



Using MMX™ Instructions to Compute the 2x2 Haar Transform

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Note: Please be aware that the software programming article below is posted as a public service and may no longer be supported by Intel.

Copyright © Intel Corporation 2004

* Other names and brands may be claimed as the property of others.

Using MMX™ Instructions to Compute the 2x2 Haar Transform

March 1996

CONTENTS

1.0. INTRODUCTION

2.0. 2X2 HAAR TRANSFORM

2.1. Forward 2x2 Haar Transform Algorithm

2.2. Forward 2x2 Haar Core

2.3. Inverse 2x2 Haar Core

3.0. PERFORMANCE GAINS

3.1. Scalar Performance

3.2. MMX™ Code Performance

4.0. CODE LISTINGS

4.1. Forward 2x2 Haar Transform

4.2. Inverse 2x2 Haar Transform

1.0. INTRODUCTION

The Intel Architecture (IA) media extensions include single-instruction, multi-data (SIMD) instructions. This application note presents a code sample that computes the 2x2 Haar transform. The 2x2 Haar transform is computed by adding and subtracting adjacent image or array elements. MMX code can speed up calculations of the Haar transform significantly because MMX instructions permit eight 16-bit additions or subtractions in a single clock.

The forward transform code shows how changing the order of sums and differences permits single instruction multiple data (SIMD) operations without reordering data. Also, the forward transform code demonstrates use of the multiply-and-accumulate instruction, PMADDWD, which has a latency of three clocks even when one of the operands is in memory.

2.0. 2X2 HAAR TRANSFORM

The 2x2 Haar transform is used to decompose an image into four bands whose spatial frequencies and information contents differ. These differences allow sub-band compression methods to control the bit rate by quantizing bands differently and to control the decode time by removing one or more bands from the bit stream. The 2x2 Haar transform is identical to Daubechies' wavelet with a degree-2 scaling function. Therefore, this function is sometimes referred to as a wavelet.

The input of the forward transform code is the address of the input image buffer, the number of rows and columns, and the addressees of the output band buffers. The data size of the input image is one byte and the data size of the output bands is two bytes. The number of rows must be divisible by two and the number of columns must be divisible by eight.

The input of the inverse transform is the addresses of the four band buffers, the address of the reconstructed image, and the number of rows and columns of the reconstructed image. Like the forward transform, the data size of the bands is two bytes, and the data size of the reconstructed image is one byte, the number of rows must be divisible by two, and the number of columns must be divisible by eight.

2.1. Forward 2x2 Haar Transform Algorithm

The one-dimensional (1D) Haar transform replaces adjacent elements with their sums and differences. A two-dimensional (2D) transformation of an array executes the transform along rows followed by columns as shown in Example 1. Each of the four results of the 2D transformation of a block represents a different band. Example 1 shows bands following a 2D transformation. The implementation presented here stores the four bands in separate buffers.

Example 1. Forward 2D Haar Transformation with 1D Transform

P_0	P_1		Pixels of 2x2 image
P_2	P_3		
(P_0+P_1)	(P_0-P_1)		Block following 1D
(P_2+P_3)	(P_2-P_3)		transform along rows
$(P_0+P_1)+(P_2+P_3)$			
$(P_0-P_1)+(P_2-P_3)$			Block following 1D
$(P_0+P_1)-(P_2+P_3)$	$(P_0-P_1)-(P_2-P_3)$		transform along columns
Band ₀	Band ₂		Bands in array format
	Band ₁	Band ₃	

Example 1 shows how the 1D Haar function transforms a 2D array with two passes. A 2D array can be transformed in a single pass with operations that combine all four of the elements of a 2x2 block. The first set of equations in Example 2 shows how bands are computed using results of Example 1. However, sums and differences in this order do not permit SIMD operations without reordering the data. Therefore, the calculations are made in the order shown in the second set of equations in Example 2. These equations perform adjacent row operations before adjacent column operations, which is the more conventional method.

Example 2. Forward 2x2 Haar Transform Equations Which Permit SIMD Operations

Band0	=	$(P_0 + P_1) + (P_2 + P_3)$	Conventional order of
Band1	=	$(P_0 + P_1) - (P_2 + P_3)$	calculations for the
Band2	=	$(P_0 - P_1) + (P_2 - P_3)$	Haar transform
Band3	=	$(P_0 - P_1) - (P_2 - P_3)$	
Band0	=	$(P_0 + P_1) + (P_2 + P_3)$	Order of calculations
Band1	=	$(P_0 - P_1) + (P_2 - P_3)$	which permit SIMD
Band2	=	$(P_0 + P_1) - (P_2 + P_3)$	operations with MMX
Band3	=	$(P_0 - P_1) - (P_2 - P_3)$	instructions

Using MMX™ Instructions to Compute the 2x2 Haar Transform

March 1996

2.2. Forward 2x2 Haar Core

The core of the forward 2x2 Haar transform is listed in Example 3. Eight one-byte values from row_0 (four pairs of $P_0 P_1$) and row_1 (four pairs of $P_2 P_3$) are loaded in lines 1 and 3. The PUNPCKLBW and PUNPCKHBW instructions expand the data into 16-bit words. When instruction 10 completes, the first four values loaded from row_0 are in MMX register MM0, and the first four values loaded from row_1 are in MM1. These are data for two 2x2 blocks. Also, the second four values from row_0 are in MM4, and the second four values from row_1 are in MM5. These are data for two additional 2x2 blocks.

Four 2x2 blocks of data are decomposed into four bands in lines 11 through 33 using the second set of equations in Example 2. The sums and differences in parentheses are computed with the PADDW and PSUBW instructions. These instructions cannot be used for the final sum or difference because addends are in the same register, and subtrahends and minuends are in the same register. Therefore, the PMADDWD instruction is used to calculate these sums and differences. For example, when subtracting a 16-bit word from an adjacent 16-bit word in an MMX register, the PMADDWD instruction multiplies the minuend by 1 and the subtrahend by -1 and adds the products. The 32-bit results are reduced to 16-bit results with the PACKSSDW instruction. Results of the forward transform are stored with four 64-bit write MOVQ instructions. Each MOVQ writes results for four 2x2 blocks for a single band.

Example 3. Forward 2x2 Haar Transform Core

```
fwave:
1   movq    mm0,    [eax]           ;get row0
2   pxor   mm7,    mm7           ;0 in mm7
3   movq    mm1,    [ebx]         ;get row1
4   movq    mm4,    mm0           ;copy row0
5   punpcklbw mm0,    mm7         ;unpack low row0
6   movq    mm5,    mm1           ;copy row1
7   punpckhbw mm4,    mm7         ;unpack high row0 in mm4
8   movq    mm2,    mm0           ;copy unpacked row0
9   punpcklbw mm1,    mm7         ;unpack low row1
10  punpckhbw mm5,    mm7         ;unpack high row1
11  paddw   mm0,    mm1           ;row0 + row1
12  psubw   mm2,    mm1           ;row0 - row1
13  movq    mm1,    mm0           ;copy row0 + row1
14  pmaddwd mm0,    TAPS0         ;low b0
15  movq    mm3,    mm2           ;copy row0 - row1
16  pmaddwd mm2,    TAPS1         ;low b1
17  movq    mm6,    mm4           ;copy high row0
18  pmaddwd mm1,    TAPS2         ;low b2
19  paddw   mm4,    mm5           ;row0 + row1 high
20  pmaddwd mm3,    TAPS3         ;low b3
21  subw    mm6,    mm5           ;row0 - row1 high
22  movq    mm5,    mm4           ;copy row0 + row1 high
23  movq    mm7,    mm6           ;copy row0 - row1 high
24  pmaddwd mm4,    TAPS0         ;high b0
25  pmaddwd mm5,    TAPS2         ;high b2
26  add     eax,    8             ;increment row0 counter
27  add     ebx,    8             ;increment row1 address
28  pmaddwd mm6,    TAPS1         ;high b1
29  packssdw mm0,    mm4         ;pack low and high b0
30  add     ecx,    8             ;increment b0 address
31  add     edx,    8             ;increment b1 address
32  pmaddwd mm7,    TAPS3         ;high b3
33  packssdw mm1,    mm5         ;pack low and high b2
34  movq    [ecx],    mm0         ;store b0
35  packssdw mm2,    mm6         ;pack low and high b1
```

Using MMX™ Instructions to Compute the 2x2 Haar Transform

March 1996

```
36     add     edi,      8           ;inc b2 address early
37     add     esi,      8           ;inc b3 address early
38     movq    [edx],    mm2        ;store b1
39     packssdw mm3,    mm7         ;pack low and high b3
40     movq    [edi],    mm1        ;store b2
41     movq    [esi],    mm3        ;store b3
42     cmp     eax,      ILOOP      ;row0 addr = in loop lim?
43     jnz    jnz     fwave        ;jump if not end of row
```

2.3. Inverse 2x2 Haar Core

Equations for the inverse 2x2 Haar transform are given in Example 4. The core of the inverse 2x2 Haar transform code which implements these equations is listed in Example 5.

Example 4. Inverse 2x2 Haar Transform Equations

$$\begin{aligned}P_0 &= (\text{Band}_0 + \text{Band}_1) + (\text{Band}_2 + \text{Band}_3) / 4 \\P_1 &= (\text{Band}_0 - \text{Band}_1) + (\text{Band}_2 - \text{Band}_3) / 4 \\P_2 &= (\text{Band}_0 + \text{Band}_1) - (\text{Band}_2 + \text{Band}_3) / 4 \\P_3 &= (\text{Band}_0 - \text{Band}_1) - (\text{Band}_2 - \text{Band}_3) / 4\end{aligned}$$

The inverse transform uses band data to construct 2x2 blocks as described in Section 2.1. The code begins by loading band data. Each MOVQ instruction loads four 16-bit values from a single band. For this reason, unlike the forward transform, there are no calculations which involve data in the same register. In other words, all sums and differences are computed with PADDW and PSUBW instructions. The PSRAW shift-right instruction is used to normalize the results by dividing them by four. Results are clamped to 255, the maximum value for one byte, with the PACKUSWB instruction.

Although the output data size is one byte, the clamping operation moves all P_0 and P_2 results into the lower 32-bits of MM0 and MM3, respectively, and P_1 and P_3 results to the upper 32-bits of MM0 and MM3, respectively. Since P_0 must be interleaved with P_1 , and P_2 interleaved with P_3 the data is reordered. The reorder operation uses the PUNPCKHBW, PUNPCKLBW, and PADDW instructions. Finally, eight values of row_0 and eight values of row_1 are written with MOVQ instructions.

Example 5. Inverse 2x2 Haar Transform Cor

```
iwave:
1     movq    mm0,      [eax]       ;load 4 b0's
2     pxor    mm6,      mm6         ;0 in mm8
3     movq    mm1,      [ebx]       ;load 4 b1's
4     movq    mm4,      mm0         ;copy b0 into mm4
5     movq    mm2,      [ecx]       ;load 4 b2's
6     paddw   mm0,      mm1         ;b0 + b1 in
7     movq    mm3,      [edx]       ;load 4 b3's
8     movq    mm5,      mm2         ;copy b2 into mm5
9     psubw   mm4,      mm1         ;b0 - b1 in mm4
10    paddw   mm2,      mm3         ;b2 + b3 in mm2
11    movq    mm1,      mm0         ;b0 + b1 in mm1
12    psubw   mm5,      mm3         ;b2 - b3 in mm5
13    paddw   mm0,      mm2         ;p0 = (b0+b1) + (b2+b3)
14    movq    mm3,      mm4         ;b2 - b3 in mm3
15    psubw   mm1,      mm2         ;p1 = (b0+b1) - (b2+b3)
16    psraw   mm0,      2           ;p0/4
17    paddw   mm3,      mm5         ;p2 = (b0-b1) + (b2=b3)
18    psraw   mm1,      2           ;p1/4
19    psubw   mm4,      mm5         ;p3 = (b0-b1)-(b2-b3)
20    add     eax,      8           ;inc b0 index
21    psraw   mm3,      2           ;p2/4
22    add     ebx,      8           ;inc b1 index
```

Using MMX™ Instructions to Compute the 2x2 Haar Transform

March 1996

```
23     psraw           mm4,    2           ;p3/4
24     add            ecx,    8           ;inc b2 index
25     packuswb mm0,   mm1,           ;clamp p0 and p1
26     pxor           mm2,    mm2        ;0 in mm2
27     packuswb mm3,   mm4,           ;clamp p2 and p3
28     movq           mm1,    mm0        ;clamped p0 and p1 in mm1
29     punpcklbw      mm0,    mm2        ;reorder with b0 in low byte
30     movq           mm4,    mm3        ;clamped p2 and p3 into mm4
31     punpckhbw      mm2,    mm1        ;reorder with p1 in high byte
32     paddw          mm0,    mm2        ;reordered p0 and p1 in mm0
33     punpcklbw      mm3,    mm6        ;reorder with p2 in low byte
34     add            esi,    8           ;inc p2 and p3 index
35     punpckhbw      mm6,    mm4        ;reorder with p3 in high byte
36     movq           [edi],  mm0        ;store 8 bytes p0 and p1
37     paddw          mm3,    mm6        ;reordered p2 and p3 in mm2
38     add            edx,    8           ;inc b3 index
39     add            edi,    8           ;inc p0 and p1 index
40     movq           [esi],  mm3        ;store 8 bytes p2 and p3
41     sub            ebp,    8           ;row0 addr = in loop lim?
42     jnz            iwave             ;jump if not end of row
```

3.0. PERFORMANCE GAINS

This section describes the performance improvement compared with traditional IA scalar code executing one operation per instruction.

There is approximately a 200 percent performance gain for MMX code. Results presented here assume all data is in the L1 cache. Performance gains are reduced if there are cache misses.

3.1. Scalar Performance

The total number of instructions required to compute the forward 2x2 Haar transform is:

$$4x (\textit{read} + \textit{copy} + \textit{add} + \textit{subtract} + \textit{write})$$

Since each instruction can be paired, the total number of clocks for four pixels is 10, or 2.5 clocks per pixel. The inverse transform requires four additional shift instructions. If these shifts are paired with other instructions the total number of clocks for four pixels is 12, or 3.0 clocks per pixel.

3.2. MMX™ Code Performance

The MMX code version of the forward transform executes in 24 clocks. Since this version processes four 2x2 blocks, the number of clocks per pixel is 1.5, which is a performance gain of 1.7. The inverse transform executes in 22 clocks, or 1.4 clocks per pixel. The resulting performance gain is 2.2.

The performance gain can be attributed to the following:

- The MMX code version uses 64-bit registers rather than 32-bit registers.
- The MMX code version uses instructions that facilitate multiple operations with a single instruction.

4.0. CODE LISTINGS

The code listings in this section are for the forward and reverse 2x2 Haar transform.

4.1 Forward 2x2 Haar Transform

```
;Calling program prototype is:
;void fwavemmx(
;  char * input,
;    int nrows,
;    int ncols,
;    short int * out0,
;    short int * out1,
;    short int * out2,
;    short int * out3
;  )
;    TITLE fwavemmx
;    .486P
.model FLAT
PUBLIC _fwavemmx
DATA SEGMENT
    NCOL DD ?
    NCOLM2 DD ?
    NROW DD ?
    ILOOP DD ?
    OLOOP DD ?
    ASIZE DD ?
    ALIGN DD 8
    TAPS0X dw 1,1,1,1
    TAPS1X dw 1,1,1,1
    TAPS2X dw 1,-1,1,-1
    TAPS3X dw 1,-1,1,-1
    TAPS0 dd ?,?
    TAPS1 dd ?,?
    TAPS2 dd ?,?
    TAPS3 dd ?,?
DATA ENDS
TEXT SEGMENT
    inPtr$ = 8 ;input pointer
    rows$ = 12 ;number rows
    cols$ = 16 ;number columns
    out0Ptr$ = 20 ;b0 pointer
    out1Ptr$ = 24 ;b1 pointer
    out2Ptr$ = 28 ;b2 pointer
    out3Ptr$ = 32 ;b3 pointer
fwavemmx PROC NEAR
    push ebx
    mov ebp, esp
    push eax
    push ebx
    push ecx
    push edx
    push esi
    push edi
    lea eax, TAPS0X
    movq mm0, [eax]
    movq TAPS0, mm0
    lea eax, TAPS1X
    movq mm0, [eax]
    movq TAPS1, mm0
    lea eax, TAPS2X
```

Using MMX™ Instructions to Compute the 2x2 Haar Transform

March 1996

```

movq    mm0,    [eax]
movq    TAPS2,  mm0
lea     eax,    TAPS3X
movq    mm0,    [eax]
movq    TAPS3,  mm0
mov     eax,    _rows$[ebp]    ;number rows
mov     ebx,    _cols$[ebp]    ;number columns
mov     NROW,   eax            ;store number rows
mov     NCOL,   ebx            ;store number columns
mul     ebx     ;input array size
mov     ASIZE,  eax            ;store input array size
shl     ebx,    1              ;double number columns
mov     NCOLM2, ebx            ;store 2Xncol
mov     eax,    _inPtr$[ebp]   ;pointer to input array
mov     ebx,    eax            ;copy input pointer
add     ebx,    NCOL            ;address next row
mov     ILOOP,  ebx            ;addr next row=end inner loop
mov     ecx,    eax            ;copy addr first row
add     ecx,    ASIZE           ;make end outer loop
mov     OLOOP,  ecx            ;store end outer loop
mov     ecx,    _out0Ptr$[ebp] ;pointer b0 out
mov     edx,    _out1Ptr$[ebp] ;pointer b1 out
mov     edi,    _out2Ptr$[ebp] ;pointer b2 out
mov     esi,    _out3Ptr$[ebp] ;pointer b3 out
sub     ecx,    8
sub     edx,    8
sub     edi,    8
sub     esi,    8

fwave:
movq    mm0,    [eax]          ;get row0
pxor    mm7,    mm7           ;0 in mm7
movq    mm1,    [ebx]          ;get row1
movq    mm4,    mm0           ;copy row0
punpcklbw mm0,    mm7          ;unpack low row0
movq    mm5,    mm1           ;copy row1
punpckhbw mm4,    mm7          ;unpack high row0 in mm4
movq    mm2,    mm0           ;copy unpacked row0
punpcklbw mm1,    mm7          ;unpack low row1
punpckhbw mm5,    mm7          ;unpack high row1
paddw   mm0,    mm1           ;row0 + row1
psubw   mm2,    mm1           ;row0 - row1
movq    mm1,    mm0           ;copy row0 + row1
pmaddwd mm0,    TAPS0         ;low b0
movq    mm3,    mm2           ;copy row0 - row1
pmaddwd mm2,    TAPS1         ;low b1
movq    mm6,    mm4           ;copy high row0
pmaddwd mm1,    TAPS2         ;low b2
paddw   mm4,    mm5           ;row0 + row1 high
pmaddwd mm3,    TAPS3         ;low b3
psubw   mm6,    mm5           ;row0 - row1 high
movq    mm5,    mm4           ;copy row0 + row1 high
movq    mm7,    mm6           ;copy row0 - row1 high
pmaddwd mm4,    TAPS0         ;high b0
pmaddwd mm5,    TAPS2         ;high b2
add     eax,    8              ;increment row0 counter
add     ebx,    8              ;increment row1 address
pmaddwd mm6,    TAPS1         ;high b1
packssdw mm0,    mm4          ;pack low and high b0
add     ecx,    8              ;increment b0 address
add     edx,    8              ;increment b1 address
pmaddwd mm7,    TAPS3         ;high b3
packssdw mm1,    mm5          ;pack low and high b2
movq    [ecx], mm0            ;store b0
packssdw mm2,    mm6          ;pack low and high b1

```

Using MMX™ Instructions to Compute the 2x2 Haar Transform

March 1996

```

        add         edi,      8           ;inc b2 address early
        add         esi,      8           ;inc b3 address early
        movq        [edx],    mm2         ;store b1
        packssdw   mm3,      mm7         ;pack low and high b3
        movq        [edi],    mm1         ;store b2
        movq        [esi],    mm3         ;store b3
        cmp         eax,      ILOOP       ;row0 addr = in loop lim?
        jnz        fwave        ;jump if not end of row
        add         eax,      NCOL        ;new row inc row0 addr
        add         ebx,      NCOL        ;new row inc row1 addr
        mov         ebp,      ILOOP       ;old in loop lim
        add         ebp,      NCOLM2      ;new in loop lim
        mov         ILOOP,    ebp         ;store new in loop lim
        cmp         eax,      OLOOP       ;row0 addr = out loop lim?
        jnz        fwave
        pop         edi
        pop         esi
        pop         edx
        pop         ecx
        pop         ebx
        pop         eax
        pop         ebp
        ret         0

fwavemmx ENDP
_TEXT ENDS
END
```

4.2. Inverse 2x2 Haar Transform

```

;Inverse Haar wavelet transform for 2x2 block for composition
;Code reads four streams of 16-bit words and writes two quad-words,
;each with 8 byte values for two adjacent rows
;Calling program prototype is:
;void iwaveasm(
;    char * out,
;    int nrows,
;    int ncols,
;    short int * in0,
;    short int * in1,
;    short int * in2,
;    short int * in3
;)
        TITLE
        .486P

.model FLAT
PUBLIC _iwavemmx
DATA SEGMENT
        NCOL      DD      ?
        NROW      DD      ?
        OLOOP     DD      ?
        ASIZE     DD
        L2CNT     DD      ?
DATA ENDS
TEXT SEGMENT
        outPtr$   = 8
        rows$     = 12
        cols$     = 16
        in0Ptr$   = 20
        in1Ptr$   = 24
        in2Ptr$   = 28
        in3Ptr$   = 32
_iwavemmx PROC NEAR
        push     ebp
        mov      ebp, esp
```

Using MMX™ Instructions to Compute the 2x2 Haar Transform

March 1996

```

    push    eax
    push    ebx
    push    ecx
    push    edx
    push    esi
    push    edi
    mov     eax,    _rows$[ebp]    ;number rows
    mov     ebx,    _cols$[ebp]    ;number columns
    mov     NROW,   eax            ;store number rows
    mov     NCOL,   ebx            ;store number columns
    mul     ebx                ;output array size
    mov     ASIZE,  eax            ;store output array size
    shl    eax,1                ;L2 count is twice row*col
    mov     L2CNT,  eax            ;factor 2 bytes in short int
    mov     eax,    _in0Ptr$[ebp]   ;pointer b0 out
    mov     ebx,    _in1Ptr$[ebp]   ;pointer b1 out
    mov     ecx,    _in2Ptr$[ebp]   ;pointer b2 out
    mov     edx,    _in3Ptr$[ebp]   ;pointer b3 out
    mov     edi,    _outPtr$[ebp]   ;pointer to input array
    mov     esi,    edi            ;copy input pointer
    add    esi,    NCOL            ;address next row
    mov     ebp,    edi            ;copy addr first row
    add    ebp,    ASIZE            ;make end outer loop
    mov     OLOOP,  ebp            ;store end outer loop
    mov     ebp,    NCOL            ;inner loop counter
    sub    esi,    8                ;adjust row1 pointer

iwave:
    movq    mm0,    [eax]            ;load 4 b0's
    pxor    mm6,    mm6                ;0 in mm8
    movq    mm1,    [ebx]            ;load 4 b1's
    movq    mm4,    mm0                ;copy b0 into mm4
    movq    mm2,    [ecx]            ;load 4 b2's
    paddw   mm0,    mm1                ;b0 + b1 in mm0
    movq    mm3,    [edx]            ;load 4 b3's
    movq    mm5,    mm2                ;copy b2 into mm5
    psubw   mm4,    mm1                ;b0 - b1 in mm4
    paddw   mm2,    mm3                ;b2 + b3 in mm2
    movq    mm1,    mm0                ;b0 + b1 in mm1
    psubw   mm5,    mm3                ;b2 - b3 in mm5
    paddw   mm0,    mm2                ;p0 = (b0+b1) + (b2+b3)
    movq    mm3,    mm4                ;b2 - b3 in mm3
    psubw   mm1,    mm2                ;p1 = (b0+b1) - (b2+b3)
    psraw   mm0,    2                ;p0/4
    paddw   mm3,    mm5                ;p2 = (b0-b1) + (b2=b3)
    psraw   mm1,    2                ;p1/4
    psubw   mm4,    mm5                ;p3 = (b0-b1)-(b2-b3)
    add    eax,    8                ;inc b0 index
    psraw   mm3,    2                ;p2/4
    add    ebx,    8                ;inc b1 index
    psraw   mm4,    2                ;p3/4
    add    ecx,    8                ;inc b2 index
    packuswb mm0,    mm1                ;clamp p0 and p1
    pxor    mm2,    mm2                ;0 in mm2
    packuswb mm3,    mm4                ;clamp p2 and p3
    movq    mm1,    mm0                ;clamped p0 adn p1 in mm1
    punpcklbw mm0,    mm2                ;reorder with b0 in low byte
    movq    mm4,    mm3                ;clamped p2 and p3 into mm4
    punpckhbw mm2,    mm1                ;reorder with p1 in high byte
    paddw   mm0,    mm2                ;reordered p0 and p1 in mm0
    punpcklbw mm3,    mm6                ;reorder with p2 in low byte
    add    esi,    8                ;inc p2 and p3 index
    punpckhbw mm6,    mm4                ;reorder with p3 in high byte
    movq    [edi], mm0                ;store 8 bytes p0 and p1
    paddw   mm3,    mm6                ;reordered p2 and p3 in mm2
```

Using MMX™ Instructions to Compute the 2x2 Haar Transform

March 1996

```

    add     edx,      8           ;inc b3 index
    add     edi,      8           ;inc p0 and p1 index
    movq    [esi],    mm3        ;store 8 bytes p2 and p3
    sub     ebp,      8           ;row0 addr = in loop lim
    Jnz     iwave          ;jump if not end of row
    add     edi,      NCOL        ;new row inc row0 addr
    add     esi,      NCOL        ;new row inc row1 addr
    mov     ebp,      NCOL        ;old in loop lim
    cmp     edi,      OLOOP      ;row0 addr = oout loop lim
    Jnz     iwave
    pop     edi
    pop     esi
    pop     edx
    pop     ecx
    pop     ebx
    pop     eax
    pop     ebp
    ret     0
_iwavemmx ENDP
_TEXT
END
```