



# Using MMX™ Instructions to Implement 2D Sprite Overlay

Information for Developers and ISVs

From Intel® Developer Services  
[www.intel.com/IDS](http://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 Implement 2D Sprite Overlay**

---

March 1996

## **CONTENTS**

1.0. INTRODUCTION

2.0. SPRITE OVERLAY FUNCTION

2.1. Sprite Engine

2.2. Sprite Overlay Core

3.0. PERFORMANCE GAINS

3.1. Scalar Performance

3.2. MMX™ Technology Performance

4.0. SPRITE OVERLAY CODE LISTING

## 1.0. INTRODUCTION

The Intel Architecture (IA) media extensions include single-instruction, multi-data (SIMD) instructions. This application note presents examples of code that exploits these instructions.

Sprites are computer characters which generally appear in the foreground. They are implemented by overlaying small sprite images on a large background image. Characters do not cover the entire sprite image, which is usually rectangular, so some parts of the image are transparent.

Methods for implementing sprites differ depending on the application. The MMX™ technology. Sprite Overlay function described here uses a read-modify-write approach. The function reads the background under the sprite from video memory and uses the new `pcmpeqb` instruction to construct a mask which permits the background to show through transparent regions of the sprite image. Eight sprite pixels and eight background pixels are combined with the MMX instruction sequence `pcmpeqb`, `pand` and `por` without any branches. This function writes only in the region covered by the sprite.

This approach to implementing sprites cannot be used in systems with video cards which do not permit reading directly from video memory. In systems in which direct access is permitted the approach works best for a single small sprite because reading from video memory is slow. When more than one sprite are implemented those in back must be written first so that if sprites overlap those in front will overwrite appropriate parts of sprites in back by treating them as background.

More complex applications implement sprites differently. One approach used in applications with several sprites is to write the entire scene from back to front every time. Only valid sprite pixels are written. If a sprite does not change the number of branches can be reduced by branching to a location which represents several branches. This can be done with codes which indicate which sprite pixels are valid. For example, there are sixteen possible combinations of valid and not valid pixels for a group of four pixels. Branching to one of sixteen locations which represents a sequence of valid and not valid values eliminates three out of four branches. Further information about sprites is available in *Zen of Graphics Programming* by Michael Abrash, and *Tricks of the Game Programming Gurus* by LaMothe, Ratcliff, Seminatore and Tyler.

## 2.0. SPRITE OVERLAY FUNCTION

The overlay function is one of several functions called by a sprite engine to control sprites.

### 2.1. Sprite Engine

The skeleton of a simple sprite engine for a single sprite which describes how an overlay function is related to other sprite functions is shown in Example 1. There are two basic phases in the main loop of this engine for a single sprite. The first of these is the control phase, and the second is the draw phase. The control phase determines when, where, and what to draw, and the draw phase restores the original background which the sprite has overwritten and redraws the sprite with the overlay function. The engine in Example 1 makes a character appear alive by moving and animating it. The sprite is moved by drawing it in a different location on the background, and it is animated by drawing a different frame so parts of its body appear to move. This simple engine animates the sprite by drawing the next frame in a sequence. Sophisticated sprite engines take into account factors such as collisions with features in the background and other sprites and 3D modeling.

More than one sprite can be implemented in the sprite engine in Example 1 by changing the structure of the main while loop so there is an outer loop with an inner loop which replaces the background followed an inner loop which draws the sprites with the Sprite Overlay function. The Sprite Overlay pseudocode function and the corresponding `sprite_overlay` MMX function call save the background behind the sprite in a buffer before the sprite is written, so the background can be restored. If there are several sprites the background of a sprite may include part of a sprite behind it. Therefore, the loop which restores the background should draw the backgrounds saved in buffers from front to back, and the loop which overlays the sprites should draw the sprites from back to front.

*Example 1. Skeleton of a Simple Engine for a Single Sprite*

```
//Some Structures
typedef struct image_type {
//information about images and a pointer to the image buffer
}Image, *ImagePtr;
type struct sprite_type
    int x_cur, y_cur;           //current position
    int x_old, y_old;         //previous position
    int width, height;        //sprite size
    int move_period, move_count; //loops until moved
    int animate_period, animate_count; //loops per frame
    int frame_total, frame_count //frame currently displayed
    char * sprite_background; //background under sprite
    char * frame[NumberSpriteFrames];
    int state;
}Sprite, *SpritePtr;

//Some Variables
Image background, sprites;
Sprite thesprite;
//Display Sprite
main()
{
    //Initialize
    SetVideoMode(mode); //set VGA card mode
```

## Using MMX™ Instructions to Implement 2D Sprite Overlay

March 1996

```
//Read in background image and display it on the screen.
AllocateBuffer((ImagePtr)&background, ScreenWidth*ScreenHeight);
LoadBuffer("background.dat", (ImagePtr)&background);
DecompressBuffer((ImagePtr)&background);
PaletteToVGA((ImagePtr)&background);
DisplayBuffer((ImagePtr)&background, ScreenWidth*ScreenHeight);
FreeBuffer((ImagePtr)&background);
//Initialize sprite structure. Transfer sprite images from large image.
AllocateBuffer((ImagePtr)&sprites, NumberSpriteFrames*
    SpriteWidth*SpriteHeight);
LoadBuffer("sprites.dat", (ImagePtr)&sprites);
DecompressBuffer((ImagePtr)&sprites);
InitializeSpriteMotion((SpritePtr)&current_frame, x, y,
    move_period, move_count, animate_period, animate_count);
InitializeSpriteFrames((ImagePtr)&sprites,
    (SpritePtr)&thesprite, NumberSpriteFrames);
FreeBuffer((ImagePtr)&sprites);
//Save background under sprite, and draw the first sprite frame.
SpriteOverlay((SpritePtr)&thesprite);
//Main Loop
while(thesprite.state == ALIVE) {
    if(++thesprite.animate_count >          //test for draw new frame
        thesprite.animate_period) {
        thesprite.animate_count = 0; //reset animation counter
        if(++frame_count > frame_total) {
            thesprite.frame_count = 0; } //reset frame number
        draw_flag = 1;
    }
    if(++thesprite.move_count > thesprite.move_period) {
        thesprite.move_count = 0; //reset move counter
        move_flag = draw_flag = 1;
    }
    if(move_flag) { //determine where to draw the sprite
        SpritePosition((SpritePtr)&thesprite); //update position
        if(thesprite.state == DEAD) draw_flag = 0;
    }
    if(draw_flag) {
        ReplaceBackground((SpritePtr)&thesprite);
        SpriteOverlay((SpritePtr)&thesprite);
        draw_flag = move_flag = 0; //reset flags
    }
}
}
```

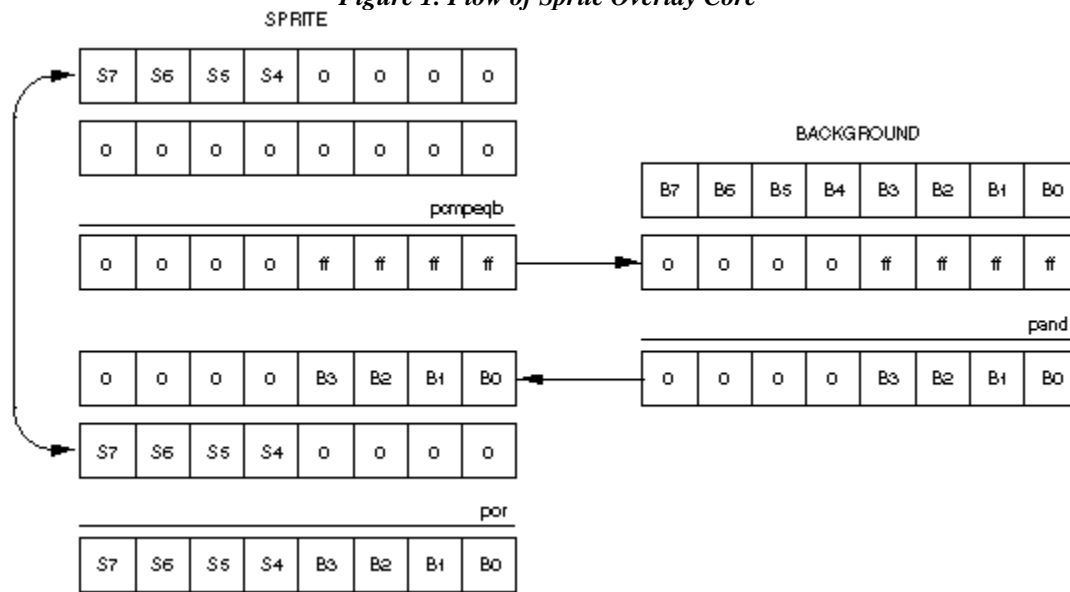
### 2.2. Sprite Overlay Core

Figure 1 illustrates the flow of MMX instructions used to draw sprite pixels which are not transparent. Transparent pixels have values equal to zero. The example shows a case in which the lower 4 bytes (those on the right) are transparent and the upper 4 are not transparent. The first step sprite data are compared with mask data equal to zero using the pcmpeqb instruction. In cases in which the corresponding bytes of sprite data are zero, the mask data are changed to ff, and in cases in which the corresponding bytes of sprite data are not zero, the mask data remain equal to zero. The second step the mask and background data are combined with the pand instruction. This step retains background data in locations where the sprite data is zero, and sets background data equal to zero in locations where sprite data is not zero. The third step the sprite and background data are combined with the por instruction.

# Using MMX™ Instructions to Implement 2D Sprite Overlay

March 1996

Figure 1. Flow of Sprite Overlay Core



The core of the sprite overlay code is listed in Example 2. Sprite data is loaded in instruction 3 and background data is loaded in instruction 5. Values of the mask are initialized to zero in instruction 4, and values are set to ff where the sprite is transparent in instruction 6. In instruction 8 mask values which were equal to ff are set equal to values of the background data, and mask values which were equal to zero remain zero. Sprite data and background data are combined in instruction 10, and written to the video buffer in instruction 12. Background data which is overwritten by sprite data is saved in instruction 11.

Background data is saved so that the background can be restored when the sprite is moved.

Example 2. Sprite Overlay Core

LoopTop:

```

1      add    ebx, 8          ;counter for buffer region under sprite
2      add    ecx, 8          ;counter for video address to write sprite
3      movq   mm0, [eax]      ;read sprite
4      pxor   mm7, mm7        ;zero in mm7 which is the mask
5      movq   mm1, [ecx]      ;read background under sprite
6      pcmpeqb mm7, mm0       ;make mask where sprite is transparent
7      add    eax, 8          ;counter for sprite buffer
8      pand   mm7, mm1        ;pixels of region under sprite to draw

9      add    ebp, 8          ;increment inner loop counter
10     por    mm7, mm0        ;combine sprite and background
11     movq   [ebx], mm1      ;save background overwritten by sprite

12     movq   [ecx], mm7      ;write sprite
13     cmp    ebp, edi        ;done with current sprite row?
14     jne    LoopTop

```

### 3.0. PERFORMANCE GAINS

Several factors make comparison of the MMX technology performance and scalar performance of sprite overlay functions difficult. One factor is reading video data is system dependent. A second factor is that sprite overlay functions tend to be optimized for the application for which they are written, so sprite overlay performance varies from application to application.

#### 3.1. Scalar Performance

A section of scalar code which handles the transparency problem in a manner similar to the MMX code in Example 2 is shown in Example 3. The loop processes four pixels. This code does not represent optimized scalar code which would be used to implement sprites. Rather it contrasts the efficiency of scalar code and MMX code when scalar code implements sprite overlay in a manner similar to the method used by the MMX code presented here.

*Example 3. Scalar Sprite Overlay Core*

```
Position0:
    mov     eax, [ebx]           ;4 pixels sprite data
    mov     ecx, [edx]           ;4 pixels background data to ecx
    xor     ebp, ebp             ;0->ebp. ebp holds sprite overlay results
    mov     [edi], ecx           ;store background
    add     ebx, 4                ;increment sprite address
    mov     esi, 0x000000ff      ;load mask
    and     esi, eax             ;determine if pixel is transparent
    jz     Position1            ;jump to position 1 if transparent
    or     ebp, esi              ;first pixel of sprite in sprite overlay
    jmp    Position2            ;do the next pixel
Position1:
    mov     esi, 0x000000ff      ;reload mask

    and     esi, ecx             ;first pixel of background in esi

    or     ebp, esi              ;background in sprite overlay
    add     edi, 4                ;increment background address
    mov     [edx], ebp           ;write sprite overlay
    add     edx, 4                ;increment video address
    cmp    ebx, LAST_SPRITE_ADDRESS
    jnz    Position0
```

#### .2. MMX™ Technology Performance

Instructions in the inner loop take eight clocks to execute which is equivalent to one clock per pixel if reading from video memory is not taken into account.

Misaligned 64-bit accesses have a three clock penalty. The code presented here does nothing to avoid this problem. Misalignment can often be avoided, but there is a generally a cost. For example, instead of a single sprite image for a particular sprite there could be eight images each with the valid pixels shifted with respect to each other along rows. The sprite engine would choose the sprite with no misalignment penalty. Costs of this method include one additional pass through the overlay loop each row because the width of the sprite would increase, overhead to determine which sprite to call and memory required for seven extra sprite images.

## Using MMX™ Instructions to Implement 2D Sprite Overlay

---

March 1996

MMX technology benefits are due to several factors:

- MMX technology adds the pcmpeqb instruction which avoids conditional branches.
- MMX technology permits single instruction multiple data (SIMD) operations.
- MMX technology adds the 64-bit movq instruction which reduces memory stalls when loading non-cacheable video data.
- MMX technology adds registers reducing register pressure.

## 4.0. SPRITE OVERLAY CODE LISTING

```
;sprite overlay
;This function assumes that the width of the sprite is divisible by 8
    TITLE testmmx
    .486
.model FLAT
PUBLIC _sprite_overlay
_DATA SEGMENT
_DATA ENDS
_TEXT SEGMENT
_spritePtr$    = 8           ;the sprite
_backPtr$     = 12          ;buffer to save region under sprite
_videoPtr$    = 16          ;video buffer with large background
_offset_video$ = 20         ;offset from beginning of video buffer
_width_screen$ = 24         ;width of large background image
_width_sprite$ = 28
_height_sprite$ = 32
_sprite_overlay PROC NEAR
    push ebp
    mov ebp, esp
    push eax
    push ebx
    push ecx
    push edx
    push esi
    push edi

    mov ecx, _videoPtr$[ebp]
    mov edi, _offset_video$[ebp]
    mov edx, _width_screen$[ebp]
    add ecx, edi

    mov eax, _spritePtr$[ebp]
    mov ebx, _backPtr$[ebp]
    mov edi, _width_sprite$[ebp]
    mov esi, _height_sprite$[ebp]

    sub edx, edi
    xor ebp, ebp

    sub ebx, 8
    sub ecx, 8
LoopTop:

    add    ebx, 8           ;counter for buffer region under sprite
    add    ecx, 8           ;counter for video address to write sprite
    movq   mm0, [eax]      ;read sprite
    pxor   mm7, mm7        ;zero in mm7
    movq   mm1, [ecx]      ;read background under sprite
    pcmpeqb mm7, mm0       ;make mask where sprite is transparent
    add    eax, 8           ;counter for sprite buffer
    pand   mm7, mm1        ;pixels of region under sprite to draw

    add    ebp, 8           ;inner loop counter
    por    mm7, mm0        ;combine sprite and background
```

## Using MMX™ Instructions to Implement 2D Sprite Overlay

---

March 1996

```
        movq    [ebx], mm1        ;save background overwritten by sprite

        movq    [ecx], mm7        ;write sprite overlay results
        cmp    ebp, edi          ;done with current sprite row?
        jne    LoopTop
        add    ecx, edx          ;advance region to next buffer row
        xor    ebp, ebp          ;reset inner loop counter
        dec    esi              ;last row of sprite ?
        jnz    LoopTop

        pop    edi
        pop    esi
        pop    edx
        pop    ecx
        pop    ebx
        pop    eax
        pop    ebp
        ret    0

_sprite_overlay ENDP
_TEXT ENDS
END
```