Using MMX™ Instructions to Convert RGB To YUV Color Conversion

Information for Developers and ISVs

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

March 1996

Using MMX™ Instructions to Convert RGB To YUV Color Conversion

March 1996

CONTENTS

1.0. INTRODUCTION

The Intel Architecture (IA) media extensions include single-instruction, multi-data (SIMD) instructions. This application note presents examples of code demonstrate how to convert RGB Color-Space Pixels to YUV Color-Space Pixels. Components of the YUV color space are linear combinations of the components of the RGB color space. Therefore, RGB to YUV color conversion is computed by multiplying a 3x3 coefficient matrix by a vector of RGB values.

The code presented here shows how to use the MMX instructions to significantly speed up RGB to YUV color conversion. The code includes the quadword shift instructions, PSLLQ and PSRLQ, which are used to position data in the 64-bit MMX registers to facilitate single instruction multiple data (SIMD) operations. Once positioned, packed-multiply-accumulate, PMADDWD, packed-add, PADDD, and packed-right-shift, PSRAD, instructions perform the multiplications, additions, and shifts required to compute Y, U, and V values. The 32-bit to 16-bit conversion, PACKSSDW, and 16-bit to 8-bit conversion instructions reduce the data size and clamp YUV values.

2.0. RGB TO YUV COLOR CONVERSION

Color spaces are three-dimensional (3D) coordinate systems in which each color is represented by a single point. Colors appear as their primary components red, green and blue, in the RGB color space. RGB is the format generally used by monitors. Each color appears as a luminance component, Y, and two chrominance components, U and V, in the YUV space. Luminance, the intensity perceived, is decoupled from the chrominance components so the intensity can be varied without affecting the color. The YUV format is used by PAL, the European television transmission standard, and it is the defacto standard used for image and video compression.

The parameters of the color conversion routine presented here are the address of the RGB buffer, which stores the input data, the number of rows and columns, and the addresses of the separate Y, U, and V buffers, which store the output data. The R, G, and B values are interleaved, and the data size of each is one byte. The data size of the Y, U, and V results are one byte, also. Therefore, the size of the RGB buffer in units of bytes is three times the product of the number of rows and columns, and the sizes of the YUV buffers in units of bytes is the product of the number of rows and the number of columns.

2.1 RGB To YUV Color Conversion Equations

Two sets of equations for RGB to YUV color conversion are given in Example 1. The first set is a floating-point version. The second set describes calculations made in the MMX code presented here. MMX registers execute integer operations. Coefficients in the second set are equal to the product of 32768, which equals 2^{15}, and the coefficients in the first set of equations rounded to the nearest integer and divided by 32768. The code adds 128 to the results for U and V to assure they are positive.

Example 1. RGB to YUV Color Conversion Equations

```
Y = 0.299R 0.587G + 0.114B Conventional floating-point equations
U =-0.146 R - 0.288 G + 0.434 B
V = 0.617 R - 0.517 G - 0.100 G
Y = [(9798 R + 19235G + 3736 B) / 32768] Equations used by code.
U = [(-4784 R - 9437 G + 4221 B) / 32768] + 128
V = [(20218R - 16941G - 3277 B) / 32768] + 128
```

The steps used to transform RGB to YUV are described in Example 2. A full loop processes 24 bytes. The arrangement of data shown in step 1 represents that for three loads. Effective use of MMX instructions requires that data be positioned in registers to take advantage of the SIMD capabilities of the MMX technology. A method for arranging data which permits efficient calculation of YUV values from interleaved RGB input is described in step 2. This facilitates the calculations in step 3. Steps 2 and 3 are described in Example 3. The first phase of step 2, represented by the shift instruction, varies depending on the arrangement of data loaded in step 1. Generally one instruction, and never more than three are required to in this phase. Step 2 positions data in the locations shown in the second two instructions shown in step 2 regardless of the locations when data is loaded in step 1. A first register is loaded, using the 8-bit to 6-bit unpack operation, with 16-bit values arranged $R_BB_AG_AR_A$ and a second register is similarly loaded with $B_BG_BR_BB_A$ where an R, a G, and a B value in the first register are associated with pixel A and an R, a G, and a B value in the second register are associated with adjacent pixel B. Step 3 shows how the pmaddwd instruction takes advantage of this arrangement. The operand used with the register containing $R_BB_AG_AR_A$ is a 64-bit local variable containing four 16-bit values in the form $C_R0C_BC_R$. The 32-bit results of the PMADDWD instruction are C_RR_B and $C_GG_A+C_RR_A$. The operand with the register containing $B_BG_BR_BB_A$ is the 64-bit local variable containing the four 16-bit values $C_BC_G0C_B$.

The 32-bit results of the PMADDWD instruction are C_BB_B+C_GG_B and B_AC_B. These results are combined with a 32-bit add to give C_BB_B+C_GG_B+C_RR_B and C_BB_A+C_GG_A+C_RR_A. The 32-bit results are shifted by 15 bits, the equivalent of dividing by 32768, and packed to reduce the data size to 8 bits. Values of the coefficients C_R, C_G, and C_B differ for the calculations of Y, U, and V.

Example 2. RGB to YUV MMX Technology Color Conversion Algorithm Steps

```
Step 1: Load 8-bit data

        load mm0 with 1 byte data                                   mm0 =   G2R2B1G1R1B0G0R0
        copy mm0 to mm1                                             mm1 =   G2R2B1G1R1B0G0R0
Step 2: Position data and expand to 16-bits giving R_BB_AG_AR_A and B_BG_BR_BB_A in
MMX registers.
        shift mm1 right 16                                          mm1 =   00G2R2G1B1R1B0
        unpack mm0 low bytes so data size is 2 bytes        mm0 =   R1B0G0R0
        unpack mm2 low bytes so data size is 2 bytes        mm2 =   B1G1R1B0
Step 3: Convert RGB to 32-bit YUV
        multiply-accumulate mm0 using operand C_R0C_BC_R    mm0 =   C_RR1, C_GG0+C_RR0
        multiply-accumulate mm1 using operand C_BC_G0C_B    mm1 =   C_BB1+C_GG1, C_RR0
        add mm0 and mm1                                     mm0 =   C_BB1+C_GG1+C_RR1,
                                                                        C_BB0+C_GG0+C_RR0
        shift 32-bit results right 15 bits                  mm0 =   (C_BB1+C_GG1+C_RR1)/2^15,

        {C_BB0+C_GG0+C_RR0)/2^15

        Do step 3 for Y, U and V
        Repeat above steps so there are 4 values for each Y, U and V.
        Pack 4 values so each is 16-bits.
        At this point 8 bytes have been processed. Repeat the steps above twice to
        process the remaining 16 bytes. Note the data arrangement in step 1 and
        instruction 1 in step 2 will vary.
Step 4: Add offset, reduce results to 1 byte and store
        add an offset to 16-bit U and V values
        pack and clamp 16-bit results into 8 bits
        write 8 one byte Y, U and V results
```

2.2 Subsampling YUV

The code presented here computes all U and V results and writes them into a buffer. In the cases of transmission and image and video compression U and V are generally subsampled because the eye is more sensitive to luminance represented by Y than chrominance represented by U and V. The code can be easily modified to subsample U and V. For example, subsampling with four Y values for each U and V value can be carried out by computing averages of U and V for 2x2 blocks. The averages of a two 2x2 blocks at a time are computed by first adding values in adjacent columns with two PMADDWD instructions, one instruction for each row of the 2x2 blocks. The PMADDWD operands are 16-bit data along the rows and a constant equal to four 16-bit ones. The sum of the two PMADDWD results yields sums of the values in the 2x2 blocks. Right shifts of these sums by two bits with a PSRAD instruction gives averages for U or V.

2.3 Color Conversion Core

Sections of the loop which is the core of the color conversion code are listed in Example 4. Sections listed demonstrate how the Y component is obtained. Code which computes the U and V components is similar. The loop has 122 instructions, of which 116 are paired. A total of eight pixels are processed by the loop. Therefore, there are three 64-bit loads of interleaved RGB data. The first load is on line 1, and the third load is on line 49. After data loaded it is shifted, and its size is increased to 16-bits following a load. The first shift executed to position data is on line 4. Steps taken to position the data differ throughout the loop, but the resulting pattern is always $R_BB_AG_AR_A$ and $B_BG_BR_BB_A$. Lines 5 and 7 increase the data size to 16-bits. All of the multiplications and two of the additions required to compute two Y components are carried out with the pmaddwd instruction on lines 9 and 11. Similar operations to compute U and V components are carried out on lines 11, 13, 15, and 17. The PMADDWD instruction increases the size of the data to 32-bits. The final two additions required to compute two Y components occur on line 18. Results of these additions are shifted by 15-bits, corresponding to division by 32768, on line 36. These two 32-bit values for Y are packed into two 16-bit locations with two additional 32-bit values for Y on line 46. These results are stored in a local variable to relieve register pressure on line 57. Line 107 reads the results back into a register where they, and for additional 16-bit Y results, are packed as 8-bit values on line 110. The PACKUSWB clamps the values between 255 and 0. The 8 Y results computed by the loop are store on line 115.

Example 4. Sections of the RGB to YUV MMX Technology Color Conversion Core

```
RGBtoYUV:
1       movq            mm1,     [eax]    ;load G2R2B1G1R1B0G0R0
2       pxor            mm6,     mm6      ;0 -> mm6
3       movq            mm0,     mm1      ;G2R2B1G1R1B0G0R0 -> mm0
4       psrlq           mm1,     16       ;00G2R2B1G1R1B0 -> mm1
5       punpcklbw       mm0,     ZEROS    ;R1B0G0R0 -> mm0
6       movq            mm7,     mm1      ;00G2R2B1G1R1B0 -> mm7
7       punpcklbw       mm1,     ZEROS    ;B1G1R1B0 -> mm1
8       movq            mm2,     mm0      ;R1B0G0R0 -> mm2
9       pmaddwd         mm0,     YR0GR    ;yrR1,ygG0+yrR0 -> mm0
10      movq            mm3,     mm1      ;B1G1R1B0 -> mm3
11      pmaddwd         mm1,     YBG0B    ;ybB1+ygG1,ybB0 -> mm1
12      movq            mm4,     mm2      ;R1B0G0R0 -> mm4
13      pmaddwd         mm2,     UR0GR    ;urR1,ugG0+urR0 -> mm2
14      movq            mm5,     mm3      ;B1G1R1B0 -> mm5
15      pmaddwd         mm3,     UBG0B    ;ubB1+ugG1,ubB0 -> mm3
16      punpckhbw       mm7,     mm6      ;00G2R2 -> mm7
17      pmaddwd         mm4,     VR0GR    ;vrR1,vgG0+vrR0 -> mm4
18      paddd           mm0,     mm1      ;Y1Y0 -> mm0
36      psrad           mm0,     15       ;32-bit scaled Y1Y0 -> mm0
37      movq            TEMP0,   mm6      ;R5B4G4R4 -> TEMP0
38      movq            mm6,     mm3      ;R3B2G2R2 -> mm6
39      pmaddwd  mm6,   UR0GR    ;urR3,ugG2+urR2 -> mm6
40      psrad           mm2,     15       ;32-bit scaled U1U0 -> mm2
41      paddd           mm1,     mm5      ;Y3Y2 -> mm1
42      movq            mm5,     mm7      ;B3G3R3B2 -> mm5
43      pmaddwd  mm7,   UBG0B    ;ubB3+ugG3,ubB2 -> mm7
44      psrad           mm1,     15       ;32-bit scaled Y3Y2 -> mm1
45      pmaddwd         mm3,     VR0GR    vrR3,vgG2+vgR2 ->mm3
46      packssdw mm0,   mm1      ;Y3Y2Y1Y0 -> mm0
47      pmaddwd         mm5,     VBG0B    ;vbB3+vgG3,vbB2 -> mm5
48      psrad           mm6,     mm7      ;U3U2 -> mm6
51      movq            mm7,     mm1      ;B7G7R7B6G6R6B5G5 -> mm1
52      psrad           mm6,     15       ;32-bit scaled U3U2 -> mm6
53      paddd           mm3,     mm5      ;V3V2 -> mm3
54      psllq           mm7,     16       ;R7B6G6R6B5G500 -> mm7
55      movq            mm5,     mm7      ;R7B6G6R6B5G500 -> mm5
56      psrad           mm3,     15       ;32-bit scaled V3V2 -> mm3
57      movq TEMPY,     mm0      ;32-bit scaled Y3Y2Y1Y0 -> TEMPY
107     movq            mm6,     TEMPY    ;32-bit scaled Y3Y2Y1Y0 -> mm6
108     packssdw        mm0,     mm7      ;32-bit scaled U7U6U5U4 -> mm0
109     movq            mm4,     TEMPU    ;32-bit scaled U3U2U1U0 -> mm4
110     packuswb mm6,   mm2      ;all 8 Y values -> mm6
```

```
        movq        mm7,      OFFSETB  ;128,128,128,128 -> mm7
        paddd       mm1,      mm5      ;V7V6 -> mm1
        paddw       mm4,      mm7      ;add offset to U3U2U1U0/256
        psrad       mm1,      15       ;32-bit scaled V7V6 -> mm1
        movq        [ebx],    mm6      ;store Y
        dec         edi                ;decrement loop counter
        jnz         RGBtoYUV ;do 24 more bytes if not 0
```

3.0. PERFORMANCE GAINS

Performance gains for color conversion from MMX instructions are difficult to specify because colors are generally converted with the use of tables. Although tables are less accurate than calculations, they are much more efficient. MMX technology color conversion performance is somewhat better than that of typical lookup table code and is gives more accurate results.

3.1 Scalar Performance

An example of IA color conversion code which uses lookup tables requires three instructions to read data, four instructions to increment read addresses, three instructions to read lookup tables, two instructions to combine table results, two shifts to get the correct YUV value to be stored, three instructions to write results, and three instructions to increment write addresses. If all instructions could be paired and all data were in the L1 cache the number of clocks per pixel using a lookup table would be 10.

A modified version of equations shown in Example 1 are given in Example 5. C code compiled with an optimizing compiler executes the first set of floating-point equations and clamps results in 108 clocks. C code executes the second set of integer equations in 125 clocks.

Example 5. Modified RGB to YUV Color Conversion Floating Point Equations

```
Y = 0.299 R + 0.587 G + 0.114 B Modified floating-point equations
U = 0.492 (B - Y)
V = 0.877 (R - Y)
Y = [(9798 R + 19235G + 3736 B) >>15]     Modified integer equations
U = [(16122 (B - Y))>>15]
V = [(25203 (R - Y))>>15]
```

3.2. MMX Code Performance

The MMX code takes 64 clocks to convert eight pixels of interleaved 24-bit RGB to 24-bit YUV with 15-bit accuracy. This result corresponds to conversion of one pixel in eight clocks. This result lower than the lookup table rate and it is more accurate. The speedup of MMX code compared with optimized C code for color space transformation calculations is more than a factor of 10. The high MMX code conversion rate and accuracy can be attributed to:

- MMX instructions facilitate multiple operations with a single instruction.

MMX code has a the fast multiply accumulate instruction, PMADDWD. The multiply accumulate operation requires three instructions and has significantly longer latency with conventional IA instructions.

4.0. YUV TO RGB COLOR CONVERSION: CODE LISTING

```
;rgbtoyuv.asm
;The loop processes interleaved RGB values for 8 pixels.
;The notation in the comments which describe the data locate
;the first byte on the right. For example in a register containing
;G2R2B1G1R1B0G0R0, R0 is in the position of the lease significant
;byte and G2 is in the position of the most significant byte.
;The output is to separate Y, U, and V buffers. Both input and
;output data are bytes.
        TITLE rgbtoyuv
        .486P
.model FLAT
PUBLIC _rgbtoyuv
_DATA SEGMENT
ALIGN   8
ZEROSX  dw      0,0,0,0
ZEROS   dd      ?,?
OFFSETDX        dw      0,64,0,64       ;offset used before shift
OFFSETD         dd      ?,?
OFFSETWX        dw      128,0,128,0     ;offset used before pack 32
OFFSETW         dd      ?,?
OFFSETBX        dw      128,128,128,128
OFFSETB         dd      ?,?
TEMP0           dd      ?,?
TEMPY   dd      ?,?
TEMPU           dd      ?,?
TEMPV   dd      ?,?
YR0GRX  dw      9798,19235,0,9798
YBG0BX  dw      3736,0,19235,3736
YR0GR   dd      ?,?
YBG0B   dd      ?,?
UR0GRX  dw      -4784,-9437,0,-4784
UBG0BX  dw      14221,0,-9437,14221
UR0GR   dd      ?,?
UBG0B   dd      ?,?
VR0GRX  dw      20218,-16941,0,20218
VBG0BX  dw      -3277,0,-16941,-3277
VR0GR   dd      ?,?
VBG0B   dd      ?,?
_DATA ENDS
_TEXT SEGMENT
_inPtr$ =        8
_rows$  =       12
_columns$       =       16
_outyPtr$       =       20
_outuPtr$       =       24
_outvPtr$       =       28
_rgbtoyuv PROC NEAR
        push    ebp
        mov     ebp,    esp
        push    eax
        push    ebx
        push    ecx
        push    edx
        push    esi
        push    edi
        lea     eax,    ZEROSX  ;This section gets around a bug
        movq    mm0,    [eax]   ;unlikely to persist
        movq    ZEROS,  mm0
        lea     eax,    OFFSETDX
```

```
        movq    mm0,     [eax]
        movq    OFFSETD, mm0
        lea     eax,     OFFSETWX
        movq    mm0,     [eax]
        movq    OFFSETW, mm0
        lea     eax,     OFFSETBX
        movq    mm0,     [eax]
        movq    OFFSETB, mm0
        lea     eax,     YR0GRX
        movq    mm0,     [eax]
        movq    YR0GR,   mm0
        lea     eax,     YBG0BX
        movq    mm0,     [eax]
        movq    YBG0B,   mm0
        lea     eax,     UR0GRX
        movq    mm0,     [eax]
        movq    UR0GR,   mm0
        lea     eax,     UBG0BX
        movq    mm0,     [eax]
        movq    UBG0B,   mm0
        lea     eax,     VR0GRX
        movq    mm0,     [eax]
        movq    VR0GR,   mm0
        lea     eax,     VBG0BX
        movq    mm0,     [eax]
        movq    VBG0B,   mm0
        mov     eax,     _rows$[ebp]
        mov     ebx,     _columns$[ebp]
        mul     ebx               ;number pixels
        shr     eax,     3        ;number of loops
        mov     edi,     eax      ;loop counter in edi
        mov     eax,     _inPtr$[ebp]
        mov     ebx,     _outyPtr$[ebp]
        mov     ecx,     _outuPtr$[ebp]
        mov     edx,     _outvPtr$[ebp]
        sub     edx,     8        ;incremented before write
RGBtoYUV:
        movq    mm1,     [eax]    ;load G2R2B1G1R1B0G0R0
        pxor    mm6,     mm6      ;0 -> mm6
        movq    mm0,     mm1      ;G2R2B1G1R1B0G0R0 -> mm0
        psrlq   mm1,     16       ;00G2R2B1G1R1B0-> mm1
        punpcklbw        mm0,     ZEROS    ;R1B0G0R0 -> mm0
        movq    mm7,     mm1      ;00G2R2B1G1R1B0-> mm7
        punpcklbw        mm1,     ZEROS    ;B1G1R1B0 -> mm1
        movq    mm2,     mm0      ;R1B0G0R0 -> mm2
        pmaddwd mm0,     YR0GR    ;yrR1,ygG0+yrR0 -> mm0
        movq    mm3,     mm1      ;B1G1R1B0 -> mm3
        pmaddwd mm1,     YBG0B    ;ybB1+ygG1,ybB0 -> mm1
        movq    mm4,     mm2      ;R1B0G0R0 -> mm4
        pmaddwd mm2,     UR0GR    ;urR1,ugG0+urR0 -> mm2
        movq    mm5,     mm3      ;B1G1R1B0 -> mm5
        pmaddwd mm3,     UBG0B    ;ubB1+ugG1,ubB0 -> mm3
        punpckhbw        mm7,     mm6;     00G2R2 -> mm7
        pmaddwd mm4,     VR0GR    ;vrR1,vgG0+vrR0 -> mm4
        paddd   mm0,     mm1      ;Y1Y0 -> mm0
        pmaddwd mm5,     VBG0B    ;vbB1+vgG1,vbB0 -> mm5
        movq    mm1,     8[eax]   ;R5B4G4R4B3G3R3B2 -> mm1
        paddd   mm2,     mm3      ;U1U0 -> mm2
        movq    mm6,     mm1      ;R5B4G4R4B3G3R3B2 -> mm6
        punpcklbw        mm1,     ZEROS    ;B3G3R3B2 -> mm1
        paddd   mm4,     mm5      ;V1V0 -> mm4
        movq    mm5,     mm1      ;B3G3R3B2 -> mm5
        psllq   mm1,     32       ;R3B200 -> mm1
        paddd   mm1,     mm7      ;R3B200+00G2R2=R3B2G2R2->mm1
```

```
punpckhbw        mm6,     ZEROS    ;R5B4G4R3 -> mm6
movq     mm3,     mm1      ;R3B2G2R2 -> mm3
pmaddwd  mm1,     YR0GR    ;yrR3,ygG2+yrR2 -> mm1
movq     mm7,     mm5      ;B3G3R3B2 -> mm7
pmaddwd  mm5,     YBG0B    ;ybB3+ygG3,ybB2 -> mm5
psrad    mm0,     15       ;32-bit scaled Y1Y0 -> mm0
movq     TEMP0,   mm6      ;R5B4G4R4 -> TEMP0
movq     mm6,     mm3      ;R3B2G2R2 -> mm6
pmaddwd  mm6,     UR0GR    ;urR3,ugG2+urR2 -> mm6
psrad    mm2,     15       ;32-bit scaled U1U0 -> mm2
paddd    mm1,     mm5      ;Y3Y2 -> mm1
movq     mm5,     mm7      ;B3G3R3B2 -> mm5
pmaddwd  mm7,     UBG0B    ;ubB3+ugG3,ubB2
psrad             mm1, 15  ;32-bit scaled Y3Y2 -> mm1
pmaddwd  mm3,     VR0GR    ;vrR3,vgG2+vgR2
packssdw          mm0,     mm1      ;Y3Y2Y1Y0 -> mm0
pmaddwd  mm5,     VBG0B    ;vbB3+vgG3,vbB2 -> mm5
psrad    mm4,     15       ;32-bit scaled V1V0 -> mm4
movq     mm1,     16[eax]  ;B7G7R7B6G6R6B5G5 -> mm7
paddd    mm6,     mm7      ;U3U2 -> mm6
movq     mm7,     mm1      ;B7G7R7B6G6R6B5G5 -> mm1
psrad    mm6,     15       ;32-bit scaled U3U2 -> mm6
paddd    mm3,     mm5      ;V3V2 -> mm3
psllq    mm7,     16       ;R7B6G6R6B5G500 -> mm7
movq     mm5,     mm7      ;R7B6G6R6B5G500 -> mm5
psrad    mm3,     15       ;32-bit scaled V3V2 -> mm3
movq     TEMPY,   mm0      ;32-bit scaled Y3Y2Y1Y0 -> TEMPY
packssdw          mm2,     mm6      ;32-bit scaled U3U2U1U0 -> mm2
movq     mm0,     TEMP0    ;R5B4G4R4 -> mm0
punpcklbw         mm7,     ZEROS    ;B5G500 -> mm7
movq     mm6,     mm0      ;R5B4G4R4 -> mm6
movq     TEMPU,   mm2      ;32-bit scaled U3U2U1U0 -> TEMPU
psrlq    mm0,     32       ;00R5B4 -> mm0
paddw    mm7,     mm0      ;B5G5R5B4 -> mm7
movq     mm2,     mm6      ;B5B4G4R4 -> mm2
pmaddwd  mm2,     YR0GR    ;yrR5,ygG4+yrR4 -> mm2
movq     mm0,     mm7      ;B5G5R5B4 -> mm0
pmaddwd  mm7,     YBG0B    ;ybB5+ygG5,ybB4 -> mm7
packssdw          mm4,     mm3      ;32-bit scaled V3V2V1V0 -> mm4
add      eax,     24       ;increment RGB count
add      edx,     8        ;increment V count
movq     TEMPV,   mm4      ;(V3V2V1V0)/256 -> mm4
movq     mm4,     mm6      ;B5B4G4R4 -> mm4
pmaddwd  mm6,     UR0GR    ;urR5,ugG4+urR4
movq     mm3,     mm0      ;B5G5R5B4 -> mm0
pmaddwd  mm0,     UBG0B    ;ubB5+ugG5,ubB4
paddd    mm2,     mm7      ;Y5Y4 -> mm2
pmaddwd  mm4,     VR0GR    ;vrR5,vgG4+vrR4 -> mm4
pxor     mm7,     mm7      ;0 -> mm7
pmaddwd  mm3,     VBG0B    ;vbB5+vgG5,vbB4 -> mm3
punpckhbw         mm1,     mm7      ;B7G7R7B6 -> mm1
paddd    mm0,     mm6      ;U5U4 -> mm0
movq     mm6,     mm1      ;B7G7R7B6 -> mm6
pmaddwd  mm6,     YBG0B    ;ybB7+ygG7,ybB6 -> mm6
punpckhbw         mm5,     mm7      ;R7B6G6R6 -> mm5
movq     mm7,     mm5      ;R7B6G6R6 -> mm7
paddd    mm3,     mm4      ;V5V4 -> mm3
pmaddwd  mm5,     YR0GR    ;yrR7,ygG6+yrR6 -> mm5
movq     mm4,     mm1      ;B7G7R7B6 -> mm4
pmaddwd  mm4,     UBG0B    ;ubB7+ugG7,ubB6 -> mm4
psrad    mm0,     15       ;32-bit scaled U5U4 -> mm0
paddd    mm0,     OFFSETW  ;add offset to U5U4 -> mm0
psrad    mm2,     15       ;32-bit scaled Y5Y4 -> mm2
```

```
        paddd   mm6,    mm5     ;Y7Y6 -> mm6
        movq    mm5,    mm7     ;R7B6G6R6 -> mm5
        pmaddwd mm7,    UR0GR   ;urR7,ugG6+ugR6 -> mm7
        psrad   mm3,    15      ;32-bit scaled V5V4 -> mm3

        pmaddwd mm1,    VBG0B   ;vbB7+vgG7,vbB6 -> mm1
        psrad   mm6,    15      ;32-bit scaled Y7Y6 -> mm6
        paddd   mm4,    OFFSETD ;add offset to U7U6
        packssdw        mm2,    mm6     ;Y7Y6Y5Y4 -> mm2
        pmaddwd mm5,    VR0GR   ;vrR7,vgG6+vrR6 -> mm5
        paddd   mm7,    mm4     ;U7U6 -> mm7
        psrad   mm7,    15      ;32-bit scaled U7U6 -> mm7
        movq    mm6,    TEMPY   ;32-bit scaled Y3Y2Y1Y0 -> mm6
        packssdw        mm0,    mm7     ;32-bit scaled U7U6U5U4 -> mm0
        movq    mm4,    TEMPU   ;32-bit scaled U3U2U1U0 -> mm4
        packuswb        mm6,    mm2     ;all 8 Y values -> mm6
        movq    mm7,    OFFSETB ;128,128,128,128 -> mm7
        paddd   mm1,    mm5     ;V7V6 -> mm1
        paddw   mm4,    mm7     ;add offset to U3U2U1U0/256
        psrad   mm1,    15      ;32-bit scaled V7V6 -> mm1
        movq    [ebx],  mm6     ;store Y
        packuswb        mm4,    mm0     ;all 8 U values -> mm4
        movq    mm5,    TEMPV   ;32-bit scaled V3V2V1V0 -> mm5
        packssdw        mm3,    mm1     ;V7V6V5V4 -> mm3
        paddw   mm5,    mm7     ;add offset to   V3V2V1V0
        paddw   mm3,    mm7     ;add offset to   V7V6V5V4
        movq    [ecx],  mm4     ;store U
        packuswb        mm5,    mm3     ;ALL 8 V values -> mm5
        add     ebx,    8       ;increment Y count
        add     ecx,    8       ;increment U count
        movq    [edx],  mm5     ;store V
        dec     edi             ;decrement loop counter
        jnz     RGBtoYUV ;do 24 more bytes if not 0
        pop     edi
        pop     esi
        pop     edx
        pop     ecx
        pop     ebx
        pop     eax
        pop     ebp
        ret     0
_rgbtoyuv ENDP
_TEXT ENDS
END
```