mm0               ; 2   ; x1

movq      mm3, x2                ; 3   ; x2
paddsw    mm0, mm1                     ; t1 = x[1] + x[6]

movq      mm4, x5                ; 4   ; x5
psllw     mm0, SHIFT_FRW_COL           ; t1

movq      mm5, x0                ; 5   ; x0
paddsw    mm4, mm3                     ; t2 = x[2] + x[5]

paddsw    mm5, x7                      ; t0 = x[0] + x[7]
psllw     mm4, SHIFT_FRW_COL           ; t2

movq      mm6, mm0               ; 6   ; t1
psubsw    mm2, mm1               ;   1 ; t6 = x[1] - x[6]

movq      mm1, mptr tg_2_16      ; 1   ; tg_2_16
psubsw    mm0, mm4                     ; tm12 = t1 - t2

movq      mm7, x3                ; 7   ; x3
pmulhw    mm1, mm0                     ; tm12*tg_2_16

paddsw    mm7, x4                      ; t3 = x[3] + x[4]
psllw     mm5, SHIFT_FRW_COL           ; t0

paddsw    mm6, mm4               ;   4 ; tp12 = t1 + t2
psllw     mm7, SHIFT_FRW_COL           ; t3

movq      mm4, mm5               ; 4   ; t0
psubsw    mm5, mm7                     ; tm03 = t0 - t3

paddsw    mm1, mm5                     ; y2 = tm03 + tm12*tg_2_16
paddsw    mm4, mm7               ;   7 ; tp03 = t0 + t3

por       mm1, mptr one_corr           ; correction y2 +0.5
psllw     mm2, SHIFT_FRW_COL+1         ; t6

pmulhw    mm5, mptr tg_2_16            ; tm03*tg_2_16
movq      mm7, mm4               ; 7   ; tp03

psubsw    mm3, x5                      ; t5 = x[2] - x[5]
psubsw    mm4, mm6                     ; y4 = tp03 - tp12

movq      y2,  mm1               ;   1 ; save y2
paddsw    mm7, mm6               ;   6 ; y0 = tp03 + tp12

movq      mm1, x3                ; 1   ; x3
psllw     mm3, SHIFT_FRW_COL+1         ; t5

psubsw    mm1, x4                      ; t4 = x[3] - x[4]
movq      mm6, mm2               ; 6   ; t6

movq      y4,  mm4               ;   4 ; save y4
paddsw    mm2, mm3                     ; t6 + t5

pmulhw    mm2, mptr ocos_4_16          ; tp65 = (t6 + t5)*cos_4_16
```

```
        psubsw     mm6, mm3                ;   3 ; t6 - t5

        pmulhw     mm6, mptr ocos_4_16           ; tm65 = (t6 - t5)*cos_4_16
        psubsw     mm5, mm0                ;   0 ; y6 = tm03*tg_2_16 - tm12

        por        mm5, mptr one_corr            ; correction y6 +0.5
        psllw      mm1, SHIFT_FRW_COL            ; t4

        por        mm2, mptr one_corr            ; correction tp65 +0.5
        movq       mm4, mm1                ; 4   ; t4

        movq       mm3, x0                 ; 3   ; x0
        paddsw     mm1, mm6                      ; tp465 = t4 + tm65

        psubsw     mm3, x7                       ; t7 = x[0] - x[7]
        psubsw     mm4, mm6                ;   6 ; tm465 = t4 - tm65

        movq       mm0, mptr tg_1_16       ; 0   ; tg_1_16
        psllw      mm3, SHIFT_FRW_COL            ; t7

        movq       mm6, mptr tg_3_16       ; 6   ; tg_3_16
        pmulhw     mm0, mm1                      ; tp465*tg_1_16

        movq       y0,  mm7                ;   7 ; save y0
        pmulhw     mm6, mm4                      ; tm465*tg_3_16

        movq       y6,  mm5                ;   5 ; save y6
        movq       mm7, mm3                ; 7   ; t7

        movq       mm5, mptr tg_3_16       ; 5   ; tg_3_16
        psubsw     mm7, mm2                      ; tm765 = t7 - tp65

        paddsw     mm3, mm2                ;   2 ; tp765 = t7 + tp65
        pmulhw     mm5, mm7                      ; tm765*tg_3_16

        paddsw     mm0, mm3                      ; y1 = tp765 + tp465*tg_1_16
        paddsw     mm6, mm4                      ; tm465*tg_3_16

        pmulhw     mm3, mptr tg_1_16             ; tp765*tg_1_16

        por        mm0, mptr one_corr            ; correction y1 +0.5
        paddsw     mm5, mm7                      ; tm765*tg_3_16

        psubsw     mm7, mm6                ;   6 ; y3 = tm765 - tm465*tg_3_16

        movq       y1,  mm0                ;   0 ; save y1
        paddsw     mm5, mm4                ;   4 ; y5 = tm765*tg_3_16 + tm465

        movq       y3,  mm7                ;   7 ; save y3
        psubsw     mm3, mm1                ;   1 ; y7 = tp765*tg_1_16 - tp465

        movq       y5,  mm5                ;   5 ; save y5

        movq       y7,  mm3                ;   3 ; save y7

ENDM
```