```
// init shared hardware device
CSharedHardware *pGrfx = new CSharedHardware();
// init direct draw
if (!pGrfx->InitDirectDraw()) return FALSE;

// init d3d
if (!pGrfx->InitDirect3D(USE_RAMP)) return FALSE ;

// set up cooperative level
if (!pGrfx->SetCoopLevel(hWnd, DDSCL_NORMAL)) return FALSE;
```

Initialize Direct3D with Ramp driver.

Do revisit the code in Section 14.4.2 if you need to refresh your memory on how to enumerate and select a driver.

16.3.2 Using the Ramp Driver—The First Try

Once we've loaded the new driver, can't we just go ahead and create objects CSurface3D and CTriangle3D or CTriangleTex as usual and run them using the Ramp driver, instead of the RGB driver? The answer is no.

When we first tried running our simple triangle with the Ramp driver, we saw our background being painted correctly, but our CTriangle3D (shaded triangle) was drawn as a black triangle. When we tried using CTriangleTex (texture mapped triangle), our application *crashed,* right in the middle of the Ramp driver rendering module, with no clue as to what was wrong.

The key lies in remembering how the *Ramp model* operates—through lookup tables (see section 15.3.4). The Ramp driver uses only the Blue component of a color specification and then accesses "a lookup table" to interpret the final result. The Ramp driver builds lookup tables from material definitions. If no lookup table has been built (because, say, no material was created), then the rendering module crashes. Solution: create materials.

16.3.3 Creating Materials for the Ramp Driver

We have repeated the definition of the D3DMATERIAL structure here for easy reference:

```
typedef struct _D3DMATERIAL {
    DWORD               dwSize;
    D3DCOLORVALUE       dcvDiffuse;   |
    D3DCOLORVALUE       dcvAmbient ;  | Four different color components.
    D3DCOLORVALUE       dcvSpecular;  |
    D3DCOLORVALUE       dcvEmissive;  |
    D3DVALUE            dvPower ;       ◊ Specify sharpness of specular reflections.
    D3DTEXTUREHANDLE    hTexture;       ◊ Combine a texture with specified coloring.
    DWORD               dwRampSize;     ◊ Shading gradient of colors in Ramp/Mono model.
} D3DMATERIAL, *LPD3DMATERIAL;
```

The Ramp driver builds lookup tables based on the specifications in a material structure:

- For materials with no specularity, the driver builds a linear "color ramp" ranging from the ambient color to the maximum diffuse color.
- For materials with specularity, the driver builds a two-stage color ramp; the first stage ranges from the ambient color to the maximum diffuse color, and the second stage ranges from maximum diffuse color light to the maximum specular color. The gradient of the specular ramp is not linear, and it is controlled by the dvPower field.
- For materials with textures, the Ramp driver builds a color ramp for each color in the texture.
- The Ramp driver references the dwRampSize field to determine the size of the ramp built for each color.

For example, the following code sequence builds a color ramp with sixteen shades of red:

```
mMaterialDesc.dcvDiffuse.dvR = D3DVALUE(1.00);
mMaterialDesc.dcvDiffuse.dvG = D3DVALUE(0.00);
mMaterialDesc.dcvDiffuse.dvB = D3DVALUE(0.00);
m_MaterialDesc.hTexture = NULL;
m_MaterialDesc.dwRampSize = 16;
```

In this next example, the Ramp driver builds a color ramp with eight shades for each color in the associated texture:

```
mMaterialDesc.dcvDiffuse.dvR = D3DVALUE(1.00);
mMaterialDesc.dcvDiffuse.dvG = D3DVALUE(1.00);
mMaterialDesc.dcvDiffuse.dvB = D3DVALUE(1.00);
mMaterialDesc.hTexture = hTexture;
mMaterialDesc.dwRampSize = 8
```

16.3.4 Rendering a Triangle with the Ramp Driver

Now that we've taken a look at how the Ramp driver builds its lookup tables, let's create a CTriangleRamp to render a triangle with a shaded Ramp driver.

```
BOOL CTriangleRamp::Init(LPDIRECT3D pD3D, LPDIRECTDRAWPALETTE pPalette, UINT nRes)
{
```

Create material to "set" palette entries for color.

```
    pD3D->CreateMaterial(&m_pMaterialFns, NULL);
    m_MaterialDesc.dcvDiffuse.dvR = D3DVALUE(1.00);
    m_MaterialDesc.dcvDiffuse.dvG = D3DVALUE(1.00);
    m_MaterialDesc.dcvDiffuse.dvB = D3DVALUE(1.00);
    m_MaterialDesc.hTexture = NULL;
    m_MaterialDesc.dwRampSize = 16;
    m_pMaterialFns->SetMaterial(&m_MaterialDesc);
    m_pMaterialFns->GetHandle(p3dFns, &m_hMaterial);
```

The texture handle is NULL, the MaxDiffuse color is WHITE, and the RampSize is 16. The Ramp driver will create fourteen shades of gray between BLACK and WHITE.

Standard code to allocate system memory space for an Execute Buffer

```
    #define nTRIS  1
    #define nVERTS nTRIS*3
    m_sztEx = sizeof(D3DTLVERTEX) * nVERTS;
    m_sztEx += sizeof(D3DINSTRUCTION) * 5;
    m_sztEx += sizeof(D3DSTATE) * 2;
    m_sztEx += sizeof(D3DPROCESSVERTICES);
    m_sztEx += sizeof(D3DTRIANGLE) * nTRIS;
    m_pSysExBuffer = new BYTE [m_sztEx];
    memset(m_pSysExBuffer, 0, m_sztEx);
```

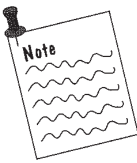
Use standard code to initialize vertices and then override the colors.

```
    D3DTLVERTEX *aVerts = (D3DTLVERTEX *)m_pSysExBuffer;
    setupVertices(nTRIS, aVerts);
    int    i;
    for (i=0; i<nTRIS; i++) {
        aVerts[0].color = RGBA_MAKE(000, 000, 255, 255);
        aVerts[1].color = RGBA_MAKE(000, 000, 128, 255);
        aVerts[2].color = RGBA_MAKE(000, 000, 000, 255);
        aVerts += 3;
    }
```

The Ramp driver uses only the blue component and ignores the red and green components.

PART V

The notable addition when setting up instructions for a triangle with the ramp model is the D3DOP_STATELIGHT opcode with its D3DLIGHTSTATE_MATERIAL operand. (We're using the OP_STATE_LIGHT macro.) Any materials that we've created have merely instructed the Ramp driver on how we want our lookup tables built. We use the D3DOP_STATELIGHT instruction in an Execute Buffer to instruct the Ramp driver to use a specific material for all future rendering.



The D3DOP_STATELIGHT instruction seems to turn off the render state. The default state is inoperative, and triangles will not be rendered unless you reset the render state. The D3DOP_STATERENDER specification must *follow* the D3DOP_STATELIGHT instruction, as render states set before the Light state become inoperative. You may want to set all render states that concern you and not assume the value of any state.

Set up instructions in Execute Buffer.

```
DWORD dwStart = sizeof(D3DTLVERTEX) * nVERTS;
LPVOID lpTmp = (LPVOID)(m_pSysExBuffer + dwStart);
OP_STATE_LIGHT(1, lpTmp);
    STATE_DATA(D3DLIGHTSTATE_MATERIAL, m_hMaterial, lpTmp);
OP_STATE_RENDER(1, lpTmp);
    STATE_DATA(D3DRENDERSTATE_SHADEMODE, D3DSHADE_GOURAUD, lpTmp);
OP_PROCESS_VERTICES(1, lpTmp);
    PROCESSVERTICES_DATA(D3DPROCESSVERTICES_COPY, 0, nVERTS, lpTmp);
OP_TRIANGLE_LIST(nTRIS, lpTmp);
for (i=0; i<nTRIS; i++) {
    ((LPD3DTRIANGLE)lpTmp)->v1 = i*3+0;
    ((LPD3DTRIANGLE)lpTmp)->v2 = i*3+1;
    ((LPD3DTRIANGLE)lpTmp)->v3 = i*3+2;
    ((LPD3DTRIANGLE)lpTmp)->wFlags = 0;
    lpTmp = ((char*)lpTmp) + sizeof(D3DTRIANGLE);
}
OP_EXIT(lpTmp);
DWORD dwLth = (LPBYTE)lpTmp - m_pSysExBuffer - dwStart;
```

Tell the renderer that we want it to use our material to render all future triangles. Note that we reset the render state to Gouraud, even though this is the default state.

We are now ready to render our triangle with the model. The Ramp model only seems to set palette colors once an instruction stream has been executed. Our code currently sets the palette on every End Scene. You may want to execute an instruction stream with just the D3DOP_STATELIGHT instruction to update the palette during an initialization stage.

16.4

16.3.5 How Does the Ramp Driver Perform?

Table 16-8 compares the performance of the RGB and the Ramp color model drivers. We've shown results for various rendering options using Scene 2 (16 × 5000) from our previous tests.

TABLE 16-8 Comparing the Direct3D RGB and Ramp Color Model Drivers

Rendering Option	RGB Model	Ramp Model
Gouraud	55.3 milliseconds	4.3 milliseconds
Flat Shaded	55.3 milliseconds	1.8 milliseconds
Gouraud and Specular	60.3 milliseconds	4.3 milliseconds
Gouraud and Dither	55.3 milliseconds	20.6 milliseconds
Texture Map and Gouraud	62.5 milliseconds	16.7 milliseconds
Texture Map Copy Mode	14.4 milliseconds	14.9 milliseconds

Wow!

- *Look at the speed of the Flat Shaded, Gouraud, and Gouraud and Specular options. Now we're really screaming along!*
- The performance of Gouraud and Dither is not too shabby either. You may not want to use it on all your triangles, but at this performance level, you could use it on some.
- The only "disappointment" is that the performance of texture mapping in Copy mode has not improved. It would have been great if we could use texture mapping widely, but at this performance level you probably would want to limit its use.

EXTRA CREDIT

We did not measure the cost of state changes midstream in an Execute Buffer. We also did not study the impact of texture size on texture mapping. These are excellent extra credit opportunities.

16.4 Optimizing Texture Mapping

Before we close, we'd like to include some advice from the Direct3D documentation on optimizing texture mapping:

- Texture mapping performance is heavily influenced by cache behavior. Keep textures small; the smaller the textures are, the better chance they have of being maintained in the main CPU's secondary cache.

- Do not change the textures on a per primitive basis. Try to keep polygons grouped in order of the textures they use.
- Use square textures whenever possible. Textures whose dimensions are 256 × 256 are the fastest. If your application uses four 128 × 128 textures, for example, try to ensure that all the textures use the same palette, and place them all into one 256 × 256 texture. This technique also reduces the amount of texture swapping required. Of course, you should not use 256 × 256 textures unless your application requires that much texturing, because, as already mentioned, textures should be kept as small as possible.

Well, we've come to the end of this road. Cheers, and may all your 3D applications really sizzle.

WHAT HAVE YOU LEARNED?

We measured the performance of our simple RGB color model triangle, both its inner workings and its various rendering options. We tried some optimizations and found that the returns were decent for long Execute Buffers, but overall performance was still far from stellar.

Next we learned how to use the Ramp color model driver, including using materials and D3DOP_STATELIGHT to direct the driver to create its lookup tables. And we were rewarded with a dramatic improvement in performance.

We've spent sufficient time on Direct3D's Immediate mode. In the next chapter we will cover mixing our 3D results with 2D and video.

WH
THIS CH

17.1

CHAPTER 17

Mixing 3D with Sprites, Backgrounds, and Videos

WHY READ THIS CHAPTER?

You might as well ask, "Why would I need to mix other graphics media types with 3D?" Well, here are some scenarios that might prompt mixing:

- You could create your application to be entirely 3D based. But 3D modeling and rendering is performance intensive. Drawing some objects with faster 2D mechanisms may bring an improvement in performance.
- You have your own object types, with their own rendering codes, and you want to intermingle these objects in a 3D model.
- Say you have designed 3D exploratorium within which you have real-life characters communicating with the Explorer. You have motion video footage of these characters, and you'd like to transparently overlay the video in your 3D world.

In short, you may want to mix media types because of performance advantages and/or because you want to add richness. In this chapter first you'll learn how to mix a 3D object within a 2D world, and then you'll learn how to use a video as a texture map within a 3D world.

17.1 Mixing a 3D Object on a 2D Background

We've already seen how mixing works in Part II, where we mixed a sprite on top of a background. In fact, over the course of Part II, we looked at a variety of options for mixing—using GDI, DirectDraw, and RDX.

In Part II we mixed a sprite on top of a background by:

- creating a CSurface from among the various options;

- creating a CBackground from among any options suitable to the CSurface and attaching the CBackground to the CSurface;
- creating a CSprite from among any options suitable to the CSurface and attaching the CSprite to the CSurface; and
- Blting the CBackground first, Blting the CSprite on top of the CBackground, and then refreshing the screen with the mixed image.

17.1.1 Our 3D Surface Is Also a 2D Surface

But wait! Let's think about where we are. We got access to a 3D surface in the first place by "querying" for 3D capabilities. As long as we retained access to the original 2D surface—that is, as long as we did not call *IDirectDrawSurface::Release()*—we can still use its innate 2D-ness.

So to mix a 3D object on top of a 2D background, we could

- create a CSurface suitable to be "extended" for 3D capabilities and then "extend" the 2D surface to a 3D surface while retaining access to the original 2D surface.
- create a 2D background from options suited to the 2D surface and then attach this background to the dual 2D-3D surface.
- create a 3D triangle from available render styles and then attach the 3D triangle to the dual CSurface.
- Blt the background first as usual, Blt the 3D sprite on top of the background, and then refresh the screen with the mixed image.

Here's the 3D version of the *FollowMouse()* method that handles dual surfaces:

```
long CSurface3d::FollowMouse(CPoint &point, int nTime)
{
```

Pre Scene Init: Set up to use 3D driver and clear Z-Buffer (if any).

```
    m_p3dFns->BeginScene();
    if (m_bIsZEnabled) {
        D3DRECT    drDst;
        drDst.x1 = 0;
        drDst.y1 = 0;
        drDst.x2 = m_dwWidth;
        drDst.y2 = m_dwHeight;
        #define nRECTS 1
        m_p3dViewport->Clear(nRECTS, &drDst, D3DCLEAR_ZBUFFER);
    }
```

Set up BLTPARAMS structure for dual-surface usage.

```
BLTPARAMS xDst;
xDst.pddsDesc = &m_SurfDesc;
xDst.pddsFns = m_p2dFns;
xDst.p3dFns = m_p3dFns;
xDst.p3dViewport = m_p3dViewport;
```

Blt background to dual surface. Blt either 2D or 3D background based on Init.

```
if (m_nNeedLock & BKGLOCK)
    m_p2dFns->Lock(NULL, &m_SurfDesc, DDLOCK_WAIT, NULL);
if (m_pBackground != NULL) {
    RECT rSrc = {0, 0, m_dwWidth, m_dwHeight};
    POINT ptDst = {0,0};
    m_pBackground->Blt(&xDst, &ptDst, &rSrc);
}
if (m_nNeedLock & BKGLOCK)
    m_p2dFns->Unlock(NULL);
```

2D background may need surface to be locked.

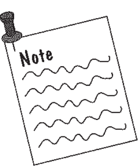
Blt attached 3D Triangle.

```
if (m_pTriangle != NULL)
    m_pTriangle->Blt(&xDst, &point);
```

Scene End Stage: End Scene, refresh screen, and return.

```
m_p3dFns->EndScene();
// offset dst rect accounting for client area
long lRight = m_ptZeroZero.x + m_dwWidth;
long lBottom = m_ptZeroZero.y + m_dwHeight;
RECT rDst = {m_ptZeroZero.x, m_ptZeroZero.y, lRight, lBottom};
RECT rSrc = {0, 0, m_dwWidth, m_dwHeight};
// set palette and refresh screen
gpPrimary->SetPalette(gpPalette);
gpPrimary->Blt(&rDst, m_p2dFns, &rSrc, DDBLT_WAIT, NULL);
// return
return TRUE;
}
```

Notice the code added to pass both the 2D and the 3D descriptor to the object renderers in the BLTPARAMS structure. Also notice the code added to lock and unlock the 2D surface for most 2D background rendering (a hardware-accelerated 2D background would not need a Lock/Unlock).



Some hardware 3D devices may not allow 2D functions to be invoked between *BeginScene()* and *EndScene()*. These devices will set the DDCAPS2_NO2DDURING3DSCENE flag in the 2D caps structure (hwCaps.dwCaps2). If this flag is set, you will need to modify the FollowMouse code to render a 2D background *before BeginScene()*, but render a 3D Background *after BeginScene()*. We found that the HEL drivers do not impose this restriction, so we have not built this check into our current example.

PART V

17.1.2 Measuring Background Performance

Table 17-1 compares the performance of Bltting a sprite with both 2D and 3D rendering paths.

TABLE 17-1 Comparing 2D and 3D Backgrounds

Rendering Path	Time
CBackgroundCCode	7.1 milliseconds
CBackgroundP5	6.8 milliseconds
CBackgroundTex	46.5 milliseconds
CBackground3D	3.8 milliseconds*

* CBackground3D fills the background with a constant color; whereas all other options transfer an image to the screen. Therefore the comparison of CBackground3D with the other options is not a true apples-to-apples comparison. The figure is shown for reference.

CBackgroundTex is an implementation of a texture-mapped 3D background object. You implement a texture-mapped 3D background by loading a texture object and setting its handle in the background material structure. Check the source code for the Timing Application on our Internet site. (Note that unlike triangle textures, a background texture need not be sized using powers of two.)

A CBackgroundTex is texture mapped to the surface and is not merely Bltted to the surface. The implication is that if the source and destination sizes differ, the source is stretched (or shrunk) to fit the destination rectangle. Texture mapping is much costlier, as the results of our measurements demonstrate.

If all you need is a simple Blt of a background image, then as the performance results indicate, using 2D backgrounds behind 3D objects offers significant performance boosts over using texture mapping.

17.2 Mixing in Sprites

Hey, can't we add sprites to our dual surface just like we did with backgrounds? Technically, yes. But our code lets us have only one active sprite at a time. If we wanted to have more than one sprite, we would need to maintain some form of list (or array) of sprites and draw all the active sprites within our Refresh functions.

Since the Intel RDX library provides code to manage lists of sprites and draw them in back-to-front order, let's just use RDX to mix sprites in. If you've forgotten, or haven't had a chance to play with RDX yet, do take a quick trip through Chapter 8.

17.2.1 Using RDX to Mix in Sprites

The RDX programming model allows us to

- create a surface of a specified size and pixel depth;
- create mixable objects (such as sprites, backgrounds, grids, and AV objects) and connect them to the surface;
- manipulate attributes of the objects (such as draw order, position, transparency, and visibility); and
- mix and render all visible objects attached to a surface by invoking a single *srfDraw()* function provided by the surface object.

Typically you attach the surface to a window using *srfSetDestWindow()*, and the window is automatically refreshed by *srfDraw()*. RDX also has a *srfSetDestMemory()* function that we can use to specify that the output of *srfDraw()* be sent to a memory buffer that we provide. Let's use *srfSetDestMemory()* to have RDX output its data into our dual surface:

```
long CSurface3d::FollowMouse(CPoint &point, int nTime)
{
    // pre-scene init
    m_p3dFns->BeginScene();
    if (m_bIsZEnabled) {
        D3DRECTdrDst;
        drDst.x1 = 0; drDst.y1 = 0;
        drDst.x2 = m_dwWidth; drDst.y2 = m_dwHeight;
        m_p3dViewport->Clear(1, &drDst, D3DCLEAR_ZBUFFER);
    }

    // setup BLTPARAMS struct for dual-surface usage
    BLTPARAMS xDst;
    xDst.pddsDesc = &m_SurfDesc;
    xDst.pddsFns = m_p2dFns;
    xDst.p3dFns = m_p3dFns;
    xDst.p3dViewport = m_p3dViewport;

    // Blt either 2D or 3D background to Dual-Surface
    if (m_nNeedLock & BKGLOCK)
        m_p2dFns->Lock(NULL, &m_SurfDesc, DDLOCK_WAIT, NULL);
    if (m_pBackground != NULL) {
        RECT rSrc = {0, 0, m_dwWidth, m_dwHeight};
        POINT ptDst = {0,0};
        m_pBackground->Blt(&xDst, &ptDst, &rSrc);
    }
    if (m_nNeedLock & BKGLOCK) m_p2dFns->Unlock(NULL);
```

PART V

Draw RDX objects by invoking srfDraw on the Dual Surface's m_hSurfmember.

```
if (m_bIsRdx) {
    m_p2dFns->Lock(NULL, &m_SurfDesc, DDLOCK_WAIT, NULL);
    srfSetDestMemory(m_hSurf, m_SurfDesc.lpSurface, m_SurfDesc.lPitch);
    srfDraw(m_hSurf);
    m_p2dFns->Unlock(NULL);
}

// Blt 3D triangle
if (m_pTriangle != NULL)
    m_pTriangle->Blt(&xDst, &point);

// SceneEnd
m_p3dFns->EndScene();
// offset dst rect accounting for client area
long lRight = m_ptZeroZero.x + m_dwWidth;
long lBottom = m_ptZeroZero.y + m_dwHeight;
RECT rDst = {m_ptZeroZero.x, m_ptZeroZero.y, lRight, lBottom};
RECT rSrc = {0, 0, m_dwWidth, m_dwHeight};
// set palette and refresh screen
gpPrimary->SetPalette(gpPalette);
gpPrimary->Blt(&rDst, m_p2dFns, &rSrc, DDBLT_WAIT, NULL);
return TRUE;
}
```

Lock DirectDraw Surface and pass its data pointer to RDX using *srfSetDetMemory()*. Then draw all objects using *srfDraw()*. RDX draws its objects directly onto the surface with or without transparency.

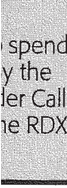
In the new *FollowMouse()* method that we have outlined above we are drawing our background first and then mixing in the RDX output (a composite of all the RDX objects). Finally we add in our 3D object on top of the RDX and background combo.

GENERIC OBJECT RENDERING

Our *FollowMouse()* method is becoming fairly long. We are starting to invoke multiple object renderers of various types. Wouldn't it be nice to have a long generic list of objects that you could invoke within a loop using a generic object render function?

You could redesign our classes, so that all objects such as backgrounds, Sprites, and RDX objects derive from a generic object. Next you could define the generic object to have a Blt function that takes an enhanced `BLTPARAMS` structure. You would then find yourself set up to invoke a generic Blt function within a loop, and the objects will take care of all the object-specific details. In fact, you would find yourself set up to the point that the various CSurfaceXXX-derived objects were only doing initialization and tear down.

Pretty neat, huh! We didn't do this, because then we would have had to spend time explaining *our* architecture, which is not the point of the book! By the way, this is pretty much how RDX works. In fact, take a look at the Render Callback effect (`FX_RENDER_CALLBACK`) if you want to tie in 3D objects into the RDX generic object architecture.



You'll probably point out that if we're using RDX, we can have our background be an RDX background (CBackgroundRDX) and not have to worry about any CBackground code either. That is true. Very astute of you! In fact, Table 17-2 has measurements of mixing the various 2D and 3D objects (the sprite measurements were for sixteen sprites of about 5,000 pixels each, and the background measurements were for a background of 734 × 475 pixels).

TABLE 17-2 Measuring Mixed 2D and 3D Objects

Backgrounds		Sprites/Triangles	
Rendering Path	**Time**	**Rendering Path**	**Time**
CBackgroundCCode	7.1 milliseconds		
CBackgroundP5	6.8 milliseconds		
CBackgroundRDX	4.0 milliseconds	CSpriteRDX	1.1 milliseconds
CBackgroundTex	46.5 milliseconds	CTriangleTex (Ramp/CopyMode)	16.2 milliseconds
CBackground3D	3.8 milliseconds	CTriangle3D (Ramp/Flat)	2.2 milliseconds

Following are some observations based on the results:

- The MMX technology optimizations that RDX has used for background drawing make CBackgroundRDX run at the speed of color filling. Wow! There is a clear benefit to mixing 2D and 3D.
- With Ramp mode triangles being rendered in the low-millisecond speeds, our Execute Buffer overhead starts becoming important again. These tests were performed with only sixteen triangles. It becomes worthwhile to invest in code for long Execute Buffers, when you are rendering many small triangles with the Ramp mode driver.
- Flat-shaded Ramp mode triangles compare well with spriting. However, texture mapping at 16 milliseconds (half the 30 fps budget) still takes quite a bit of time. A judicious mix of shaded and texture-mapped triangles would be the way to go. And, of course, using 2D sprites wherever possible is also a good way to go.

17.2.2 Adding RDX Objects at Front and Back

What if you want to add RDX objects behind and ahead of the 3D object? Well, RDX lets you create multiple surfaces. So you can solve this issue by creating two RDX surfaces and retaining one as the "behind" surface and

the other as the "ahead" surface. All objects attached to the behind surface, using *objSetDestination()* will get drawn behind the 3D object. And all objects attached to the ahead surface will get drawn on top of the 3D object. This is a simple extra credit exercise. Go on! Try it for yourself.

17.3 Mixing in Video

Mixing in video is a little more complex than mixing sprites or backgrounds. The following factors need to be considered:

- Video files are actually a series of images that need to be displayed sequentially. To mix 3D on top of video, we would need to mix our 3D image whenever a new video frame is drawn—lest we "lose" sight of our 3D object.
- A video file is recorded at a specific frame rate. Playback of frames in the video must be synchronized to a timer, so that they can be displayed at the recorded frame rate.
- Video files are usually recorded in high-color resolutions to capture the broad range of colors in natural situations. Video codecs prefer to choose their own palettes, since they reduce the color range for palletized displays. They typically produce very poor quality if they are forced to use a specified palette.

The issues of synchronized drawing and timed playback are dealt with in detail in Chapter 10. We will use the same code to mix our 3D sprite on top of a video object.

17.3.1 Handling Palettes

We do need to add some code to handle palettes. Our 2D objects use colors only from the system palette, and there is no palette conflict between 2D and video objects. But Direct3D uses more than the system palette. Let's look at the code needed to manage palettes among these media.

There is no fast and high-quality solution to sharing palettes. Our code shows you how to communicate palettes amongst video and 3D objects. Since video codecs don't like palettes to be forced on them, we have written our code to tell Direct3D to use the video object's palette.

Following is the code that takes a palette from a video object file and uses this palette with Direct3D surfaces. Note that Direct3D expects the palette to be set on the 2D surface before *any* 3D functionality is requested.

Create a palette object

```
LOGPALETTE *plogPalette;
PBYTE pTmp = new BYTE [sizeof (LOGPALETTE) + sizeof (PALETTEENTRY)*256];
plogPalette = (LOGPALETTE *)pTmp;
plogPalette->palVersion = 0x0300;
plogPalette->palNumEntries = 256;
if (!rdxGetVideoPalette(plogPalette)) return FALSE;
```

Use RDX to talk to a video object and get its palette

Change palette entry flags to not allow D3D to change any of them

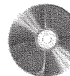
```
PALETTEENTRY *pPal = (PALETTEENTRY *) (pTmp + sizeof(LOGPALETTE));
for (int i = 0; i < 256; j++) pPal[i] .peFlags = D3DPAL_READONLY;
```

Querying for a palette from a video file takes a lot of steps. RDX simplifies these steps. So our code uses RDX to talk to the video codec. Refer back to Section 10.2 for an explanation of how to manage video with RDX. For quick reference, we've included here the essential code to query a palette from an AVI file using RDX:

```
BOOL rdxGetVideoPalette::GetPalette(LOGPALETTE *plogPalette, LPSTR *pFile)
(
     // first create a hFile object and load our AVI file
     err = hfilCreate(&m_hFil);
     macExitIfRdxError(err, FALSE);
     err = hfilLoad (m_hFil, pFile);
     macExitIfRdxError(err, FALSE);

     // create the AV object and initialize it with the video file
     err = avCreate(&m_hAV);
     macExitIfRdxError(err, FALSE);
     err = avAddVideoTrack(m_hav, m_hFil, 0, &m_hVid);
     macExitIfRdxError(err, FALSE);

     // get the palette from the video object
     DINORVAL err = vidGetPalette(m_hVid, pLogPalette);
     macExitIfRdxError(err, FALSE);
     return TRUE;
}
```

17.3.2 Using Video as a Texture Map

After seeing how to mix 3D and video on a DirectDraw surface, it is a fairly simple extrapolation to use video as a texture map. We merely modify our previous code to provide the Texture Map Address when we call *srfSetDestinationMemory()*.

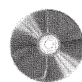


Run the demo for this chapter on the CD and check the Texture Mapped Video option.

WHAT HAVE YOU LEARNED?

By the end of the chapter, you should have learned how to

- mix a 3D object on a 2D background using Direct3D and DirectDraw;
- mix a 3D object with RDX sprites and a background (Direct3D, DirectDraw, or RDX);
- mix a 3D object on top of a video file where the video file is played through RDX and can be either VFW or ActiveMovie based; and
- make the simple modification needed to use video as a texture map source (that is, if you perused the source code on the CD).



You've reached the end of our 3D coverage. We hope you have learned a lot.

P

WE'D
RICK
GOTTL

PART VI

Processors and Performance Optimization

WE'D LIKE TO EXTEND AN ACKNOWLEDGEMENT TO FRANK BINNS, SHUKY ERLICH, BRUCE BARTTLET, JULIE A BRAJENOVICH, K. SRIDHARAN, RICK MANGOLD, BOB FABER, BEV BACKMAYER, DEBBIE MARR, BOB REESE, TOM WALSH, MICKEY GUTTMAN, BENNY EITAN, KOBY GOTTLIEB, ODED LEMPEL, AND DAVID BISTRY

Chapter 18 **The Pentium Processor Family**

- Basic processor terms
- Overview of Pentium and Pentium Pro processors
- Overview of MMX technology
- Identifying processor models and features using *CPUID*

Chapter 19 **The Pentium Processor**

- Detailed overview of processor components
- Instruction pipelining
- Integer pairing and scheduling rules
- Address Generation Interlock (AGI)
- Branch prediction
- How to optimize the sprite sample

Chapter 20 **The Pentium Processor with MMX Technology**

- MMX architecture
- Instruction set and data types
- EMMS usage guideline
- Saturation versus wraparound
- MMX pairing and scheduling rules
- Optimizing the MMX sprite
- Using scheduling rules to optimize the sprite

Chapter 21 **VTune and Other Performance Optimization Tools**

- VTune's coverage of pairing and scheduling rules
- Static and dynamic analysis
- Hot-spot monitor and time-based and event-based sampling
- VTune usage hints
- ReadTime StampCounter—RDTSC
- Using the PMonitor library

Chapter 22 **The Pentium II Processor**

- Architectural overview and new features
- Pentium II performance counters
- MMX pairing rules
- Detailed component description including event counters for each unit
- Write Combining memory type to speed graphics performance
- Branch mispredictions, partial stalls, and the 4:1:1 decoder template

Chapter 23 **Knowing Your Data and Optimizing Memory**

- Overview of memory subsystem
- Differences between Pentium and Pentium Pro member processors
- Cache differences, DCU splits, partial memory stalls
- MMX stack alignment
- Accessing cached memory
- Writing to video memory

When it comes to developing multimedia applications, you'll quickly realize that you're dealing with a huge amount of data—most of which is typically used once or twice and then thrown away. Unlike database, word processing, or transaction-based applications, multimedia applications must quickly display a sequence of pictures to give the illusion of motion; they must pump audio data in real time to play uninterrupted sound sequences; or they must render a 3D model to give the illusion of a 3D world. There are lots of calculations to make, lots of data to move around. In order to get smooth motion video, audio, and 3D, you still have to fine-tune your applications for the platform they are running on.

We decided to include this section because we believe that multimedia applications and processor optimization go hand in hand—at least for now. Some developers think optimization is an art; some think it's a science. We think it's a mix of both.

First we cover the Pentium family processors, their architecture, and how they work with code and data. We optimize our sprite sample for each of the processors we cover—the Pentium processor, the Pentium processor with MMX technology, and the Pentium II.

When you think about optimizing multimedia applications, don't just think about applying the optimization rules of the processor (pairing, AGIs, register contentions, and so forth);[1] you should first think about your data access pattern. Optimizing for the processor is most useful when you access the data in the L1 cache. From our experience, you should not try to squeeze every cycle out of your code; you need only focus on the code segments that are called very often and those that consume most of the CPU cycles.

Rather than telling you how to optimize your code, we'll tell you how we go about optimizing ours. Once the code is written, we typically use Intel's VTune to figure out how to schedule instructions for optimal pairing and how the code behaves when it runs on the PC—that's the science part. Since VTune does not know how to fix the code for us, we use our knowledge of the processor scheduling rules and rearrange the code for optimal pairing—that's the art part.

We typically start with VTune's static analyzer, which helps us figure out how to schedule the instructions for the specific processor we're optimizing for. Once the code is operational, we run it with VTune's hot-spot system monitor. It tells us how much of the CPU the application is using and which pieces of code are consuming most of the time. We then zoom in on these segments and try to optimize them even more—if possible.

To learn more about the behavior of a particular section of code, we run VTune's dynamic analyzer. With the dynamic analyzer we can collect an exact execution trace of certain sections of code and then analyze the traced instructions. It gives us information about branching, L1, L2, and cache hit rate, unaligned accesses, and many other things. With dynamic analysis, we get a better understanding of the behavior of our code.

As a general rule, we pay special attention to memory accesses when we write our code. We always try to guarantee that data that we want to use is already in the L1 cache when it is time to access it. We do this by fetching data ahead of time, by operating on a smaller subset of data at one time, or by changing our data access pattern.

We also pay special attention to branches—especially with the Pentium Pro and Pentium II processors. If we can avoid a branch instruction, we do. If not, we use the dynamic analyzer or the processor event counters to figure out how often we miss branches—then we see if we can do better by rearranging the branch logic.

1. The terms *pairing, AGI,* and *L1, L2 caches* are defined in Chapter 18.

CHAPTER 18

The Pentium Processor Family

WHY READ THIS CHAPTER?

You must be familiar with the Intel Inside® logo. But do you really know what's inside? Do you really know how the Pentium processors work? The real question is, "Do you want to know?" Come along, we'll take you on the grand Pentium processor tour.
In this chapter, you will

- be introduced to terms and concepts used throughout this part of the book;
- get an overview of the four Intel processors: the Pentium, Pentium with MMX technology, Pentium Pro, and Pentium II; and
- learn how to identify the different processors and detect their model-specific features using *CPUID.*

As of today, there are four major Pentium processors: the Pentium, Pentium Pro, Pentium processor with MMX technology, and, just recently, the Pentium II processor. Although the internal architecture of each processor differs from the others, all the processors are 100 percent compatible with the Intel 486 instruction set. As a law of evolution, newer processors offer new architectural features and new instructions to enhance the performance of existing applications, and to enable new classes of applications.

In this chapter we'll give you a brief overview of each of the processors without going into too many details. If some of the terms or concepts are not clear, refer to the chapter that covers that specific processor. Chapters 19,

20, and 22 give detailed information about the architecture of the internal components of each of the Pentium processors: one chapter for the Pentium processor, one for MMX technology, and one for the Pentium II processor. We don't cover the Pentium Pro because its features are a subset of the Pentium II processor.

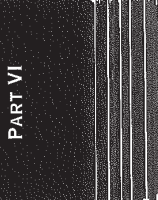
We also devote one chapter to VTune and other performance optimization tools that make it easier for you to optimize your code, and, finally, in the last chapter of the book we discuss memory optimization issues and techniques related to memory, the caches, and the system bus.

18.1 Basic Concepts and Terms

Before we delve into too much detail, let's review some concepts and terms relevant to processors in general and to Intel Architecture (IA) processors in particular.

- *L1 Cache.* First level cache. The L1 cache is on-chip static memory that can provide data in 1 clock cycle on a cache hit. A cache hit occurs when the requested data is already in the cache; otherwise you get a cache miss, and the data is brought in from main memory or the second level cache (L2).
- *L2 Cache.* Second level cache. Typically the L2 cache is off-chip static memory that runs more slowly than L1 cache. Some Pentium Pro processor models have the L2 cache on the chip running at the speed of the processor core. The L2 is typically much larger than the L1 cache (256K–1 MB). The L2 cache has a cache miss/hit behavior similar to that of the L1 cache.
- *Cache line.* A cache line describes the smallest unit of storage that can be allocated in the processor's L1 cache; that is, when you read a byte or more from main memory, the entire cache line is burst into the L1 cache from the memory location where the bytes are read from. Cache lines are typically aligned on a byte boundary equal to their width. For example, all four Pentium processors have a 32-byte cache line aligned on a 32-byte boundary. If a read crosses the cache line boundary, the processor brings in both cache lines (64 bytes) from the memory location where the bytes are read from.
- *Fetch.* The process of loading raw opcodes from the cache or memory into one of two prefetch buffers inside the processor.
- *Decode.* The process of parsing and interpreting the raw opcodes. In the Pentium Pro processors, the IA instructions are decoded into micro-opcodes.

- *Writeback.* The process of committing the final results to IA registers, the cache, or main memory.
- *Micro-op.* The decoder in the Pentium Pro and the Pentium II processors breaks the IA instructions into micro-instructions (*micro-op codes*). These micro-ops are necessary for the *out-of-order execution* model used in these processors.
- *Out-of-order execution.* Pentium Pro processors execute micro-ops based on the readiness of their data rather than the order in which they entered the execution unit. This is out-of-order execution.
- *Branch Target Buffer (BTB).* The Branch Target Buffer holds a history of branches that were mispredicted during the execution of an application. It stores the address of the mispredicted branch instruction, the branch target address, and the result of the misprediction. When the same instructions show up again (in a loop for example), the branch prediction unit uses this information to predict the outcome of the branch.
- *Return Stack Buffer (RSB).* The Return Stack Buffer can correctly predict return addresses for procedures that are called from different locations in succession. The RSB is useful for unrolling loops that contain function calls, and it removes the need to in-line procedures called inside the loop.
- *U and V pipes.* The Pentium processor has two execution pipelines that operate in parallel and can sustain an execution rate of up to two instructions every clock cycle. These two pipes are known as the U and V pipes.
- *Pipelining.* The process of overlapping operations in the processor pipeline is called pipelining. As a result of breaking the instruction execution into multiple stages (fetch, decode, execution, and writeback), the processor can execute multiple instructions at the same time—each in a different execution stage. For example, one instruction could be in the prefetch stage, one in decode, one in execution, and one in writeback. This process is very similar to the assembly line at automobile plants, where one person installs a front door, another person installs the back door, and someone else installs the headlight, and so on. All of them are working at the same time but on different cars. Analogously, in the processor, all units are working at the same time but on different instructions.



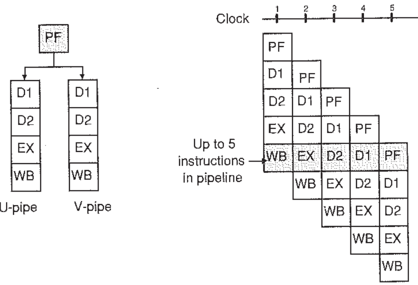
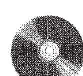
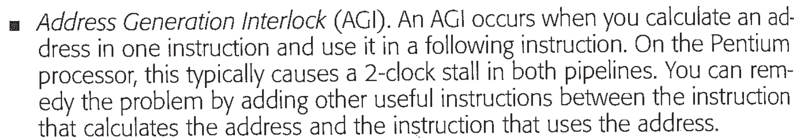


FIGURE 18-1 The Pentium processor pipeline can execute up to five instructions at any one time.

- *Superpipelining.* This is the same thing as pipelining except that the pipeline is deeper. For example, the Pentium processor is pipelined since it has five stages; the Pentium Pro and Pentium II are superpipelined because they have twelve execution stages.

- *Pairing.* Two instructions pair if and only if the first instruction can execute in the U pipe and the second instruction can execute in the V pipe. The pairing rules exist because only the U pipe can execute complex instructions, therefore, we need to ensure that a complex instruction is followed by a simple instruction. In the Pentium processor, you can have both the U and V pipes executing instructions as long as they adhere to the Pentium pairing rules. You can find the Pentium pairing rules in later chapters or in the *Intel Architecture Optimization Manual* found on the companion CD.





- *Address Generation Interlock* (AGI). An AGI occurs when you calculate an address in one instruction and use it in a following instruction. On the Pentium processor, this typically causes a 2-clock stall in both pipelines. You can remedy the problem by adding other useful instructions between the instruction that calculates the address and the instruction that uses the address.

  In the Pentium Pro and Pentium II processors, even though AGIs can theoretically happen, they are not as obvious or obstructive because of the out-of-order execution model that these processors use. So, basically, you don't have to worry about AGIs occurring in these two processors.

- *Partial stalls.* Partial stalls occur when you load a small register *(al)* and follow it with an access (read/write) to the larger register *(ax or eax)*. These occur only with Pentium Pro and Pentium II processors. Depending on the status of the pipeline, partial stalls can be extremely taxing on application performance. Removing partial stalls is typically one of the major optimizations that you can get from the Pentium Pro processors. More about this in a later chapter.

18.2 The Pentium Processors

Let's look at an overview of the Pentium and Pentium Pro processors and their MMX technology counterparts (see Figure 18-2).

18.2.1 The Pentium Processor

The Pentium processor marked a significant step over its predecessor, the Intel 486. It uses two parallel execution pipelines—U and V—which make it possible for the processor to execute up to two instructions in parallel. Each pipeline is divided into five execution stages, so the execution of up to five instructions can be pipelined (or overlapped) at any given cycle (see Figure 18-3).

The Pentium processor has also improved the performance of floating-point operations drastically and separated the instruction and data L1 caches, so the processor can fetch instructions and access data all within the

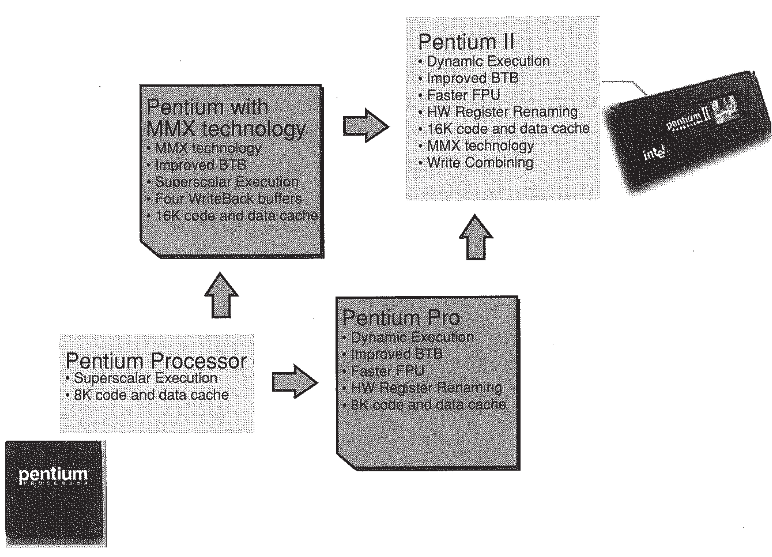


FIGURE 18-2 Evolution of the Pentium processor family.

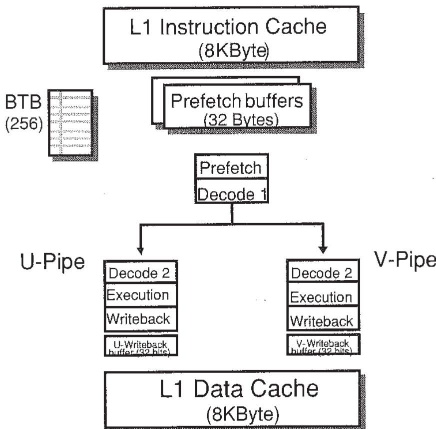


FIGURE 18-3 An architectural view of the Pentium processor.

same clock cycle. It includes two dedicated[2] 32-bit Write buffers, which make it possible for the processor to queue memory writes without stalling the execution of instructions within the processor.

Finally, the Pentium processor includes two prefetch buffers of 32 bytes each (one cache line each). With these prefetch buffers the processor can prefetch instructions from two different execution paths: one fetches from the next consecutive instruction address, and one fetches speculatively from a branch target address. The prefetch buffers, in conjunction with the Branch Target Buffer (BTB), help the prediction unit to speculate on the outcome of previously encountered branches.

18.2.2 The Pentium Pro Processor

Intel then introduced the Pentium Pro processor. Instead of a five-stage pipeline, the processor moved to a decoupled twelve-stage superpipelined architecture with in-order execution at both ends of the pipeline, and out-of-order execution in the middle. With the out-of-order execution capability, the processor can speculatively process instructions out of sequence. This capability is extremely useful when an instruction stalls while the processor waits for data to be read from memory or waits for the result of an earlier operation to be available. (See Figure 18-4.) The in-order units, in the front

2. The U pipe can only queue data in the U Write buffer, and the V pipe can only queue data in the V Write buffer.

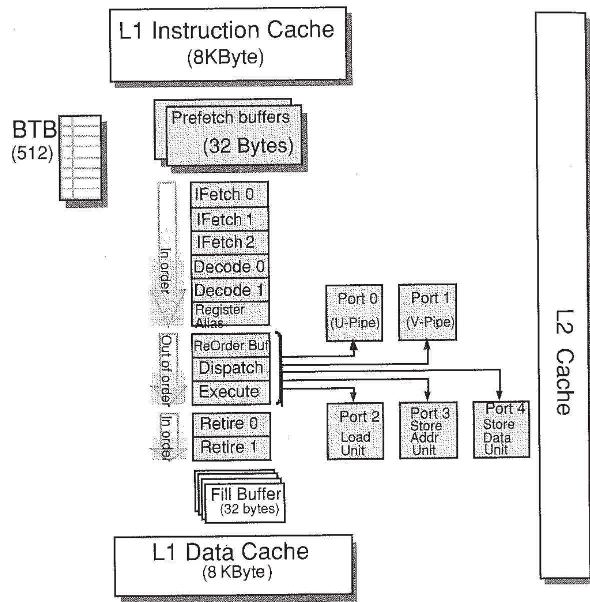


FIGURE 18-4 An architectural view of the Pentium Pro processor.

end and the back end, guarantee that instructions maintain the same sequence when they enter and when they exit the execution unit.

In addition to the two execution U and V pipelines (Port 0 and Port 1), the Pentium Pro processor added three more units: Port 2 loads data from the cache or memory, Port 3 calculates store addresses, and Port 4 stores data to the cache or memory.

Similar to the Pentium processor, the Pentium Pro maintains separate L1 data and instruction caches of 8K each. But the L2 cache has been moved inside the chip to provide faster access to data in the L2 cache. The processor also doubles the size of the BTB to 512 entries in order to improve the branch prediction rate. With a bigger BTB, more mispredicted branch instructions can be remembered just in case they get encountered later.

18.2.3 The Pentium Processor with MMX Technology

In 1997 Intel introduced the Pentium processor with MMX technology, which adds fifty-seven new instructions to the Pentium instruction set. MMX technology is geared to multimedia applications. The size of the L1 caches is doubled, and the number of Write buffers is increased to four.

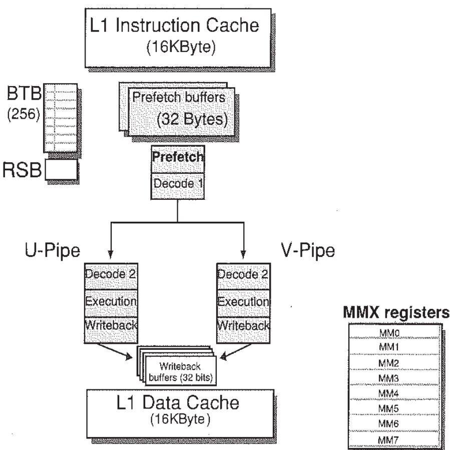


FIGURE 18-5 Pentium with MMX technology processor.

Since this processor is based on the Pentium processor, the internal architecture is identical except for a few changes. The size of the L1 caches is doubled to 16K each, and the number of 32-bit Write buffers is increased to four. A Return Stack Buffer (RSB) has been added; it can correctly predict return addresses for procedures that are called from different locations in succession (see Figure 18-5).

Finally, the number of 32-bit Write buffers is doubled to four undedicated buffers in this processor. Unlike the Pentium processor, which had a dedicated Write buffer for each pipe, in this processor the U and V pipes can write to any of the four Write buffers. This setup is beneficial when there are many writes to uncached memory.

18.2.4 The Pentium II Processor

The Pentium II processor was just released in the middle of 1997. It is basically a Pentium Pro processor with MMX technology. Similar to the Pentium with MMX technology, the Pentium II processor contains fifty-seven MMX instructions and eight MMX registers. The capacity of the L1 data and instruction caches has been doubled to 16K each, and the Return Stack Buffer (RSB) has been added. (See Figure 18-6.)

Unlike the Pentium Pro processor, the Pentium II moved the L2 cache off the chip. Notice that in the Pentium Pro processor, the L2 cache runs at the same speed as the core; however, in the Pentium II processor, the external L2 cache runs at half or one third the speed of the processor core. You'll find out more about this facet of the processor in the Pentium II chapter.

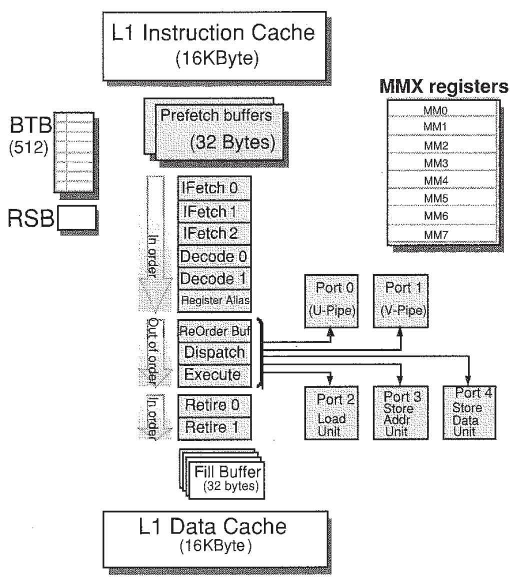


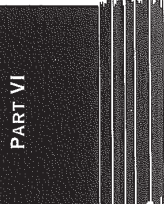
FIGURE 18-6 Architectural view of the Pentium II processor.

18.3 Identifying Processor Models

As the Intel architecture evolved with new features, Intel realized that it was essential to provide a simple way for software to identify the availability of such features. Starting from the Intel 386 processor, Intel provided a signature at processor reset. Later Intel added a special instruction, *CPUID*, so that applications could identify features related to a specific processor model.

The signature of the processor includes the vendor ID, model, and stepping. It also specifies whether certain features of the processor are supported; for example, MMX technology, `CMOVxx`, and `FMOVxx` instructions. In the Pentium Pro processor, *CPUID* also returns information about the organization of instructions and data caches.

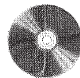
Let's see how you can use *CPUID* to figure out whether or not MMX technology is supported on a certain processor. First you need to check whether the processor supports the *CPUID* instruction. An Intel processor supports the *CPUID* instruction if you can change bit 21 of the *eflags* register. The following code snippet checks whether *CPUID* is supported.



```
BOOL CpuIdSupported()
{
    BOOL fSupported;

    _asm {
        // Try to change bit 21.. save a copy of it in ecx.
        pushfd                          // Push EFLAGS to stack
        pop     eax                     // EAX=EFLAGS
        mov     ecx, eax                // save it for later comparison
        and     ecx, 200000h            // isolate bit21
        xor     eax, 200000h            // change bit21
        push    eax                     // push it on stack
        popfd                           // pop it to EFLAGS

        // Now see if it changed
        pushfd                          // Push new value of EFLAGS
        pop     eax                     // EAX=new EFLAGS
        and     eax, 200000h            // isolate bit21
        xor     eax, ecx                // compare it to last value
        mov     fSupported, eax         // EAX==0 if did not change
    }

    return (fSupported != 0);
}
```

After you have determined whether or not the *CPUID* instruction is available, you can use it to figure out if MMX technology is supported. MMX technology is supported only if bit 23 of the feature flag is set. You can obtain the feature flag by calling *CPUID* with the *eax* register set to 1.

```
BOOL MMXSupported()
{
    BOOL fSupported;

    _asm    {
        mov     eax, 1                  ; CPUID level 1
        CPUID                           ; EDX = feature flag
        and     edx, 0x800000           ; test bit 23 of feature flag
        mov     fSupported, edx         ; 0: not supported, !0: supported
    }

    return (fSupported != 0);
}
```



To give you a head start, we included a simple Dynamic Link Library on the CD that performs these operations for you. It returns information about the processor model, starting from the Intel 386, and enumerates all the information supported by the *CPUID*. You can find all the sources, binaries, and documentation on the companion CD.

WHAT HAVE YOU LEARNED?

Here is a recap of the points you will need to remember from this chapter as you read the following chapters:

- The Pentium processor implements a five-stage pipeline capable of decoding two instructions per clock.
- The Pentium Pro processor implements a twelve-stage, three-way superpipeline.
- Intel added the MMX technology to both processors, which are targeted toward multimedia applications.
- The Pentium processor has two dedicated Write buffers, the Pentium with MMX technology processor has four shared Write buffers, and the Pentium Pro processor has four 32-byte Fill buffers.
- The Pentium processor suffers AGI stalls, and the Pentium Pro does not.

CHAPTER 19

The Pentium Processor

WHY READ THIS CHAPTER?

In the previous chapter, we gave you an overview of the Pentium processor family. In this chapter, we'll peel the top off the Pentium processor and have a peek inside at the components. Then we'll delve into getting better performance from the components.

In this chapter you'll

- get a better understanding of the components of the Pentium processor, including the L1 cache, prefetch buffers, branch prediction unit, BTB, the U and V pipelines, and the Write buffers;
- learn the benefit of instruction pipelining and how to burst empty bubbles in the pipeline;
- learn the Pentium integer pairing and scheduling rules;
- see how to avoid Address Generation Interlock (AGI) stalls;
- look at the importance of branch prediction and the problems that come with misprediction;
- get an analysis of our earlier sprite sample and see how you can rearrange instructions to reduce the amount of cycles it takes to execute the sprite with this processor.

The goal of this chapter is to show you how to optimize your code to achieve optimal performance on the Pentium processor. To do that, you first need to learn about the internal components of the processor and how to extract the most out of them. For each component we'll give you a brief operational overview and then provide a few suggestions for gaining optimal operation of that component.

VTune can easily analyze your code and show you how well your instructions pair.

In this chapter, you'll learn about the L1 data and instruction caches, the prefetch unit, the BTB, the U and V execution pipelines, and the Write buffers. You'll also be introduced to the Pentium pairing rules that must be followed to achieve high application performance. Finally, you'll learn about AGIs and how to resolve them.

At the end of the chapter, we rewrite the sprite sample, from Part II, "Sprites, Backgrounds, and Primary Surfaces," in assembly language. We then show you how to use the Pentium pairing and scheduling rules to improve the performance of the sample.

19.1 Architectural Overview

The Pentium processor includes a set of features that enables it to sustain an execution rate of up to two instructions every clock cycle. These features include a five-stage pipelined architecture, dual execution pipelines (U and V), separate instruction and data L1 caches, two Write buffers, instruction prefetching, and branch prediction (see Figure 19-1).

In order to sustain a high execution rate, you must first understand how these components work and how to mold your code to satisfy their constraints. For example, you cannot assume that you have an unlimited instruction cache, so it would be best to fit your inner loops into an 8K block.

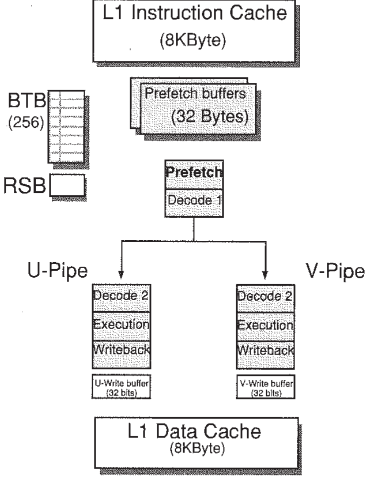


FIGURE 19-1 Pentium processor architectural diagram.

In the following sections, you'll get a detailed look into each of the Pentium processor features and understand what you can do to make them work more efficiently.

19.2 Instruction and Data L1 Caches

19.2.1 Operational Overview

The L1 cache is on-chip static memory that satisfies internal read/write requests more quickly than an external bus cycle to memory can. In addition, the L1 cache reduces the processor usage of the external bus, thus allowing other devices—DMA, bus maters, and so forth—to move data on the bus.

The Pentium processor has two independent L1 caches; one satisfies data accesses, and the other satisfies instruction fetches. The two caches exist on two separate internal buses (each bus is 64 bits wide), so the processor can load instruction and data in the same clock cycle. The Intel 486 can only load data or instructions at any given moment because its instruction and data share one L1 cache.

The Pentium Pro exhibits different behavior on cache write misses.

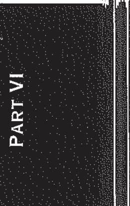
Both the instruction and the data L1 caches are divided into 32-byte cache lines—this is the minimum granularity of the L1 cache. When the processor transfers data between the L1 cache and the external bus (main memory or the L2 cache), it transfers a minimum of one cache line at a time.

On a read or write hit, the L1 cache satisfies the request in 1 clock cycle. On a read miss, the processor bursts an entire cache line into the L1 cache. If a multi-byte read crosses a cache line boundary, the next consecutive cache line is also brought into the L1 cache. On a write miss, the Pentium writes the data directly to the L2 cache or to main memory.

19.2.2 Performance Considerations

To put it simply, "Reuse it while it's in the L1 cache." If you have already brought in code or data from main memory to the L1 cache, make sure that you use it while it's still there—before it gets flushed out. Following are a few suggestions to accomplish this task.

- *Keep the size of your inner loops below 8K.* If your most executed loop does not fit in the L1 code cache, the L1 cache will thrash continuously. To fix this problem, you can break the task at hand into smaller tasks with



smaller loops that fit within the L1 cache. To find out the size of your loop, you can either look into the map file generated by the linker or use VTune's static analyzer (see Chapter 21).

You should also watch out for in-line macros and functions that, if used often, could bloat the size of your code.

- *Reuse data while it's in the L1 cache.* If possible, operate on the data while it's in the L1 cache. Since multimedia data does not typically fit in the L1 cache, you can operate on part of the data at one time rather than the full set. For example, instead of decoding the entire video frame in one loop, you can decode the top half of the frame from start to finish and then the bottom half—or whatever size fits in the L1 cache.
- *Allocate data ahead of time.* As we mentioned earlier, on a read miss, the Pentium processor brings in an entire cache line to the L1 cache. Once the requested data is available, it is forwarded immediately to the requesting instruction for processing. The processor then reads the remainder of the cache line.

  Now, while the cache line is being brought in, if another instruction accesses uncached memory or causes another read miss (from another line), the second instruction will stall until the entire cache line is completely brought in. But if the second instruction accesses data that's already in the L1 cache, the instruction executes normally.

  Accordingly, you could possibly achieve better performance if you could bring in data into the L1 cache before you're ready to use it—allocating data ahead of time.

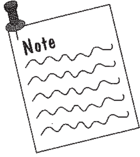
  For example, assume you're processing two buffers, A and B, sequentially and that you're processing one cache line every iteration of the loop as shown below:

```
ToOfLoop:
  Read cache line A[i]
  Read cache line B[j]    <- Waits for A[i] to be completely brought in.
  Process cache lines A[i] and B[j]
  Increment i and j by one cache line
Goto TopOfLoop
```

So before you can process the two cache lines, you have to wait for the entire A cache line and some of the B cache line to be brought in—and the same thing happens for every iteration of the code. You can rearrange the code in such a way that you can interleave bringing in the data to the L1 cache with some useful operations.

In the following code we read the first two cache lines outside the loop and then we wait for both of them to finish. At the top of the loop, rather than processing the two cache lines, we allocate the A cache line for the next iteration ahead of time. While the cache line is being brought in, we do some processing on the first two cache lines—they're already in the L1 cache. We then read the B cache line for the next iteration and then finish processing the first two cache lines. By the time we get back to the top of the loop, we should have the next A and B cache lines waiting in the L1 cache—so we accomplish the same operations without the wait.

```
Read cache line A[i]
Read cache line B[j]

ToOfLoop:
  Read cache line A[I+32]                ◊ Pre-allocate for next iteration.
  Process some of A[i] and B[j]          ◊ Process them from L1 cache.
  Read cache line B[j+32]                ◊ Pre-allocate for next iteration.
  Process remainder of A[i] and B[j]     ◊ Process them from L1 cache.
  Increment i and j by one cache line
GoTo TopOfLoop
```



One of the major enhancements in the Pentium Pro processor is the Nonblocking Read feature. The Pentium processor stalls completely when two back-to-back read misses occur. The Pentium Pro, on the other hand, allows other instructions to execute while it's waiting for data to be brought into the L1 cache.

19.3 Instruction Prefetch

19.3.1 Operational Overview

The Pentium processor includes a prefetch unit that is capable of fetching unaligned instructions and instructions split between two cache lines without any penalty. It features two 32-byte prefetch buffers that operate in conjunction with the Branch Target Buffer (BTB) to fetch raw opcodes from the cache or main memory (see the discussion on BTB below). One prefetch buffer fetches instructions sequentially; the other fetches instructions speculatively, according to the branch history in the BTB. Notice, however, that only one of the prefetch buffers is active at any given time.

Prefetches are requested sequentially until a branch instruction is fetched. When a branch instruction is fetched, the address of the instruction is looked up in the BTB, and if it is found, the behavior history of the instruction is used to determine its outcome—taken or not taken. If the branch is predicted as not taken, prefetches continue with the next sequential instruction; otherwise the other prefetch buffer is directed to start fetching from the branch target address—as if the branch will be taken.

The actual outcome of the branch is only determined when the branch instruction is executed. If the branch was mispredicted, both the U and V instruction pipelines are flushed, and prefetching activity starts all over.

19.3.2 Performance Considerations

In reality, there are no special considerations for the prefetch unit in the Pentium processor. The following general guidelines are helpful, although they will be of more use for the Pentium Pro processor.

- Align loops, branch, and function labels on 16-byte boundary.
- Keep infrequently executed code separate from inner loops, such as initialization code and error handlers, so that it will not be prefetched and decoded unnecessarily.
- Do not interleave data with code, such as jump tables, because you don't want the data to be prefetched and decoded unnecessarily.

(We'll discuss improving the performance of the prefetch unit in more detail in the Pentium II chapter.)

19.4 Branch Prediction and the Branch Target Buffer

19.4.1 Operational Overview

The Pentium processor includes a branch prediction unit (BPU), which predicts the outcome of branch instructions when they are first decoded. What's important here is that the processor take the prediction seriously and start executing instructions from the predicted address—until it finds otherwise when the actual branch result is determined. When a branch instruction is mispredicted, the processor saves the address of the instruction and the correct path (taken or not taken) in the Branch Target Buffer (BTB), which is simply a lookup table with 256 entries.

How does the BTB work? When the BPU encounters a branch instruction, it looks up the address of the instruction in the BTB. If it finds the address, the BPU looks at the history of this instruction and determines whether or not the branch should be taken. If the instruction was taken before, the BPU assumes that it will be taken again, and if not, the branch won't be taken. This is called *dynamic prediction.* If the branch is predicted taken, the BPU directs the prefetch unit to fetch raw opcodes, from the predicted branch address, into the second prefetch buffer.

The Pentium Pro's static predictions are slightly different from those of the Pentium processor.

If the BPU does not find the branch instruction in the BTB, the Pentium processor assumes that the branch will not be taken and that execution will continue sequentially with the next instruction. This is called the *static prediction.*

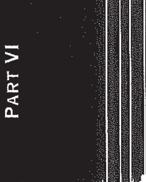
19.4.2 A Closer Look at the BTB

If you're as unlucky as I am, you probably get a ticket when you're caught speeding. The next time you get caught speeding, the officer can easily look up your record and will probably give you a bigger fine. Now, if you're a good citizen—or you just never get caught—you won't have such a record.

The BTB works in a similar fashion; it only keeps a record of mispredicted branch instructions. When an instruction is mispredicted, the instruction is "ticketed" and a record of it is kept in the BTB. The address of the instruction, the target branch address, and the result of the branch are recorded in the BTB (Figure 19-2). The next time any instruction comes through, its address is matched against the instruction address in the BTB. If the address is found, the outcome of the instruction is predicted based on the "taken/not taken" flag in the BTB. If the instruction is predicted as taken, the prefetch unit is directed to fetch instructions from the target address in the BTB (see Figure 19-2).

	Instruction Address	Target Address	Taken or Not
1	80001000	80001D00	Taken
2	80003001	None	Not
3	8000D000	80002C00	Taken
	: : :	: : :	: : :
	: : :	: : :	: : :
256			

FIGURE 19-2 BTB structure.



19.4.3 Performance Considerations

As we mentioned earlier, the actual outcome of a branch is only determined when the instruction is executed—in the execute stage. If the branch instruction was predicted correctly, the processor continues on its merry way. If the branch instruction was mispredicted, the processor flushes both pipelines and starts fetching from the correct address. As a result, the processor is stalled until the correct sequence of instructions is fetched and fed to the decoder unit.

You can determine how long it will take the processor to execute branch instructions, assuming that instruction opcodes are already in the L1 cache. Here's how the process works.

Pentium Pros exhibit a different behavior for backward branches not found in the BTB.

Branch instructions not found in the BTB are assumed not taken. Notice that this includes unconditional branches: if they're not in the BTB, they're assumed not taken. Why? As you recall, the BPU makes its prediction in the first decode stage of the pipeline. At that stage, the BPU does not know the branch target address of the instruction if the unconditional branch instruction is not in the BTB—because the instruction has not been fully decoded yet. This case is highlighted in Table 19-1.

Use Table 19-1 to determine how many clocks it takes to execute a branch instruction. Notice that the table assumes that the instructions of the correct branch address are already in the L1 code cache. If the instructions aren't in L1, it takes much longer to fetch the instructions from the L2 cache or main memory.

TABLE 19-1 Pentium Processor Branch Behavior

	Predicted	Unconditional	Conditional
Direct	Correctly	1	1
	Incorrectly	3	3U/4V
Indirect	Correctly	2	2
	Incorrectly	4	4U/5V

Now that you know how the branch prediction unit and the BTB operate, we'll leave you with a few suggestions that could help you minimize branch mispredictions in your code:

19.5

- *Minimize branch misprediction.* You can use VTune dynamic analyzer or the internal Pentium performance counters to determine if you have a high rate of branch mispredictions and to pinpoint the guilty routines. Armed with this information you can rearrange your code for better branch prediction.
- *Try to fit code with high branch misprediction in the L1 cache.* As we mentioned earlier, it only takes 3–4 cycles to recover from a branch misprediction if the correct target address is in the L1 cache. But if the mispredicted branch address is in the L2 cache or main memory, the penalty for branch misprediction is much higher. Refer to the "L1 cache" section for more information about the L1 cache.
- *Avoid loops with a huge amount of mispredicted branches.* A huge number of mispredicted branches will thrash the BTB, since it can hold only the last 256 mispredicted distinct instruction addresses. As a result, the next time the loop comes around, no history of the mispredicted instructions will exist, and as a result you could have a high branch misprediction rate.

19.5 Dual Pipelined Execution

19.5.1 Operational Overview

The Pentium processor includes two execution pipelines (U and V), which can execute two instructions in parallel (Figure 19-3a). Each pipeline is divided into five execution stages, which allow for overlapped execution of different instructions at any given time (Figure 19-3b).

At its maximum capacity, each pipeline can operate on up to five instructions at any given time, or a total of up to ten instructions in both pipelines (Figure 19-3c). Notice that although the two pipelines can operate on that many instructions at any given time, they can sustain only up to two instructions per clock cycle. The fact is, without pipelining, each instruction would require at least 5 clocks to complete. Because of pipelining, the Pentium processor can operate on five instructions at any given time and sustain an execution rate of up to two instructions per clock cycle.

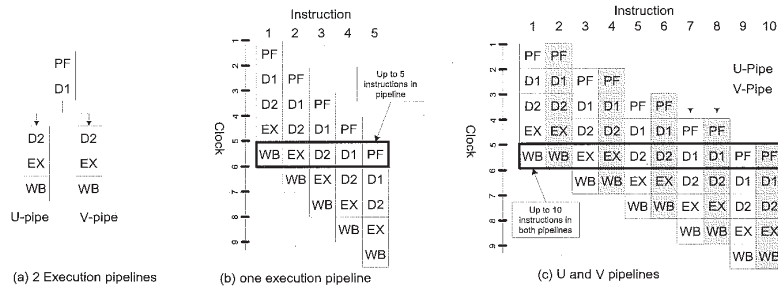


FIGURE 19-3 The Pentium processor's dual execution pipelines.

19.5.2 Performance Considerations

Typically the pipeline is not maintained at its maximum capacity because of data dependency, register contention, or other restrictions imposed by the processor. These restrictions are known as the Pentium pairing rules (we discuss these in more detail in the following section).

Figure 19-4 shows a couple of stalls caused by data dependency and instruction prefetch. In the first case (inside bold box), the processor is waiting for data or the result of an address calculation. In the second case, the fetch unit is fetching instructions from the L2 cache or main memory, which causes bubbles to propagate in the pipeline.

Similar bubbles could fill up the V pipeline if your instructions don't adhere to the Pentium pairing rules. The Pentium processor issues two consecutive instructions in both pipelines only if the first instruction is pairable in the U pipeline and the second instruction is pairable in the V pipeline. If the two instructions don't pair, both will execute in the U pipeline in 2 clock cycles, and the V pipeline will be empty—bubbly.

19.5.3 Pentium Integer Pairing Rules

The Pentium processor pairs two instructions only if they satisfy *all* the pairing rules listed in Figure 19-5. In the figure, three examples are listed for each of the rules illustrating the usage of the rule.

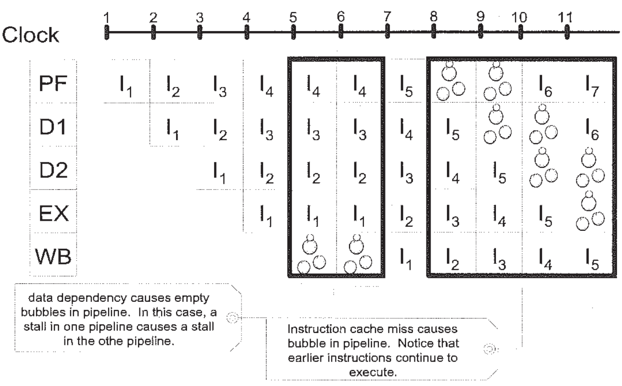


FIGURE 19-4 Data dependency and instruction fetch bubbles in the pipeline.

PU Pairable in U							
PV Pairable in V							
UV Pairable in both							
NP Not Pairable							

Pairing Rules

Pairing rule 1: Two consecutive instructions pair if the first instruction is pairable in the U pipe and the second instruction is pairable in the V pipe.

✗	*shl eax, cl* *inc esi*	*NP* *UV*	✗	*Inc esi* *Shl eax, 1*	*UV* *PU*	✓	*shl eax, 1* *inc esi*	*PU* *UV*

Pairing rule 2: The second instruction cannot *read* or *write* any subset of a register if any subset of it was *written* by the first instruction. Basically, if you write to *al, ah, ax,* or *eax* in the first instruction, you cannot read or write to any of them in the second instruction; the same applies to all other registers.

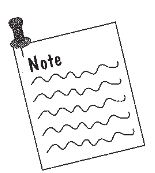
✗	*Mov **EAX**, 10* *Mov [ebp], **EAX***	✗	*Mov **AL**, 10* *Mov [ebp], **AH***	✓	*mov eax, 10* *inc ebx*
✗	*mov **EAX**, 10* *inc **EAX***	✗	*Mov **AL**, 10* *Mov **AH**, 20*	✓	*mov ebx, eax* *mov eax, 10*

Reading before writing is OK

FIGURE 19-5 Pentium integer pairing rules.

For optimal pairing, always use simple instructions such as memory moves, ALU operations, and logical operations. You can use VTune's static analyzer to easily determine the pairability of your instructions.[1]

19.5.4 Address Generation Interlock (AGI)



The Pentium Pro processor does not suffer from AGI stalls.

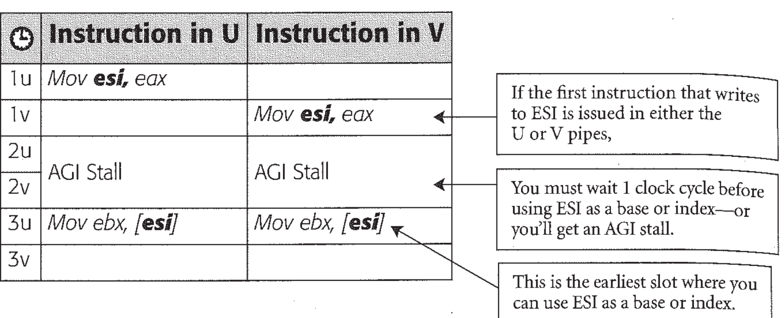
As an extension to pairing rule 2, the Pentium processor suffers a 1-clock penalty because of Address Generation Interlock (AGI). AGI stalls occur when an instruction writes to a register that is then used as a base or an index in the following clock cycle. For example, consider the following two instructions:

```
Mov esi, eax
Mov ebx, [esi]
```

The second instruction suffers from an AGI stall since it uses the *esi* register as a base register, and *esi* was just updated in the first instruction (see Figure 19-6). As a result, both processor pipelines stall for 1 clock cycle as below.

PART VI

1. You can find a list of instruction pairability in the *Intel Architecture Optimization Manual* found on the companion CD.



⏲	Instruction in U	Instruction in V
1u	*Mov* ***esi,*** *eax*	
1v		*Mov* ***esi,*** *eax*
2u	AGI Stall	AGI Stall
2v		
3u	*Mov ebx,* ***[esi]***	*Mov ebx,* ***[esi]***
3v		

FIGURE 19-6 AGI stall in the Pentium processor.

To avoid AGI stalls, you can insert other useful instructions between the two instructions—as long as the inserted instructions don't use *esi* as a base or an index.

19.6 Write Buffers

19.6.1 Operational Overview

The Pentium with MMX technology processor has four Write buffers that can be accessed by either pipe.

The Pentium processor features two 32-bit Write buffers that queue data on its way to the external bus—the L2 cache or main memory. One buffer is dedicated to the U pipe and one to the V pipe. The main purpose of the Write buffers is to enhance the performance of consecutive writes to memory. Note that the Write buffers are not used when you write to memory addresses that are already in the L1 cache; only writes to the external bus are queued in the Write buffers.

So why are the Write buffers useful? Without the Write buffers, when you write data to memory that is not part of the L1 cache—uncached memory

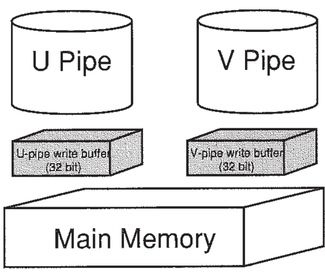


FIGURE 19-7 The Pentium processor's Write buffers.

or elsewhere—the processor has to wait until the data is completely transferred before it can move on to the next instruction. Depending on where the data has to go—main memory, the L2 cache, and so forth, the write can take a long time to complete compared to when you're writing to the L1 cache. The Write buffers can hold the data until it has a chance to write it to memory. Meanwhile the processor is allowed to continue execution at the next instruction.

Now, if the processor is asked to write another piece of data to memory—from the same pipe—while the first bit of data is still being written to memory, the processor stalls until the original data in the Write buffer is written to memory. The Write buffer is also flushed out if you read from a memory location that is not in the L1 cache or from uncached memory—and the processor stalls until the Write buffers are flushed before it executes the read.

19.6.2 Performance Considerations

You typically don't have to worry about the Write buffers unless you're writing data to video memory or some other uncached memory location. Multimedia video applications that write directly to video memory—uncached memory—could benefit greatly if developers paid special attention to the pattern in which video is written to video memory.

Since it takes time to write data to video memory, you could space out the writes and do some processing in between. The time it takes to write to video memory depends on many factors, such as the type of memory used on the graphics adapter. For the sake of simplicity, assume that on a 100-MHz processor it takes 10 CPU cycles to write a 32-bit word to video memory. If you continuously write from a register to video memory, you'll write 32 bits every 10 clock cycles—of course, you're stalling for 9 of them.[2] Now, if you have some other processing to do, you can fill the 9 cycles getting some useful work accomplished—as long as you access your data from the L1 cache. For example, in a color conversion routine, rather than converting the whole image in system memory and then writing it out to the video card, you can perform the color conversion calculations in between writes to video memory—as long as you only access registers or the L1 cache.

At this point, you might have the impression that you only need to assure that instructions pair correctly in order to gain performance. Ideally, this is

2. For a faster processor, you wait the same physical time, but you wait more processor clocks. That means you can squeeze in even more instructions between writes to memory.

true as long as your code and data are waiting in the L1 cache. Unfortunately, with multimedia applications, you cannot make such an assumption, since you typically deal with a huge amount of data, and not all of it will fit in the L1 cache at once.

Because the L2 cache and memory run much more slowly than the internal components, the processor has to wait for them to deliver code or data. Even though the L2 cache and memory can get faster, they can't have the same speed ramp as the processor. As a result, the situation gets even worse with faster processors, since they have to wait more clocks for the same response time from the L2 cache or main memory.

As a multimedia developer, you must pay special attention to how you access your data. Multimedia applications are memory intensive in nature and exert a huge demand on the memory subsystem. Depending on the nature of the data, certain access patterns are more efficient than others. For example, you can preload the data to make sure that it is in the L1 cache before you use it. You can also space out your writes to video memory and do some useful operations in between.

We could spend a whole chapter on memory optimization issues, and that is exactly what we did. We devoted the chapter at the end of this part to discussing memory optimization techniques.

19.7 Revisiting Our Sprite Sample

Great! You've made it this far. Now you can take a deep breath. But are your hands still itching to optimize something? Let's use the assembly version[3] of the sprite sample from Part II and try to figure out how we can optimize its performance on the Pentium processor.

19.7.1 Overview of the Assembly Version of CSprite

First let's see how the sprite sample works. As you know, a sprite is a regular bitmap where one of the colors is designated to be transparent. A sprite is typically overlaid on top of a background, and only the nontransparent pixels of the sprite show up against the background. Of course, there are many ways to overlay a sprite on top of a background. For example, you can read pixels from both images and merge them in memory and then write out the

3. The assembly version of the sprite was only mentioned in Part II. You can find the sources on the companion CD.

merged result to the video screen. But if the background is already in video memory, this might be an expensive solution—video memory takes a long time to write and even longer to read.

In our implementation, we assume that the background is already in video memory. We first read 4 bytes (a DWORD) from the sprite and only write out the nontransparent pixels to the screen. To do that, you could look at each pixel in the sprite to determine if it is transparent or not, and only write out the nontransparent ones. But this strategy causes a huge branch misprediction problem since you don't really know yet what's in the sprite.

Since we're dealing with a static sprite, we decided to preprocess the sprite to figure out which pixels we really need to write. The outcome of the preprocessing is a command list indicating which pixels we should care about and which we shouldn't even examine. Another advantage of the command list is that we avoid using compare instructions while we're displaying the sprite, so we are saved all the branch mispredictions that otherwise would occur.

Consider the sprite bitmap in Figure 19-8. When we preprocess the sprite, we handle one DWORD at a time and decide what we're supposed to do for that DWORD. For example, the first DWORD in line 0 says: "only draw the third pixel." The next one says, "Draw all four pixels," and so on. These are basically the commands in the command list. Table 19-2 shows us the command list we would generate for this sprite.

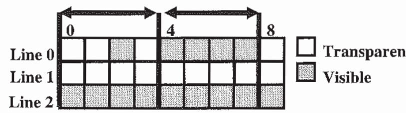


FIGURE 19-8 Simple sprite.

TABLE 19-2 Command List for a Sprite

Line	Command
0	*WriteByte3, WriteDWord, EndOfLine*
1	*SkipLine*
2	*SolidLine, EndOfSprite*

When it's time to display the sprite, we first read the command list and then do whatever the command says. Notice that we never have to deal with transparent pixels at all; we know exactly which pixels we need to write. This allows us to process the sprite in less time and reduces the bandwidth on the system bus.

To avoid branch mispredictions, we designed the commands in such a way that they could be used as an index to a jump table (Figure 19-9). The *JumpTable[]* array holds the address of the label that handles that task. You'll see what we're getting at soon.

Table Index	Pixels 0	1	2	3	Command JumpTable[]
0 (0000)					SKIPDWORD,
1 (0001)	■				WRITEBYTE0001,
2 (0010)		■			WRITEBYTE0010,
3 (0011)	■	■			WRITEBYTE0011,
4 (0100)			■		WRITEBYTE0100,
5 (0101)	■		■		WRITEBYTE0101,
6 (0110)		■	■		WRITEBYTE0110,
7 (0111)	■	■	■		WRITEBYTE0111,
8 (1000)				■	WRITEBYTE1000,
9 (1001)	■			■	WRITEBYTE1001,
10 (1010)		■		■	WRITEBYTE1010,
11 (1011)	■	■		■	WRITEBYTE1011,
12 (1100)			■	■	WRITEBYTE1100,
13 (1101)	■		■	■	WRITEBYTE1101,
14 (1110)		■	■	■	WRITEBYTE1110,
15 (1111)	■	■	■	■	WRITEDWORD,
16					SOLIDLINE,
17					SKIPLINE,
18					ENDOFLINE,
19					ENDOFSPRITE

Notice that the pixel arrangement corresponds to the lower 4 bits of the index.

FIGURE 19-9 Sprite command jump table.

Now, if you apply this jump table to the sprite in Figure 19-8, you'll end up with the command list shown in Table 19-3.

nd then
vith
rite.
dwidth

h a way
he
isk.

end up

TABLE 19-3 Command List for the Simple Sprite

Line	Command	CommandList[]
0	*WriteByte0100,*	4
	WriteDWord,	15
	EndOfLine	18
1	*SkipLine*	17
2	*SolidLine,*	16
	EndOfSprite	19

When the *BltSprite()* function is first called, it performs an unconditional jump to the first command in the command list. After a command is executed, a similar jump transfers control to the next command in the list. This process is repeated until the *EndOfSprite* is reached, which returns control to the caller routine. Notice in Figure 19-9 that each command processes at least one DWORD of the sprite.

```
// This is a pseudo code that demonstrates how the jump table works.
// The pCommandList points to the first command into the JumpTable.
BltSprite(PBYTE *pSrc, PBYTE *pDst, PBYTE *pCommandList)
{
    // Execute the first command in the list.
    goto JumpTable[*pCommandList++];

WriteByte0001:                          // Only write first byte of DWORD
    pDst[0] = pSrc[0];
    pSrc+=4, pDst+=4;                   // go to next DWORD
    goto JumpTable[*pCommandList++];

WriteByte0010:                          // Only write second byte of DWORD
    pDst[1] = pSrc[1];
    pSrc+=4, pDst+=4;
    goto JumpTable[*pCommandList ++];

WriteDWord:                             // Write entire DWORD
    ((DWORD*)pDst)[0] = ((DWORD*)pSrc)[0];
    pSrc+=4, pDst+=4;
    goto JumpTable[*pCommandList ++];

    // The rest of the operations are similar…

EndOfSprite:                            // Done…
    return;
}
```

PART VI

19.7.2 Analyzing the Performance of Our Sprite Sample

Before we go into some analysis, you should be aware that you can easily use VTune to figure out if an instruction sequence pair will work. But in order to figure out how to optimize your code, you'll still need to have knowledge of the Pentium pairing rules.

In the following illustration, we'll show you our thinking process when we hand-optimize our code. For the purpose of our analysis, let's try to optimize the assembly version of the WriteDWord command shown below.

```
WriteDWord:
    mov ecx, [esi]          ; Read DWORD from sprite
    mov [edi], ecx          ; Write DWORD to background
    inc ebx                 ; increment index pointer
    mov dl, [ebx]           ; read jump table index
    add esi, 4              ; next DWORD to sprite
    add edi, 4              ; next DWORD in background
    jmp JumpTable[edx*4]    ; Jump to next macro based on index.
```

Typically, when you schedule instructions, you start the analysis from the first instruction in a block and try to pair it with the next sequential instruction according to the Pentium pairing rules. If the first two instructions do not pair, you would skip the first instruction and try to pair the second one with the third, and so on.

In Table 19-4, you can find the nonoptimized sequence of the WriteDWord command. In the first column we see the instruction sequence where paired instructions are separated by a blank line. The second column has the num-

TABLE 19-4 The Nonoptimized Version of the *WriteDWord* Command

Before (7 clocks)	◷	Analysis Steps
1. mov ecx, [esi+1]	1	• *(1,2) do not pair because (2) uses a register written by 1(**Pairing Rule 2**)*
2. mov [edi+1], ecx 3. inc ebx	1	• *(2,3) pair; both instructions are UV pairable, and there is no dependency*
4. mov dl, [ebx] 5. add esi, 4	2	• *(4,5) pair; however, there is an additional clock because of an AGI stall—EBX was just incremented.*
6. add edi, 4	1	• *(6,7) do not pair because (7) is not pairable (**Pairing Rule 1**).*
7. jmp jumptable[edx*4]	2	• *Indirect register jumps are not pairable; they also take 2 clocks to execute when the jump address is in BTB (**PR1 & Branch Timing**).*

ber of clocks it takes to execute each pair of instructions, and the last column illustrates the step-by-step thinking process that we used to figure out if two instructions could be paired.

Based on the analysis, you can see that there are a few pairing and scheduling problems in the code. For example, instruction 4 has an AGI because the *ebx* register was just incremented in instruction 3. To avoid an AGI stall, you could switch the two instructions and use `mov dl, [ebx+1]` to reference the correct byte. See Table 19-5 for an optimized version of the WriteDWord command.

TABLE 19-5 Optimized Version of the *WriteDWord* Command from the Sprite Sample

Before (5 clocks)		Analysis Steps
1. mov ecx, [esi+1] 2. mov dl, [ebx+1]	1	• *(1,2) pair*
3. inc ebx 4. add esi, 4	1	• *(3,4) pair*
5. mov [edi+1], ecx 6. add edi, 4	1	• *(5,6) pair*
7. jmp JumpTable[edx*4	2	• *Indirect register jumps are not pairable. Take 2 clocks when jump address is in BTB.*

This simple optimization resulted in a gain of 2 clocks. In reality, it would be great if either of the above samples executed in 5 or 7 cycles. Unfortunately, both sequences take much longer to execute than indicated because we are writing the results directly to video memory, and this process, as we mentioned earlier, is very slow compared to how fast the processor can run.

Also note that the same instruction performs unaligned memory writes depending on the position of the sprite on the screen. Misaligned memory writes take more cycles to execute because the processor splits the write into smaller writes.

In Table 19-6, you can see the actual measurements for both the nonoptimized and the optimized versions of the WriteDWord command. In our measurement, we used the worst-case sprite for the WriteDWord command, where all pixels are visible and the sprite width is a multiple of 4.

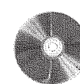
TABLE 19-6 Measured Cycle Timing for Nonoptimized and Optimized Versions of the *WriteDWord* Command

	Nonoptimized		Optimized	
Output Buffer Alignment	**Clocks/ Sprite**	**Clocks/ 4 Pixels**	**Clocks/ Sprite**	**Clocks/ 4 Pixels**
0	14550	11	14480	11
1	53600	40.5	53300	40
2	53700	40.5	53400	40
3	53600	40.5	53400	40

Notice that the 2-clock gain in performance is not even noticeable. As we mentioned before, this is because the slow video memory access chews up the 2-clock gain we saw in the optimized version.

You can use VTune to detect misaligned accesses.

Take a closer look at the measurements again. Notice that all misaligned writes to video memory result in a huge penalty compared to the aligned writes (~30 clocks/DWORD). As a result, we rewrote the WriteDWord command to perform only aligned memory writes with some shifting and masking. And, as we expected, we received a huge performance boost (~13 clocks/DWORD). You can find a copy of the aligned write implementation of the sprite on the companion CD.



19.7.3 Do I Really Need to Schedule My Code?

Absolutely! We deliberately selected this example for two reasons. First, to show you how to optimize your code from the processor's point of view. Second, to point out that other components in the system, such as video memory, can adversely affect your application performance. You will definitely benefit from scheduling instructions, especially if the data is close to the processor (basically, in registers or L1 or L2 cache). For example, sprites written into system memory execute at 8 clocks/DWORD, and sprites written into video memory execute at 38 clocks/DWORD. We have found that most multimedia algorithms, such as those for compression, decompression, image filtration, and 3D benefit from instruction scheduling.

WHAT HAVE YOU LEARNED?

At this stage, you should be familiar with the internal components of the Pentium processor, and you should have an idea of what you can do to achieve optimal performance on this processor. Here is a recap of the tips you should have picked up by reading this chapter:

- Know your data: what does it look like, where does it come from, and where is it going to? (See Chapter 23 for more.)
- Align loops, unconditional branches, and function labels to 8-byte cache boundary.
- Keep infrequently executed code and data separate from the inner loops.
- Use simple instructions for optimal pairing.
- Avoid branch mispredictions and AGI stalls.
- Measure the performance of your code, because this is the best way to get a sense of how well it is executing.

CHAPTER 20

The Pentium with MMX Technology Processor

WHY READ THIS CHAPTER?

In this chapter you'll learn about the Pentium with MMX technology processor and its own pairing and scheduling rules (the Pentium II processor is discussed in a later chapter).

In this chapter, you will

- get an architectural overview of MMX technology,
- learn about the MMX data types, instructions, and register set,
- learn the MMX pairing and scheduling rules,
- see how to mix floating-point and MMX instructions using the EMMS guideline,
- rewrite the sprite sample using MMX instructions, and
- optimize the sprite for MMX technology using the scheduling rules.

20.1 A Look at MMX Technology

With the Pentium processor, Intel implemented parallel processing with dual execution pipelines. MMX technology is the latest major addition to the Intel Architecture, including fifty-seven new instructions, and eight new 64-bit registers. With MMX technology, Intel took parallel processing to the level where a single instruction operates on multiple elements of data—this is known as Single Instruction Multiple Data (SIMD).

Although the name *MMX* might imply a specific set of applications, *multimedia,* the new instruction set is a general-purpose implementation of the

SIMD concept. It benefits all applications that perform the same operation repetitively on contiguous blocks of data.

MMX technology introduces a new set of instructions and registers. The instructions operate in parallel on BYTE, WORD, DWORD, and QWORD data types packed into 64-bit registers. They perform signed and unsigned arithmetic, logical, packing, and unpacking operations on the previously mentioned data types' boundaries. They allow for saturation or wrap-around to handle overflow and under-flow conditions.

In this chapter, you will first get an overview of MMX technology and a brief description of the instruction and register sets. You will then learn the MMX scheduling rules and how to apply them to the sprite sample.

20.2 SIMD

Typically, integer instructions operate on individual integer data elements ($A + B$) (see Figure 20-1a). SIMD instructions, on the other hand, operate on integer data arrays (A[1..n]+B[1..n]), where n is the number of elements in the array, for example, $n = 4$ in (see Figure 20-1b).

In Figure 20-1, note that the SIMD processor duplicates the same execution unit four times. Consequently, the SIMD processor can process four data elements in the same clock cycle (Figure 20-1b) while the scalar single instruction, single data (SISD) processor takes four clock cycles to process the same data (Figure 20-1a).

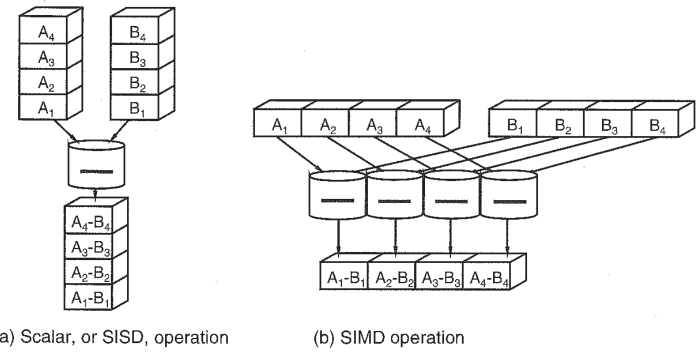


FIGURE 20-1 Scalar versus SIMD operations.

20.3 Architectural Overview

The Pentium with MMX technology is the first implementation of the MMX technology, based on the Pentium processor. Recently Intel added MMX technology to the Pentium Pro to create the Pentium II processor. In this chapter we'll discuss only the extension of MMX technology to the Pentium processor. The Pentium II processor is discussed in Chapter 22.

Figure 20-2 shows an architectural overview of the Pentium with MMX technology processor. The processor includes eight new MMX registers and fifty-seven new MMX instructions. In addition, the processor doubles the size of the L1 code and data caches to 16K each and adds a Return Stack Buffer, which reduces the overhead of function returns. Finally, the two dedicated Pentium Write buffers are replaced with four shared Write buffers—32 bits each.

20.3.1 The Pool of Four Write Buffers

In the previous chapter, we mentioned that the Pentium processor has two dedicated Write buffers, which are used to queue data writes that do not hit the L1 cache, write through cache, or write to uncached memory. In the Pentium processor without MMX technology, one buffer is dedicated to the U pipe and one is dedicated to the V pipe. As a result of this constraint, each pipeline is allowed to queue only one memory write before the pipeline gets stalled.

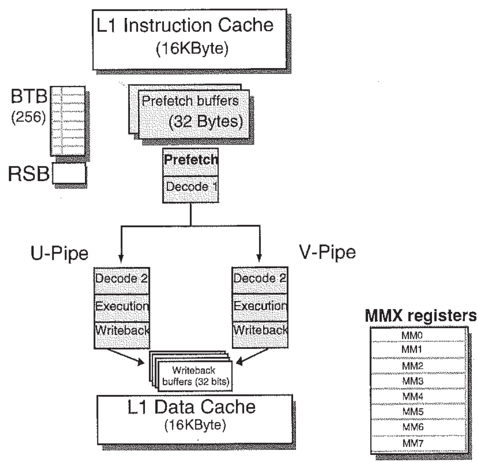
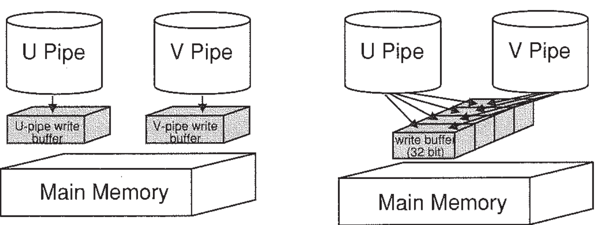


FIGURE 20-2 Architectural view of the Pentium with MMX technology processor.



a. Pentium processor b. Pentium processor with MMX technology

FIGURE 20-3 Write buffers for Pentium and Pentium with MMX technology processors.

To enhance write performance, the Pentium with MMX technology processor doubled the number of Write buffers—to four 32-bit Write buffers. It also removed the constraint that buffers are dedicated to a specific pipeline. Any of the four buffers can be accessed from either pipeline allowing up to four back-to-back 32-bit memory writes or two 64-bit writes regardless of which pipeline the writes came from. As a result, each pipeline can write up to four 32-bit writes before stalling the pipeline (see Figure 20-3).

20.3.2 MMX Uses Floating-Point Registers

To maintain operating system compatibility, MMX technology maps the MMX registers on top of the IA floating-point (FP) registers. Figure 20-4 shows a diagram of the MMX registers mapped one-to-one to the mantissa part of the floating-point registers. As a result, when you read or write to an MMX register, you read and write to one of the floating-point registers and vice versa. The only difference is how the data is interpreted in the register—after all, it's only bits. MMX instructions interpret the data as packed bytes, words, or double words; floating-point instructions interpret the same data as the mantissa part of a floating-point number.

REASONING FOR ALIASING THE MMX REGISTER

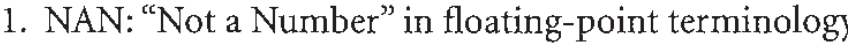
Why are the MMX registers aliased? Current operating systems save and restore the contents of the integer and floating-point registers between task switches. If a new set of registers is added to the processor, the operating system would have to be modified in order to save and restore the new MMX registers between task switches. By mapping the MMX registers onto the floating-point registers, the contents of the MMX registers are saved and restored automatically between task switches. After all, they are the same bits. As a result, you can run MMX instructions under the current operating system without encountering any problems between task switches.

FIGURE 20-4 Aliasing of MMX registers on top of the floating-point registers.

So what's the catch? First, it's obvious that you cannot rely on the contents of the floating-point registers after you execute an MMX piece of code, or vice versa. What's not readily obvious is that the processor could generate floating-point errors when you execute a floating-point instruction after an MMX instruction. Why? Since an MMX instruction treats the entire 64 register bits as packed integers, it can write any sequence of bits in the MMX register. But from the floating-point of view, certain bit combinations in the mantissa combined with certain bits in the exponent generate floating-point errors such as NAN,[1] stack overflow or under-flow. Refer to the Pentium processor programmer's manual on the CD for more details.

20.3.3 EMMS to the Rescue: How to Mix MMX and FP Instructions

That is not to say that you can never mix MMX and floating-point code fragments in the same application. Rather, you can mix the two types of instructions if and only if you can guarantee that no floating-point errors will occur when you switch from MMX to floating-point. To do so, you must use the new MMX instruction *EMMS* (Empty MMX Technology State), which marks all the floating-point registers as *Empty*. To the floating-point unit, an empty register indicates that it does not have any data in the register and, therefore, does not generate stack overflow errors.



1. NAN: "Not a Number" in floating-point terminology.

EMMS takes 0–11 cycles on the Pentium II processor.

Notice that you should use the EMMS instructions wisely since the EMMS instruction can take up to 53 clock cycles to execute on the Pentium with MMX technology.[2] Ouch! Keeping that in mind, you should use the EMMS instructions only in the following situations:

- If you plan on mixing MMX and floating-point code in the same application, insert the EMMS instruction at the end of each MMX block.
- If your DLL exports an MMX function that could be called by an application that uses floating-point operations, insert the EMMS instruction before you return from the routine.

To use the EMMS instruction properly, just remember these simple rules:

- Minimize switching between MMX and floating-point instructions because the switch can be expensive (costing up to 53 cycles).
- Never mix MMX and floating-point instructions at the instruction level—separate the MMX and floating-point calculations into separate routines and use EMMS at the end of MMX routines.
- Never assume that the state of the registers is valid across transitions because both MMX and floating-point instructions write and read from the same physical register file.
- Always insert an EMMS instruction at the end of an MMX block unless you are absolutely sure that no floating-point instruction will be used.

20.4 MMX Technology Data Types

You can interpret the 64-bit data format in an MMX register according to the instruction that you use. Notice that with the exception of EMMS and the 32-bit memory transfer instruction (`MOVD`), all MMX technology instructions operate on one of the data formats shown in Figure 20-5.

The `MOVD` instruction operates on the lower 32 bits of an MMX register, where it transfers the register's contents to memory or to an integer register (*eax, ebx,* and so forth). The `MOVD` instruction also transfers 32 bits of data from memory or an integer register to the lower 32 bits of the MMX register; in this case, the high 32 bits are set to zero.

2. The actual EMMS instruction takes only 1 clock cycle to execute, but when the first floating-point instruction executes, it takes up to 53 cycles to completely switch to the floating-point mode.

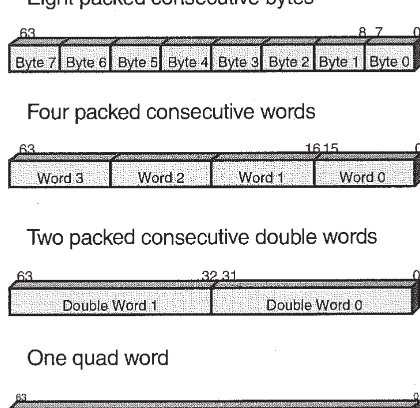


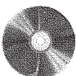
FIGURE 20-5 Data formats for MMX technology instructions.

The EMMS instruction only affects the tag bits of the floating-point registers. It sets all the tag bits to 1, indicating that the floating-point registers are empty.

20.5 The MMX Instruction Set

With the introduction of MMX technology, Intel added fifty-seven new instructions to the IA architecture. These instructions consist of arithmetic, comparison, conversion, logical, shift, and data transfer instructions.

With the exception of EMMS and data transfer instructions (`MOVQ` and `MOVD`), all MMX instructions follow the format shown in Figure 20-6.



In Table 20-1, you can find a list of the MMX instructions with a brief description of each. For a detailed description, please refer to the *Intel Architecture MMX Technology: Programmer's Reference Manual* found on the companion CD.

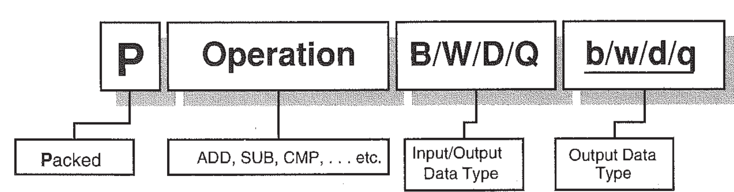


FIGURE 20-6 The MMX instruction format.

TABLE 20-1 Summary of MMX Instruction Set

Instruction Type	Instruction	Description
Arithmetic	*PADD*[**B|W|D**]	Add with Wraparound
	PADDS[**B|W**]	Add signed with saturation
	PADDUS[**B|W**]	Add unsigned with saturation
	PSUB [**B|W|D**]	Subtract with Wraparound
	PSUBS[**B|W**]	Subtract signed with saturation
	PSUBUS[**B|W**]	Subtract unsigned with saturation
	*PMULH***W**/ *PMULL***W**	Multiply four words and store high/low 32-bit result in register
	*PMADD***Wd**	Packed multiply and add
Comparison	*PCMPEQ*[**B|W|D**]	Compare if equal
	PCMPGT[**B|W|D**]	Compare if greater than
Conversion	*PACKSS***Wb** *PACKSS***Dw**	Convert signed WORD/DWORD to signed byte/word using signed saturation
	*PACKUS***Wb**	Convert signed word to signed byte using signed saturation.
	*PUNPCKH***Bw** *PUNPCKH***Wd** *PUNPCKH***Dq**	Interleave the high order 32-bit data elements of the source and destination operands across data type boundary.
	*PUNPCKL***Bw** *PUNPCKL***Wd** *PUNPCKL***Dq**	Interleave the low order 32-bit data elements of the source and destination operands across data type boundary.
Logical	PAND	Bitwise logical AND
	PANDN	Bitwise logical AND NOT
	POR	Bitwise logical OR
	PXOR	Bitwise logical XOR
Shift	*PSLL*[**W|D|Q**] *PSRL*[**W|D|Q**]	Shift left/right logical without carry across data type boundary
	PSRA[**W|D**]	Shift right arithmetic where the sign is the most significant bit (MSB) for the specific data-type.
Data Transfer	MOVD	Transfers 32 bits between MMX register and integer register or memory
	MOVQ	Transfers 64 bits between MMX register and MMX register or memory
EMMS	EMMS	Empty MMX technology state. Clears FP tag word.

paddus **B:** Add 8 bytes using unsigned arithmetic with saturation.

	FF	00	01	FF	FE	80	FF	FF
+	04	03	02	01	00	99	99	01
	FF	03	03	FF	FE	FF	FF	FF

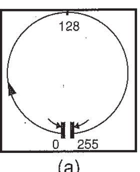
paddus **W:** Add 4 words using unsigned arithmetic with saturation.

	FF 00	01 FF	FE 80	FF FF
+	04 03	02 01	00 99	99 01
	FF FF	04 00	FF 19	FF FF

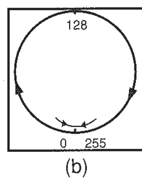
20.6

Notice that some of the instructions have few formats for signed versus unsigned and wraparound versus saturation calculations. You already know about signed versus unsigned calculations, so let's make sure you understand the wraparound versus saturation modes.

Assume that you have 2 bytes, and you want to add them together using unsigned arithmetic. Since both are unsigned, their values can only range from 0 to 255. But when you add them together, the results could range from 0 to 510, which does not fit in 1 byte. So what do you do? Well, one option is to saturate the result to 255, and the other option is to keep only the lowest 8 bits of the result (the wraparound). Let's see how that works.



(a)



(b)

FIGURE 20-7 Unsigned case (a) and wraparound mode (b).

When you use *saturate* instructions, results greater than the maximum possible value are clamped to the maximum value. Results less than the minimum value are clamped to the minimum value. In the unsigned case, the final result would be clamped to 0 and 255, and in the signed case, they would be clamped to –127 and 127. It is just like trying to go around in a broken circle.

On the other hand, when you use *wraparound* mode, you would calculate the result to whatever precision possible and only keep the lowest significant 8 bits. It is just like going around in a circle.

20.6 Using MMX Technology to Render Our Sprite Sample

It's time to revisit our spite again. This time, let's use the new MMX instructions to implement our favorite sprite. When we worked with the sprite sample in the previous chapter, we used integer instructions to selectively write only the visible pixels of the sprite using byte writes (for example,

`mov [mem], al`). Since MMX technology does not offer byte write operations, we cannot selectively write the visible pixels (1 byte at a time); rather we have to merge the sprite and the background in an MMX register and write the merged bytes to video memory. To do that, we use a Read/Modify/Write algorithm.

DEBUGGING MMX TECHNOLOGY INSTRUCTIONS

MSVC 4.1+ Support

Microsoft added support to the MMX technology instruction set starting from MSVC 4.1 in the form of in-line assembly using the *_asm* directive. Unfortunately, this support is not enabled by default, and there is no selection in the compiler settings to enable it. Microsoft added the compiler switch `/G5M` to allow the compiler to recognize and process MMX instructions.

The sprite sample makefile (project file) has the `/G5M` switch set. This is done in the compiler setting dialog box.

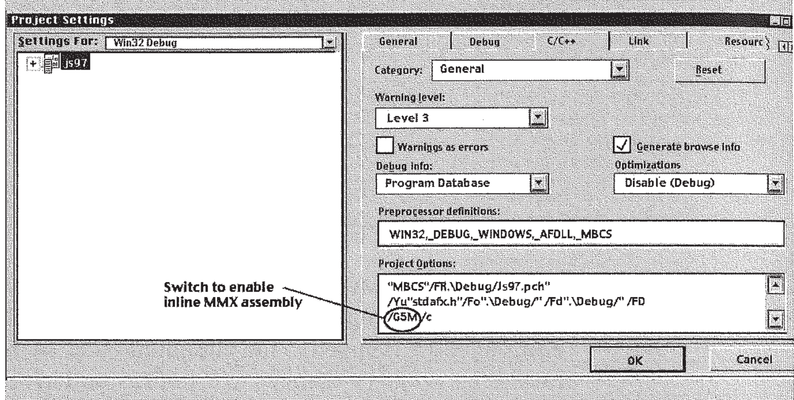


FIGURE 20-8 MSVC in-line MMX assembly support.

NuMega WinIce

NuMega WinIce also supports the new MMX instructions and registers. To view the MMX registers, use the command

WF | B, W, D where (assume an MMX register has the value 0x8877665544332211)

WFB displays MMX registers in BYTE format:

88	77	66	55	44	33	22	11

WFD displays MMX registers in DWORD format:

8877	6655	4433	2211

WFW displays MMX registers in WORD format:

88776655	4432211

With this method, first we read 8 bytes from the background image into one of the MMX registers, and then we apply the *visible* pixels to the background using some logical masking techniques. Finally, we write out the modified background to its original location.

In this sample, we'll simply re-implement the sprite sample we worked with in the previous chapter, but this time we will use MMX instructions. We start out by redefining the transparency color member variable to a 64-bit entity and duplicating the 8-bit transparency color byte to all 8 bytes.

```
class CSprite {
public:
    __int64 m_qwTransp;  // Allow space for 64 bit for MMX transparency
};

CSprite::CSprite(CBitmap &bitmap, BYTE byKeyColor)
{
    // Duplicate the transparency color accross the 8 bytes.
    memset (&m_qwTransp, byKeyColor, 8);
}
```

Next we replace the contents of the *Blt()* routine with our MMX technology sprite *Blt()* routine. This implementation of the sprite using MMX instructions is not yet optimized.

Before we start, notice that we're assuming an 8 bpp RGB color format and that the sprite is at least 8 pixels wide. The *Blt()* routine processes each scan line in two stages: The first stage handles the left side of the scan line 8 pixels at a time; the second stage handles the situations when the sprite width is not a multiple of 8 and handles the remaining pixels at the end of the scan line. Since both stages use the same technique for overlaying the sprite on top of a background, we'll discuss only the first stage, shown in the code that follows.

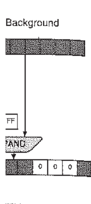
```
void CSprite::Blt(LPBYTE lpSurface, long lPitch, CPoint &point)
{
    PBYTE       pDst;
    DWORD       row, col;
    DWORD       dwHeight = m_dwHeight;
    __int64     qwTransp = m_qwTrasnp;
    PBYTE       pSprite = m_pData;

    // compute address dst and src pixels.   note pitch can be negative
    pDst = (PBYTE)((long)lpSurface + point.x + point.y * lPitch);

    int n8ByteBlocks = m_dwWidth >> 3;  // number of 8-byte blocks
```

Since the inline **_asm** cannot access class members, we have to copy it to a local variable first and then use the local within the **_asm** block.

```
_asm {
    mov     edi, pDst
    mov     esi, pSprite
    movq    mm3, qwTransp

DoOneLine:
    // Check if sprite has more than 8 bytes in width.
    mov     ecx, n8ByteBlocks
    cmp     ecx, 0
    je      LessThan8BytesLeft

DoQWord:
    // This loop processes 8 pixels at a time
    movq    mm0, [esi]   // Sprite
    movq    mm1, [edi]   // BkGnd

    movq    mm2, mm3     // Transparency Color
    pcmpeqb mm2, mm0     // Transparency Mask
    pand    mm1, mm2     // keep bkgnd pixels
    pandn   mm2, mm0     // keep sprite pixels
    por     mm1, mm2     // merge them
    movq    [edi], mm1   // write out

    add     edi, 8        // advance pointers
    add     esi, 8
    dec     ecx
    jnz     DoQWord
    }
}
```

The routine processes 8 contiguous pixels at a time starting from the left-most pixel of a sprite scan line. For each quad word (8 pixels), the routine uses the PCMPEQB instruction to create a transparency mask from the sprite pixels and the transparency color. The PCMPEQB instruction compares the 8 bytes of the sprite with the 8 bytes of the transparency color. For each byte, the result of the comparison is "FF" if the bytes match (these are the transparent pixels) and "0" if they don't match (these are the opaque pixels).

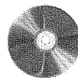
Next the PAND instruction is applied to the newly created mask and the background allowing the background pixels to occupy the space of the transparent pixels in the sprite (the ones that resulted in FF); the other pixels are zero. The PANDN instruction is then used to create a similar pattern for the opaque pixels in the sprite. In this step, the mask is first inverted and then ANDed with the original sprite pixels—this basically clears out the bytes corresponding to the transparent pixels in the sprite. Finally, the last two results are combined together with the POR instruction in order to form the image—the sprite on top of the background.



20.7 MMX Technology Optimization Rules and Penalties

Before we start the analysis of the sample, it would be helpful to go through some of the essential optimization rules and penalties for Pentium processors with MMX technology.

All the general rules that apply to the Pentium and the Pentium Pro processors apply to their counterparts with MMX technology. There are also new rules that only apply to MMX instruction scheduling, as well as associated penalties that go with them. In the following paragraphs, we will discuss MMX instruction pairing and scheduling rules as well as variations from the general Pentium scheduling rules.



Note that although the rules are listed here with minimal explanations of how to apply them, most of the rules will be demonstrated in the section where we apply MMX technology to optimizing the sprite sample. For a complete explanation, refer to the *Intel Architecture Optimization Manual* found on the companion CD.

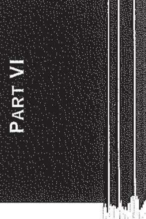
20.7.1 MMX Exceptions to General Pentium Rules

The Pentium processor with MMX technology relaxed some of the penalties we had to endure with the Pentium processor. The MMX-related rules allow for better performance on both MMX technology and integer applications. See Table 20-2 for a summary of the new rules.

TABLE 20-2 Comparison of Pentium Processor versus Pentium Processor with MMX Technology

On the Pentium Processor	On Pentium with MMX Technology
Two instructions do not pair if either of them is longer than 7 bytes	MMX instructions do not pair if the U pipe instruction is longer than 11 bytes or the V pipe instruction is longer than 7 bytes; note that prefixes are not counted here.
Prefixed instructions are only pairable in the U pipe	Instructions with 0Fh, 66H, or 67H* prefixes are pairable in either pipe. The relaxation of this restriction helps integer, floating-point, and MMX instructions. *All MMX instructions are prefixed with 0Fh.*

* 0Fh: first byte of a 2-byte opcode; 66H: operand size prefix; 67H: address size prefix.



20.7.2 MMX Instruction Pairing Rules

The pairing rules are the internal processor guidelines that must be followed in order to execute two instructions in the same clock: one instruction executes in the U pipe, and the next one executes in the V pipe.

In the previous chapter we examined the general Pentium pairing rules that are used with integer and memory operations. In this chapter, we'll examine the MMX specific pairing rules. You can find a list of the MMX instruction pairing rules in Table 20-3. Each rule is followed by three samples illustrating the application of that rule. Notice that when the pairing rules are violated, the reason for the violation is highlighted in bold for better readability.

TABLE 20-3 MMX Pairing Rules for Pentium with MMX Technology

Pair Rule 1: Two MMX instructions do not pair if they both use the MMX shifter, *(Pack Unpack, Shift instructions).*					
✗	***Pshlld*** *mm0, 2* ***Packuswb*** *mm1, mm2*	✓	*pshlld mm0, 2* *pmulhw, mm0, mm2*	✓	*Pshlld mm0, 2* *Movq mm1, mm2*
Pair Rule 2: Two MMX instructions do not pair if they both use the multiplier unit *(pmull, pmulh, pmadd).*					
✗	***Pmulhw*** *mm0, mm1* ***Pmaddwd*** *mm2, mm3*	✓	*pmulhw mm0, mm1* *pand mm1, mm2*	✓	*Pmulhw mm0, mm1* *movq mm1, mm2*
Pair Rule 3: An MMX instruction accessing memory or an integer register can only be issued in the U pipe.					
✗	*Movq mm0, Mem1* *Paddsb mm1,* ***Mem2***	✗	*movq mm1, Mem1* *movd mm2,* ***eax***	✓	*movq Mem1, mm0* *movq mm2, mm0*
Pair Rule 4: If the U pipe MMX instruction is accessing memory or an integer register, the V pipe instruction must be an MMX instruction in order for the two to pair.					
✗	*Movq mm0, Mem1* ***inc esi***	✗	*movd mm0, Mem1* ***add eax, 1***	✓	*Movq Mem1, mm0* *Pand mm2, mm0*
Pair Rule 5: The MMX destination register of the U pipe should not match the source or destination register of the V pipe *(dependency check).*					
✗	*movq* ***mm0,*** *Mem1* *movq mm1,* ***mm0***	✗	*pand* ***mm0,*** *mm1* *movq* ***mm0,*** *mm2*	✓	*Movq mm1, mm2* *Pand mm2, mm0*
Pair Rule 6: EMMS is not pairable.					

✓ Pairable
✗ Not Pairable

20.7.3 MMX Instruction Scheduling Rules

The scheduling rules are the internal processor guidelines that indicate the number of clocks it takes to execute certain instructions or when you can perform certain operations. Basically, these rules indicate when the data is ready after certain operations.

You can find a list of MMX instruction scheduling rules in Table 20-4. Study the example at the bottom of each rule to understand the restrictions imposed by the rule. We highlighted the two instructions affected by the

TABLE 20-4 MMX Scheduling Rules

Scheduling Rule 1: MMX instructions take a single clock to execute except for MMX multiply instructions, which take 3 clocks to execute. In other words, multiply instructions require 3 clocks before their data is ready for use. (See the Note on "One Clock MMX Multiply.")

Original Sequence		Actual Behavior		Optimized Sequence
Pmulhw mm0, mm1	***1u***	***Pmulhw mm0, mm1***	***1u***	***Pmulhw mm0, mm1***
Movq mm2, ***mm0***	*1v*	*Stall*	*1v*	*Inc esi*
Inc esi	*2u*	*Stall*	*2u*	*Movq mm4, mm1*
Inc edi	*2v*	*Stall*	*2v*	*Inc edi*
Pand mm3, mm1	*3u*	*Stall*	*3u*	*Pand mm3, mm1*
Movq mm4, mm1	*3v*	*Stall*	*3v*	*Dec ecx*
Dec ecx	*4u*	*Movq* ***mm2, mm0***	*4u*	***Pmulhw mm2,*** *mm3*
Jnz again	*4v*	*Inc esi, and so forth*	*4v*	*Jnz again*

The second instruction movq mm2, mm0 stalls for 3 clocks before it enters the execution stage because it needs the result of the multiply "mm0". With proper instruction scheduling, you could fill the empty slots with useful instructions.

Scheduling Rule 2: When an MMX register is updated, 1 extra cycle is needed before you can store it to memory or to an integer register; no extra clock is needed if data is moved to an MMX register.

Original Sequence		Actual Behavior		Optimized Sequence
Pand mm0, mm1	***1u***	***Pand mm0, mm1***	***1u***	***Pand mm0, mm1***
movq [esi], ***mm0***	*1v*	*Stall*	*1v*	*Add esi, 8*
Movq mm2, mm0	*2u*	*Stall*	*2u*	*Movq mm2, mm0*
Add esi, 8	*2v*	*Stall*	*2v*	*Dec ecx*
Dec ecx	*3u*	*Movq [esi],* ***mm0***	*3u*	*Movq [esi-8],* ***mm0***
Pxor mm2, mm1	*3v*	*Movq mm2, mm0*	*3v*	*Pxor mm2, mm1*

Since mm0 was just updated, the memory write instruction `MOVQ [ESI], MM0` *stalls for 1 extra clock. Notice that, In the optimized sequence, it takes only 1 clock to copy mm0 to another MMX register after the update. This rule applies only to memory or integer register writes.*

(Continued)

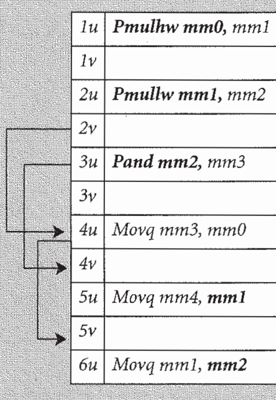
TABLE 20-4 MMX Scheduling Rules (Continued)

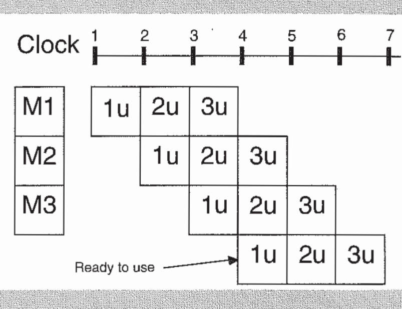
Scheduling Rule 3: No penalty for 0Fh prefix. There's a 2-clock penalty for 66h and *67h* prefixes.
Scheduling Rule 4: The Pentium processor suffers a 1-clock penalty for address resolution (AGI). (Refer to Chapter 19 for a detailed discussion of AGI stalls.)
Scheduling Rule 5: Switching between MMX technology and FP is an expensive task. (Refer to the beginning of this chapter for more information about register aliasing and EMMS.)

rule. In the first column of the table, we show the original sequence of the two instructions (back to back). In the second column we show the actual behavior of the processor when it encounters these two instructions, followed by the number of wasted clocks that result from that arrangement. In the last column, we show an optimized version of the code sequence, in which we rearranged the instructions to fill up the wasted slots.

ONE CLOCK MMX MULTIPLY

Even though the MMX multiplier takes 3 clocks to generate the results of a multiply instruction, internally the MMX multiplier consists of a three-stage pipeline. The pipelined architecture allows you to issue one multiply instruction each clock cycle—as long as you don't violate other pairing or scheduling rules. For example:





20.8 Performance Analysis of Our Sprite

Now that you know the essential optimization rules for MMX technology coding, let's have a look at the inner loop of our sprite sample (Table 20-5). As you can see, there are lots of problems here; let's see how to fix them.

Typically, we start the analysis from the first instruction in the loop and try to pair it with another instruction in the sequence according to the pairing and scheduling rules above. If we cannot find an instruction that pairs with the first instruction, we skip it and try to pair the second instruction with the third, and so on, as we saw in the previous chapter.

The original instruction sequence is listed in the first column. The second column shows the number of clock cycles it takes to execute an instruction or an instruction pair. The last column shows our step-by-step analysis of this code sequence.

TABLE 20-5 Nonoptimized MMX Technology Sprite Loop Analysis

Before (9 clocks)	⏲	Analysis Steps
DoQWord: 1. MOVQ MM0, [ESI]	1	• *(1,2) do not pair because they both access memory* (PR 3)*
2. MOVQ MM1, [EDI] 3. MOVQ MM2, MM3	1	• *(2,3) pair, since (2) is issued in the U pipe, and (3) is an MMX instruction*
4. PCMPEQB MM2, MM0	1	• *(4,5) do not pair because* mm2 *is the destination register and it is used in (5) as a source operand* (PR 5)*
5. PAND MM1, MM2 6. PANDN MM2, MM0	1	• *(5,6) pair*
7. POR MM1, MM2	1	• *(7,8) do not pair. (8) is an MMX instruction accessing memory (doesn't go in V pipe* [PR 3]*).*
8. MOVQ [EDI], MM1	2	• *(8) has a pipeline stall for one more cycle, since it is writing* mm1 *to memory, and* mm1 *was just updated. (SR 2)**
9. ADD EDI, 8 10. ADD ESI, 8	1	• *pair*
11. DEC ECX 12. JNZ DoQWord	1	• *pair*

* *PR* stands for "pairing rule"; *SR* stands for "sheduling rule."

PART VI

In the second table, Table 20-6, we show an optimized version of the instruction sequence above, where we reordered some of the instructions to fill in the empty slots.

TABLE 20-6 Optimized MMX Technology Sprite Loop Analysis

After (7 clocks)	⏲	Analysis Steps
DoQWord: 1. MOVQ MM0, [ESI] 2. MOVQ MM2, MM3	1	• *pair*
3. MOVQ MM1, [EDI] 4. PCMPEQB MM2, MM0	1	• *pair*
5. PAND MM1, MM2 6. PANDN MM2, MM0	1	• *pair*
7. POR MM1, MM2 8. ADD EDI, 8	1	• *pair*
9. ADD ESI, 8 10. DEC ECX	1	• *pair*
11. MOVQ [EDI-8], MM1	1	• *(11 & 12) don't pair because 11 is writing to memory and 12 is not an MMX instruction (PR 4)**
12. JNZ DoQWord	1	

**PR* stands for "pairing rule."

Note that the optimization resulted in a gain of 2 clocks. Pay special attention to the memory transfer instructions (1, 3, and 9 in the optimized loop). The table indicates that these instructions take only 1 or 2 clock cycles to execute. This is true if the address being accessed is in the L1 cache.[3] But if this is not the case, then it would take extra cycles to execute the instructions depending on where the data actually resides (L2 cache, uncached memory, video memory, and so forth).

Let's assume that the background image, pointed to by pDst, resides in video memory. Video memory is typically uncached and has a *very slow access pattern* relative to that of the fast processor. As a result, all reads and writes from/to video memory consume much longer than 1 clock. The 2-clock gain in the optimized sprite loop is very small compared to the time it takes to access video memory. In this case, the sprite sample is said to be I/O bound; that is, the CPU is just waiting for the memory to respond to its requests.

3. L1 cache is a small but very fast memory that resides on the processor itself. In contrast, the L2 cache is typically bigger, slower, and resides outside the processor.

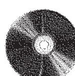
Table 20-7 shows measurements of both loops using the internal CPU clock cycle counter. We collected the measurement in eight buckets corresponding to the alignment of the sprite's top-left pixel on the screen. Please note that regardless of the alignment, the optimized version gave a small performance advantage over the nonoptimized version, which is the contribution of the gain of 2 clocks.

TABLE 20-7 Measured Cycle Timing of Both Nonoptimized and Optimized MMX Technology Sprite Loops

	Nonoptimized		Optimized	
Alignment	**Clocks/ Sprite**	**Clocks/ 8 Pixels**	**Clocks/ Sprite**	**Clocks/ 8 Pixels**
0	110407	159	109732	158
1	180585	260	179676	259
2	180425	260	179558	259
3	180546	260	179487	259
4	150358	217	149725	216
5	185099	267	184392	266
6	185399	267	184364	266
7	185398	267	184277	266

Nonetheless, each loop is consuming more cycles (158–267 clocks) than we expected from the static analysis (7–9 clocks). Again, this increase is attributed to the video memory access time being slow as compared with the processor access time.

Similar to the integer sprite we worked on in the previous chapter, unaligned memory accesses have a dramatic effect on the performance of the sprite. Note that we achieve the best performance when memory accesses are 8-byte aligned. Performance drops significantly when memory accesses are not aligned.



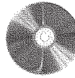
When we reimplemented the sprite sample to perform aligned memory writes, we received a huge performance gain—the sprite now executes at an average of 160 clocks/8 pixels. You can find the aligned sprite implementation on the companion CD.

20.8.1 MMX versus Integer Implementation of the Sprite

So how does this MMX implementation of the sprite compare to the integer implementation in the previous chapter? Not good! If you recall from the previous chapter, the WriteDWord command of the integer sprite could attain an average of 3 clocks/pixel for a full sprite.[4] This is about seven times faster than the sprite implementation we achieved using MMX technology! Let's have a closer look at the two sprite samples to understand why the difference is so marked and to figure out how to fix it.

The MMX sprite uses a Read/Modify/Write algorithm, which requires two accesses to uncached video memory: one for reading the initial bitmap, and one for writing the final result. The sprite in the previous chapter only accessed video memory once—when it wrote the visible pixels to the screen. The additional read from video memory degrades performance significantly for the MMX sprite.

Apparently, we made the wrong assumption about the location of the background image—it was fine for the integer sprite, but it's not appropriate for the MMX sprite. To speed up access to the background image with MMX, we decided to build the mixed sprite/background in video memory first and then send the mixed result to the screen. The only drawback here is that we have to allocate additional system memory to hold the mixed background.



The result of the new implementation is shown in Table 20-9. WOW, the MMX sprite is now faster than Speedy Gonzales, with an average of *1.7–1.8 clocks/pixel.* By moving the background to system memory, the read and write of the background image worked much faster than it did with the integer sprite—even when the integer sprite uses system memory. You can find a copy of this sample on the companion CD.

TABLE 20-8 Integer versus MMX Sprites Both Overlaid Either in Video or System Memory

	Integer (clocks/pixel)		MMX (clocks/pixel)	
Alignment	**Video**	**System**	**Video**	**System**
0	2.75	2	20	1.7
1–3	3.25	2.375	32	1.8
4	2.75	2	32	1.8
5–8	3.25	2.375	33	1.8

4. *Full Sprite* refers to a sprite that does not have any transparent pixels.

WHAT HAVE YOU LEARNED?

At this point you should have a good idea about the MMX technology, its instruction set, registers, pairing and scheduling rules, and EMMS. You should be able to manually optimize an MMX technology code fragment to obtain best performance.

Another important point to take from this chapter is that it is vital that you know your data. Know where it comes from, and where it goes to. We will talk more about this in the last chapter of this part.

REFERENCES

Intel Corporation. *The Complete Guide to MMX Technology.*

———. *Intel Architecture MMX Technology: Programmer's Reference Manual.*

———. *Intel Architecture MMX Technology: Developer's Manual.*

———. *Intel Architecture Optimization Manual.*

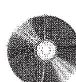
CHAPTER 21

VTune and Other Performance Optimization Tools

WHY READ THIS CHAPTER?

Your head must be steaming after reading the last couple of chapters. You're thinking: Boy! I wish there were a better way to optimize my code than this manual, tedious process! You are in luck. This chapter introduces some useful tools you can use to optimize your code for the Pentium processors. In this chapter you will

- become familiar with Intel's Visual Tuning Environment (VTune), which contains a few tools including static and dynamic code analysis, the hot-spot system monitor, and processor event counters;
- analyze the sprite in the MMX example using VTune and compare it to the previous results;
- learn how to count cycles using the internal Time Stamp Counter; and
- learn how to use the PMonitor event counter library to monitor internal processor events such as cache hits, misaligned accesses, and so forth.



You can find an evaluation copy of Intel VTune on the companion CD.

In addition to the scheduling rules discussed earlier, Intel added three programmable performance counters to provide an accurate method for measuring application performance on the Pentium processors. One counter measures the number of clock cycles executed by the processor, and the other two counters measure various internal events such as the number of data reads or writes, L1 cache hit rate, and so forth.

To program events into counters, the processor must be running in privileged level 0 (also known as ring 0). Therefore, you must write a ring 0 driver in order to be able to access these counters from a ring 3 application.

You might be thinking, "It's not enough that I have to remember all of these scheduling rules. Now I have to write a ring 0 driver." But you don't. That's why you're reading this chapter.

In this chapter you will learn about VTune—Intel's Visual Tuning Environment for Windows. With VTune you don't have to memorize the scheduling rules or write a ring 0 driver to access the performance counters. VTune remembers all the pairing and scheduling rules and provides you with a detailed analysis of your code. It can also provide you with a systemwide view of your application using either time-based sampling (TBS) or event-based sampling (EBS). You can accomplish all of this without any modification to your code.

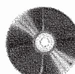
At the end of the chapter we will show you an alternate way of using the Time Stamp Counter and event counter. Unlike VTune, which monitors the entire application, this method allows you to monitor a specific portion of your code.

21.1 Overview of Performance Counters

Before we start with the actual tools, let's have a brief overview of the performance counters. The Pentium performance counters are the best means of getting accurate feedback about your application's performance. They give you insight into how the processor behaves when you run your application.

On a 200 MHz processor, it takes the TSC counter 2,924 years to roll over!

The Pentium processors include a 64-bit Time Stamp Counter (TSC), which counts the number of clocks executed by the processor. When the processor is reset, the TSC starts counting at zero. You can also program an initial value into the counter using the `WRMSR` instruction, which executes only in ring 0. Once the counter is started, you can sample its value using the `RDTSC` instruction at all processor privilege levels.



The Pentium processors also include two 40-bit counters (T0 and T1) that can be programmed to monitor various internal processor events that affect application performance. These events can be either *duration* events or *frequency* events. When monitoring duration events, the performance counters measure the *number of cycles while the event was active.* When monitoring frequency events, the event counters measure the *number of times the event occurred.* For a list of the types of events, refer to the *Intel Architecture Optimization Manual* found on the companion CD.

21.2

As with the TSC, you can program the event counters with the `WRSMR` instruction, which executes in ring 0. On the Pentium Pro and Pentium processors with MMX technology, you can read the event counters using the new `RDPMC` instruction at any privilege level. But on the Pentium processor without MMX technology, you can only use the privileged level instruction `RDMSR` to sample the event counters at ring 0.

21.2 Introducing VTune

VTune offers an easy-to-use Windows interface that simplifies optimization for the Intel Architecture. It is a collection of both simple and complex optimization methodologies that greatly help developers optimize their code for the Pentium, Pentium Pro, and Pentium with MMX technology processors.

In our discussion we will focus on the various VTune features without going into the details of how to use them. You can find more detail in VTune's online help files.

Let's start with a summary of VTune features and what they are used for. In Table 21-1, we list VTune features with a brief description of each. We also highlight the purposes of the features and how you can benefit from them. You can find out more details about these features later in the chapter.

TABLE 21-1 VTune Feature List

Feature	Description
Static analysis	Analyzes application (*.obj, .exe, or .dll*) and shows instruction pairing, warnings, and penalties for the selected processor. You can use static analysis in the first stage of instruction scheduling. Once you write your application, load the object file into VTune and examine scheduling issues.
Dynamic analysis	You can collect an *execution trace* of a range of instructions in your application. The execution trace is collected using the dynamic analyzer, and it represents the *actual* instructions executed. VTune analyzes the execution trace and presents you with a view that shows any potential problems with your application. The dynamic view includes details about BTB prediction, L1 code, and data cache hits, and other dynamic properties. You can use dynamic analysis to understand the branch and cache behavior of your application.

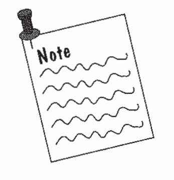
(Continued)

TABLE 21-1 VTune Feature List (Continued)

Systemwide monitoring	VTune can monitor all applications, drivers, and operating system components executing in the system. VTune gets control periodically from a timer interrupt (TBS) or from one of the internal event counters (EBS) and saves the instruction pointer where the interrupt occurred. It then displays a graphical view of the modules and their system usage. With systemwide monitoring you can determine which applications are consuming the most processor time or if the processor is idle.
Hot-spot analysis	This tool shows the percentage of CPU time spent executing each hot spot or active function relative to the execute time for the entire application or relative to the entire system. The analysis is based on the samples collected in *systemwide monitoring.* With hot-spot analysis you can zoom in on functions and instructions that take a long time to execute.

21.2.1 VTune Static Analysis

VTune's static analyzer is a smart tool that understands all the timing and scheduling rules of the Pentium processors. Basically, when you use static analysis, VTune analyzes the instructions in your code for pairing information and presents you with a simple view illustrating the results. We went through the tedious process of analyzing code in the last couple of chapters. With VTune you don't have to remember any of the pairing and scheduling rules, AGIs, or the number of micro-op codes per instruction. VTune does it for you.



It is much faster to use *.obj files for static analysis

The static analyzer accepts executable (*.exe*), object (*.obj*), and Dynamic Link Library (*.dll*) files. When you load one of these files, VTune disassembles the instructions in the file and applies the pairing rules to them.

You can see the static analysis view of the nonoptimized MMX sprite from the previous chapter in Figure 21-1. From the static analysis view, you can select the *Source View, Assembly View,* or a mix of both. You can also select which processor to consider for the analysis (Pentium, Pentium Pro, Pentium processor with MMX technology, or Pentium II). In addition, VTune provides an option for *Blended processor* analysis mode; that is where it displays any scheduling issues that affect any of the supported processors.

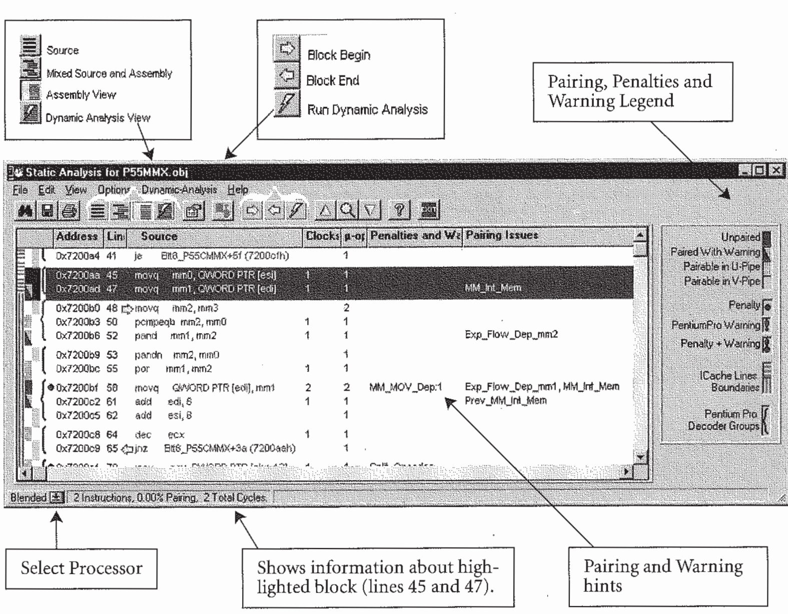
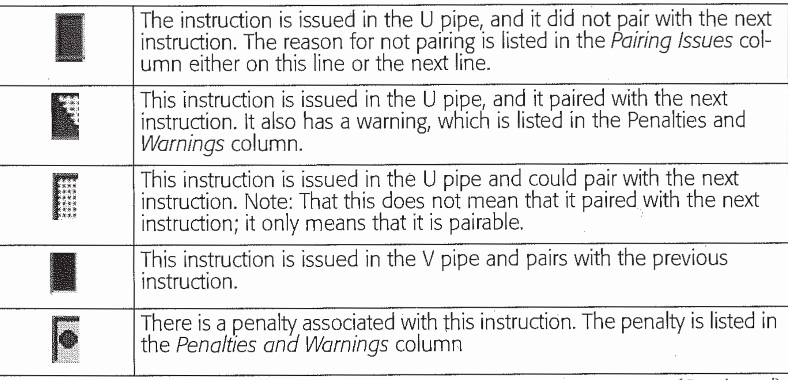


FIGURE 21-1 VTune static analysis view of the MMX sprite.

VTune presents information about each instruction in column format using either symbols, numbers, or descriptive pairing hints. Let's see what each of these columns or symbols mean. In the following table you can find a brief description of the symbols and columns of VTune's analysis view.

TABLE 21-2 VTune Symbols and Column Description



Symbol	Description
	The instruction is issued in the U pipe, and it did not pair with the next instruction. The reason for not pairing is listed in the *Pairing Issues* column either on this line or the next line.
	This instruction is issued in the U pipe, and it paired with the next instruction. It also has a warning, which is listed in the Penalties and *Warnings* column.
	This instruction is issued in the U pipe and could pair with the next instruction. Note: That this does not mean that it paired with the next instruction; it only means that it is pairable.
	This instruction is issued in the V pipe and pairs with the previous instruction.
	There is a penalty associated with this instruction. The penalty is listed in the *Penalties and Warnings* column

(Continued)

TABLE 21-2 VTune Symbols and Column Description (Continued)

	The warning listed in the *Penalties and Warnings* column affects only the Pentium Pro processors.
	There is a penalty and a warning associated with this instruction. It is listed in the *Penalties and Warnings* column.
	This indicates Instruction Cache Line boundaries. The instruction cache line size is 16 bytes for the Intel 486 and 32 bytes for the Pentium and Pentium Pro processors.
	The instructions included by these braces represent a Pentium Pro decoder group. The Pentium Pro decodes the instructions in a decoder group in 1 clock cycle. A decoder group can include up to three consecutive instructions where the first one is decoded to four or less micro-op codes, and the other two are decoded to one micro-op. This is the "4:1:1" sequence described in the Pentium II processor chapter (Chapter 22).
Address	This column shows the relative address of this instruction.
Line	This column shows the line number of the instruction in the source file.
Source	This column shows the assembly format of the instruction. In the mixed mode, the column shows the source line followed by the assembly instruction.
Clocks	For the Pentium processor, this column indicates the number of clocks it would take to execute this instruction. This, of course, assumes perfect L1 cache.
UOps	For the Pentium Pro processor, this column lists the number of micro-ops this instruction represents.
Penalties and Warnings	This column lists the shorthand explanation of a penalty or a warning. When you double-click the left mouse button on the line, you get more information about the warning.
Pairing Issues	This column lists the shorthand explanation of pairing issues related to this instruction. When you double-click on the line, you get more information about the pairing issue.

Notice that when you highlight a sequence of instructions, VTune displays the total number of cycles and instructions at the bottom status bar. For example, the highlighted instructions at lines (45–47) take 2 cycles to execute, and they have 0 percent pairing rate.

In Figure 21-1, to get more information about the warnings and penalties, you can double-click with the left mouse button on the problematic instruction. VTune pops up another window with more information about the problem (Figure 21-2). You can get even more explanations by selecting the help button Help ? associated with the problem.

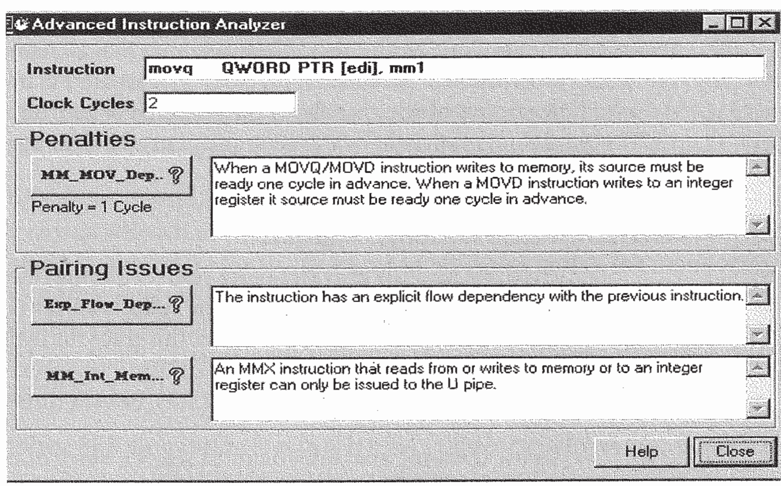


FIGURE 21-2 Explanation of problems on line 58 of the MMX sprite.

Now, let's compare the results of VTune's static analysis of the MMX sprite with our manual analysis in the previous chapter. For your convenience, we have duplicated the table from the previous chapter (Table 20-6). Notice that both methods yield the same number of clocks for each instruction and both reach the same conclusions about scheduling problems. But although it took us half an hour to figure this out, it took VTune less than half a minute to do the same.

TABLE 21-3 Nonoptimized MMX Sprite Manual Analysis

Before (9 clocks)	⏲	Analysis Steps
DoQWord: 1. MOVQ MM0, [ESI]	1	• *(1,2) do not pair because they both access memory*
2. MOVQ MM1, [EDI] 3. MOVQ MM2, MM3	1	• *(2,3) pair, since (2) is issued in the U-Pipe, and (3) is an MMX instruction*
4. PCMPEQB MM2, MM0	1	• *(4,5) do not pair because mm2 is the destination register and it is used in (5) as a source operand*
5. PAND MM1, MM2 6. PANDN MM2, MM0	1	• *(5,6) pair*
7. POR MM1, MM2	1	• *(7,8) do not pair. (8) is MMX instruction accessing memory (doesn't go in V-Pipe).*
8. MOVQ [EDI], MM1	2	• *(8) has a pipeline stall for one more cycle, since it is writing mm1 to memory, and mm1 was just updated.*
9. ADD EDI, 8 10. ADD ESI, 8	1	• *pair*
11. DEC ECX 12. JNZ DoQWord	1	• *pair*

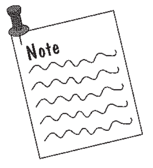
Let's have another look at the *clocks* column of the static analysis view in Figure 21-1. Notice that VTune assumes that all memory accesses take 1 clock cycle to execute. It also assumes that branch instructions take 1 clock regardless of whether the branch is taken or not. Since these assumptions are not always valid, VTune implements the dynamic analysis feature discussed below.

21.2.2 VTune Dynamic Analysis

Dynamic analysis provides more realistic timing information about your code. With dynamic analysis VTune collects an actual trace of instructions executed in your program and uses this trace for dynamic analysis. Since VTune knows exactly which instructions actually executed, it can provide better information about L1 cache hits, branch timing, and BTB hits.

To use dynamic analysis, you must select a block of instructions to analyze from the static analysis view (lines 48–65 in Figure 21-1). When you run the dynamic analyzer, VTune launches the application and collects a trace of the actual instructions executed within the selected block. When you terminate the application, VTune analyzes the collected trace and displays the result in the dynamic analysis view.

We are not showing the dynamic analysis view since it looks exactly the same as the static analysis view in Figure 21-1. The only difference is that the dynamic view displays the actual instruction pairing and a more realistic clock count. It also displays BTB hits, L1 code and data cache hits, and branch behavior.



In Windows 95, your system might hang if you use VTune dynamic analyzer in the middle of a *DirectDraw Lock* section. DirectDraw holds the *Win16Lock* between DirectDraw Lock and UnLock operations, which prevents VTune from running properly. The *Win16Lock* is a Windows 95 critical section that serializes access to GDI and USER system DLLs. As a result, the *Win16Lock* prevents Windows from running and blocks applications from using GDI or USER DLLs.

21.2.3 Systemwide Monitoring—Time- and Event-Based Sampling

So far, you've optimized your application and salvaged every wasted cycle in it. But do you know how your application behaves from the point of view of the entire system? What if your application calls an operating system or third-party function, do you know how long it takes to execute? Do you know where the CPU spends most of its time? *Simple.* Use VTune.

VTune includes a systemwide *time-* or *event-based sampling* (TBS or EBS) feature, which monitors every running component in the system. This includes operating system drivers (ring 0 and ring 3), DLLs, and other executables. VTune analyzes the time or event samples and presents a *percentage usage summary* for each module in a bar graph format (Figure 21-3).

When TBS monitoring is active, VTune gains control from a *periodic timer interrupt* where it records the instruction pointer (CS:EIP), *process ID,* and *module name* where the interrupt occurred. At the end of the monitoring session, VTune associates the collected pointers with their corresponding module and presents a *percentage usage summary* bar graph. The *y* axis of the bar graph represents the module name, and the *x* axis represents the percentage CPU usage of each module relative to the entire system.

When EBS monitoring is active, VTune gains control from a *performance counter event interrupt* where it records the instruction pointer (CS:EIP), *process ID,* and *module name* where the interrupt occurred. As with TBS, VTune associates the collected addresses with their corresponding module and generates a *percentage occurrence summary* bar graph. The *y*-axis of the bar graph represents the module name, and the *x*-axis represents the percentage of occurrence of that event within modules relative to the entire system.

We have compiled a few hints that are worth knowing about the systemwide monitoring feature in VTune:

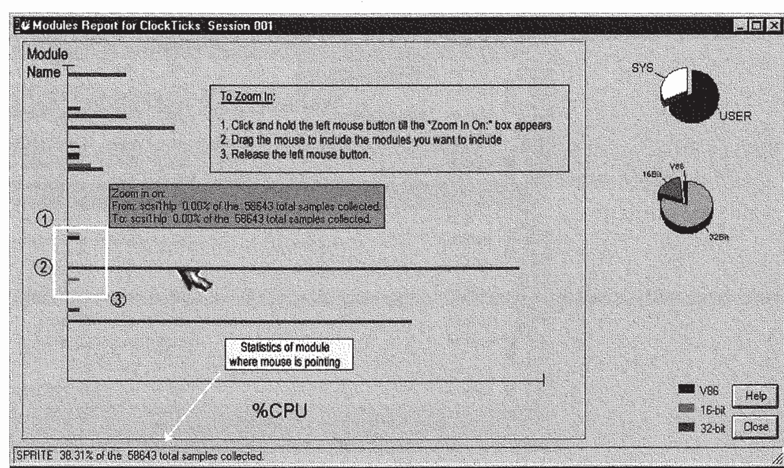


FIGURE 21-3 Systemwide monitoring module usage summary.

- For Pentium processors, you need a special processor socket to use event-based sampling (EBS) with VTune. But you don't require a special socket for the Pentium Pro processors. (Note: For Pentium processors, you can use PMonitor for event monitoring, and no special socket is needed.)
- When the mouse points to a module in the bar graph, VTune displays statistics about that module at the bottom status bar.
- You can pinpoint the amount of time that the operating system is *idle* (not executing any threads or tasks including yours). The VMM module reflects the idle CPU time in Windows 95, and the NTOSKRNL module reflects the idle CPU time in Windows NT.

When you zoom in on a function in one of the modules, VTune displays a time-based analysis view that shows the statistics for each instruction of the MMX sprite (Figure 21-4). When TBS is used, the Time column shows the hit rate of each of the instructions relative to the entire application. A high hit rate indicates that the instruction took a long time to execute.

Pay attention to the highlighted instruction on line 50 Figure 21-4. The Time column indicates that this instruction was executing 71 percent of the time when the timer interrupt occurred. But this is a simple instruction that uses only register operands and should execute in only 1 clock cycle. OK, let's look at the instruction in the previous cycle, specifically on line 47. If you remember from the previous chapter, this instruction reads data from

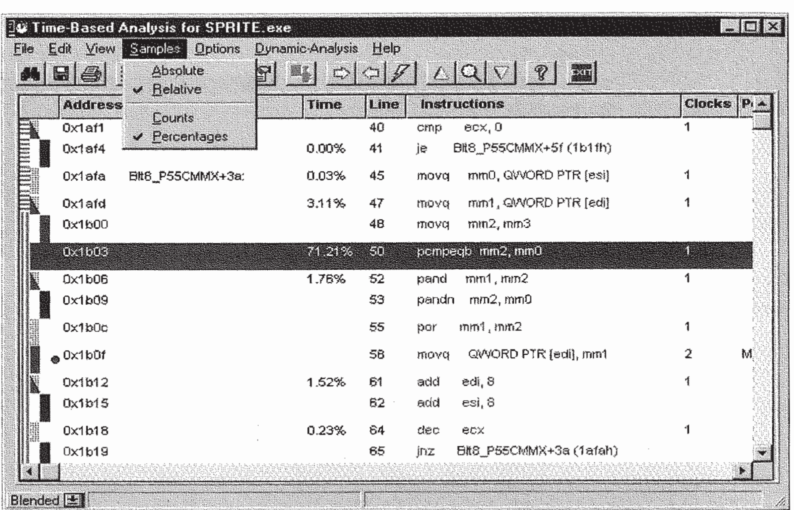


FIGURE 21-4 Time-based analysis view for the MMX sprite.

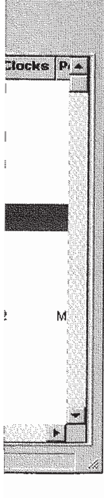
uncached video memory and takes a long time to execute. So it is likely that this instruction is the culprit spending 71 percent of the time! And it is.

Remember that VTune records the current instruction pointer when the timer interrupt occurs. When the interrupt occurs on line 47, the processor has to finish executing this instruction before it acknowledges the interrupt. But after the processor executes the instructions in lines 47 and 48 (they pair), it advances the instruction pointer to line 50 and then generates the interrupt. As a result, VTune records that line 50 was executing 71 percent of the time when the timer interrupt occurred.

You're thinking, Why doesn't VTune just adjust the instruction pointers to point to the previous instruction (or previous cycle)? VTune does not always know what the previous cycle was. For example, if the timer interrupt occurs in the middle of a branch instruction (CALL, JMP, JCC), the interrupt will occur at the branch target instruction (after it jumps). When VTune gets control, it has no idea that this is a branch target instruction, and if it is, VTune has no idea from where it was called.

USEFUL HINTS WHEN USING VTUNE

- VTune takes a long time to load an executable or a DLL because it analyzes the entire file, and that is a time-consuming process. So during the static analysis phase, you can load just the object file.
- When you run TBS, you need to stop it or set up the timer to stop after a certain time.



21.3 Read Time Stamp Counter

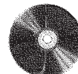
Now, let's see how we can use the Time Stamp Counter to measure a small or large portion of code using the RDTSC instruction. Since MSVC inline assembly does not recognize the RDTSC instruction, we implemented the instruction as an in-line function:

```
// Value returned in EDX:EAX which is the 64bit counter value.
1.    __inline __int64 ReadTimeStampCounter() {
2.    _asm xor eax, eax    // prevent compiler from optimizing around
RDTSC.
3.    _asm xor edx, edx
4.    _asm _emit 0x0f
5.    _asm _emit 0x31
6.    }
```

Now you can use the inline *ReadTimeStampCounter()* function to execute the RDTSC instruction. Notice that the RDTSC instruction returns its values in the *eax* and *edx* registers.

```
1.  Function()
2.  {
3.  __int64 qwStart;
4.  __int64 qwElapsedTime = qwStart;      // Force qwStart into L1 Cache
5.
6.  qwStart = ReadTimeStampCounter();
……. Code you want to measure
1.  qwElapsedTime = ReadTimeStampCounter() - qwStart;
2.  }
```

Notice that we read the qwStart variable in line 4 in order to eliminate a cache miss when we fill it with the initial value of the counter. In this case, qwStart was *preallocated* into the L1 cache. We intentionally preallocated qwStart so that we can minimize the side effects of the profiling code and achieve the most accurate results.



Depending on the amount of time it takes for your code to execute, you might want to calibrate the overhead of the RDTSC instruction (found on the CD) and subtract it from the measured time. This is necessary for measuring code fragments that take a small number of cycles to execute. In the following code we show how to calibrate the RDTSC instruction overhead using *CalibrateTimer()*:

21.4

```
main() {
    int nOverhead;
    // Invoke it once to bring in the code for the function into the
    // L1 code cache.  Then invoke it with a high counter value so it
    // would calibrate the RDTSC instruction.
    CalibrateTimer(1);
    CalibrateTimer(10000000);
    printf ("Overhead of RDTSC: %d\n", nOverhead);
}

int CalibrateTimer(int nIterations)
{
    __int64 iCounter;
    __int64 iOverhead = iCounter; // Force iCounter into the L1 cache

    // Run a counter loop executing only the RDTSC instruction
    // Figure out how much time that takes.
    iCounter = ReadTimeStampCounter();
```

```
    _asm {
        mov ecx, nIterations
LoopAgain:
        _emit 0x0f
        _emit 0x31
        dec ecx
        jnzLoopAgain
    }

    // Overhead of RDTSC loop
    iOverhead = ReadTimeStampCounter() - iCounter;
    // Now, figure out the loop overhead without the RDTSC instruction
    iCounter = ReadTimeStampCounter();

    _asm {
        mov ecx, nIterations
LoopAgain1:
        dec ecx
        jnzLoopAgain1
    }

    // Overhead of empty loop
    iCounter = ReadTimeStampCounter() - iCounter;

    // Overhead of one RDTSC instruction
    return (int)(iOverhead- iCounter)/ nIterations;
}
```

Again, notice that we called the function twice to avoid any cache misses—once to make sure the function code is loaded into the L1 code cache, and the other time to perform the actual calibration.

21.4 The PMonitor Event Counter Library

With the PMonitor library you can access the event counters from a ring 3 application. Unlike with VTune, you do not need a special socket on your Pentium processor to use PMonitor's event counters. Instead the PMonitor library implements a Windows 95 Virtual Device Driver (VxD), which executes in privileged level 0 to read the event counters.

To use the counters, first you need to program one or both counters with the events that you want to monitor. Once you start the counters, you can sample their values before and after the section of code that you want to measure.

For example, let's use counter 0 to measure the *total number of instructions executed,* and counter 1 to measure the *number of instructions executed* in the V pipe. First let's initialize and start the counters as follows:

PART VI

```
#include "Dll_If.h"      // PMonitor interface file
main()
{
    struct Pmon32Version Version;

    // Load and initialize the Pmon library
    DWORD dwPmon32Status = Pmon32Init(&Version);

    // Now program the two counters with the required events.
    if (dwPmon32Status == Pmon32_OK) {
        Pmon32Start(
            INST_EXECUTED,            Ring3,   // Counter 0 settings
            INST_EXECUTED_VPIPE,      Ring3    // Counter 1 settings
            );
    }
}
```

First we use *Pmon32Init()* to initialize the PMonitor library and make sure that it loads the VxD successfully. We then request that PMonitor program counter 0 to count the total instructions executed in the user level (ring 3) and counter 1 to count only the instructions executed in the V pipe (also in ring 3). Once the counters are started, we can use them to measure the two events as follows :

```
#include "dll_if.h"    // PMonitor Interface file
#define Get64bit(x) ((__int64 *)&x)[0]

SomeFunction()
{
    struct Pmon32Reply Start, End;

    Pmon32ReadCounters(&Start);
    **** Code To Profile
    Pmon32ReadCounters(&End);

    // Calculate the number of instructions executed.
    __int64 qwTotalInst = Get64bit(End.T0_l) - Get64bit(Start.T0_l);
    __int64 qwPipeInst = Get64bit(End.T1_l) - Get64bit(Start.T1_l);
}
```

```
Struct PMON32REPLY:
DWORD T0_l; // Counter 0 low
DWORD T0_h; // Counter 0 high
DWORD T1_l; // Counter 1 low
DWORD T1_h; // Counter 1 high
    ....
```

Notice that the *Pmon32ReadCounters()* function has a big overhead because it requires two ring transitions[1] to read the event counters—and that work consumes a lot of precious time. On the Pentium Pro and Pentium processors with MMX technology, you can eliminate such overhead by sampling

1. *Ring transition* refers to the switch between two privilege levels.

the counters with the new RDPMC instruction. So the above sequence of code can be changed as follows:

```
// Value returned in EDX:EAX which is the 64bit counter value.
__inline __int64 ReadPerformanceMonitorCounter(int nCounter) {
    _asm xor eax, eax           // Prevent compiler from optimizing -
    _asm xor edx, edx           // around RDPMC.
    _asm mov ecx, nCounter      // 0: Counter0, 1:Counter1
    _asm _emit 0x0f             // RDPMC
    _asm _emit 0x33
}

CSprite::Blt()
{
    __int64 qwTotalInst;
    __int64 qwVPipeInst = qwTotalInst;  // force qwTotalInst in L1 Cache.

    qwTotalInst = ReadPerformanceMonitorCounter(0);     // Counter 0
    qwVPipeInst = ReadPerformanceMonitorCounter(1);     // Counter 1

    **** Code To Profile

    // Calculate the number of instructions executed.
    qwTotalInst = ReadPerformanceMonitorCounter(0) - qwTotalInst;
    qwVPipeInst = ReadPerformanceMonitorCounter(1) - qwTotalInst;
}
```

WHAT HAVE YOU LEARNED?

We're positive that you would prefer to remember this chapter over the previous couple of chapters. Here are a few points to carry with you:

- VTune simplifies optimizing applications on the Pentium processors, but it does not do all the work for you.
- Start with static analysis for looking at the initial scheduling of your instructions.
- Use dynamic analysis to verify your scheduling assumptions and to understand branch and L1 cache behavior.
- Use systemwide monitoring and hot-spot view to zoom in on sections of time-consuming functions so you could optimize them if possible.
- To get more control over which pieces of code to profile, add timing code inside your application with TSC and event counters.
- Use PMonitor counters to gauge performance.

PART VI

CHAPTER 22

The Pentium II Processor

WHY READ THIS CHAPTER?

Are you ready for the latest Intel processor, the Pentium II processor? Do your applications run at their best on this new processor?

By reading this chapter, you will

- learn about the new features of the Pentium II processor and how to optimize your application for them;
- learn how to use Pentium II performance counters to measure various events that affect performance on the processor;
- understand how to properly use the Write Combining memory type to substantially speed up accesses to the video frame buffer and reduce the utilization of the system bus; and
- in the process of optimizing for the Pentium II processor, take a closer look at branch mispredictions, partial register stalls, and the 4:1:1 decoder template.

Simply put, the Pentium II processor is a Pentium Pro processor with MMX technology. In 1996 Intel introduced the Pentium with MMX technology processor, which adds MMX technology to the Pentium family of processors. In the middle of 1997 Intel extended the same technology to the Pentium Pro processor family with the introduction of the Pentium II processor. The Pentium II processor is well suited for both business and multimedia applications.

You may have noticed that we did not discuss the architecture of the Pentium Pro processor. As a matter of fact, since the Pentium II processor is derived from the Pentium Pro processor, any discussion of the Pentium II processor already incorporates the Pentium Pro processor—except for the MMX technology, of course. We'll point out the differences between the two processors early in the chapter.

We start the chapter with an architectural overview of the Pentium II processor, including a brief discussion of the internal operations of the components of the processor. We follow that with a more detailed explanation of each of the processor units and what's important for them to deliver optimal performance. For each unit we will give you a few guidelines or tips that could help you attain optimal performance on the Pentium II processor. If appropriate, we'll also include a list of useful internal event counters and an explanation of how you can use them to gauge the performance of that unit.

Finally, we'll show you how to use the *Write Combining* (WC) memory type[1] to speed up your graphics performance. WC is a new memory type that was first introduced in the Pentium Pro processor and will be widely available on systems using the Pentium II processor.

Wherever appropriate, we advise you to use VTune if we feel that it can help you with performance measurement and analysis. The latest release of Intel's VTune[2] includes support for the Pentium II processor.

22.1 Architectural Overview

As we mentioned earlier, the Pentium II is basically a Pentium Pro processor with MMX technology. The Pentium II processor moved to a twelve-stage pipelined architecture with an out-of-order execution core—as compared to the five-stage pipeline of the Pentium. In addition, the Pentium II processor includes three parallel decoders, five execution ports, a branch target buffer (BTB) with 512 entries, and four 32-byte Write buffers (see Figure 22-1).

---

1. Memory types include cached, uncached, Write Combining, and so forth.
2. We've included a three-month fully functional evaluation copy of VTune on the companion CD.

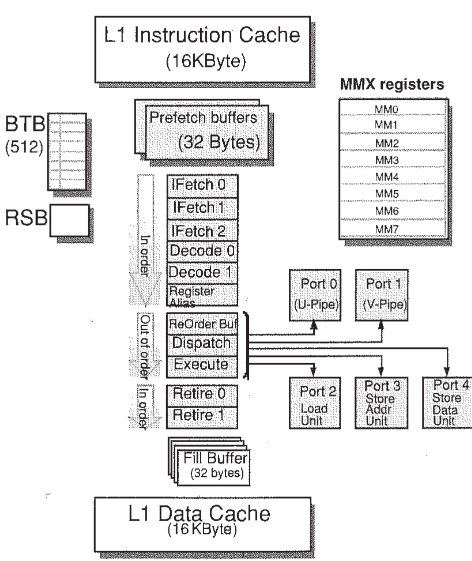


FIGURE 22-1 Architecture of the Pentium II processor.

Similar to the Pentium with MMX technology processor, the Pentium II processor doubled the size of the L1 instruction and data caches to 16K each and added eight MMX registers and a Return Stack Buffer (RSB).

22.1.1 The Life Cycle of an Instruction on the Pentium II

The Pentium II processor fetches instructions in a fashion similar to that of the Pentium processor. It uses the Branch Target Buffer (BTB) to predict branch behavior and prefetches instructions to one of the two 32-byte prefetch buffers.

Note

4:1:1 is the preferred decoder sequence.

The Pentium II processor includes three parallel decoders capable of processing up to three instructions in 1 clock cycle. The first one decodes instructions up to four micro-op codes long, and the other two can only decode instructions that are one micro-op long. In addition, the Pentium II processor includes a microcode sequencer that decodes complex instructions that are five or more micro-ops long.

The Register Allocation Table (RAT) accepts up to six micro-ops from the decoder and posts up to three micro-ops to the Reorder Buffer (ROB; a.k.a. the Reservation Station). For each micro-op, the RAT renames the logical IA-based registers to one of forty internal Pentium Pro registers and inserts them into the ROB. This is where the "out-of-order" processing begins.

The Reorder Buffer is the heart of the "out-of-order" execution. The ROB consists of forty "seats" where the micro-ops "hang out" waiting for one of the units to take care of them (they'll be dispatched, executed, or retired).

The dispatch unit determines when a micro-op is ready to execute based on the *readiness* of its data, not on the *order* in which it came in (since this is an out-of-order system). The dispatch unit marks a micro-op as "ready for execution" only when all of its operands are available.

The execution unit looks around the ROB for micro-ops that are ready to execute. Depending on the type of micro-op, one of the five execution ports executes it, marks it as "ready for retirement," and then places it back into the ROB. Note that the execution unit can execute up to five micro-ops in 1 clock cycle.

At this stage, the results of a micro-op are forwarded to other dependent micro-ops in the ROB. Also the results of branch instructions are determined, and if a branch was previously mispredicted, the fetch unit is directed to fetch instructions from the correct address, and all those mispredicted instructions are flushed out of the ROB, RAT, decoder, and fetch unit. In addition, the mispredicted instruction is logged into the BTB for better future branch prediction.

The retirement unit waits for micro-ops that are ready to retire. When a micro-op retires, its result is forwarded to the Memory Order Buffer (MOB), where it gets committed to the IA registers (*eax, ebx,* and so forth), the cache or main memory. The MOB guarantees that the results are committed in the order of the instructions as they came in. The retirement unit can retire up to three micro-ops every clock cycle.

22.1.2 Comparing the Pentium II with the Pentium Pro Processor

Following are the differences between the Pentium II and the Pentium Pro processors:

- The Pentium II processor adds fifty-seven new MMX instructions and eight MMX registers.
- The Pentium II doubles the size of the L1 caches to 16K each.
- The Pentium Pro processor has an on-chip L2 cache, which runs at the speed of the processor core. The Pentium II processor has the L2 cache off the chip, and it runs at one half to one third the speed of the core—the fraction depends on the frequency of the processor.

- Systems with the Pentium II processor have better support for the Write Combining (WC) memory type and thus better access to video frame buffers.

22.1.3 Comparing the Pentium II with the Pentium with MMX Technology Processor

The Pentium II processor has the same support for MMX technology as the Pentium with MMX technology processor. Fortunately, owing to architectural differences in its processor core, the Pentium II processor relaxes some of the scheduling constraints imposed by the Pentium with MMX technology processor. In Table 22-1 the left column lists the MMX scheduling rules of the Pentium with MMX technology scheduling, and the right column specifies whether such a rule applies to the Pentium II processor.

TABLE 22-1 Comparison of the MMX Instruction Scheduling Rules

<table>
<tr><th>Pentium with MMX Technology</th><th>Pentium II</th></tr>
<tr><td colspan="2">Two MMX shift or two MMX Multiply instructions cannot execute in the same cycle.</td></tr>
<tr><td>MMX instructions accessing memory or an integer register can only execute in the U pipe.</td><td rowspan="2">Both these rules don't apply to the Pentium II. You need only worry about the 4:1:1 decoder sequence discussed later in this chapter.</td></tr>
<tr><td>If the U pipe MMX instruction accesses memory or an integer register, the V pipe must hold an MMX instruction to pair.</td></tr>
<tr><td>The destination register of the U pipe instruction should not be accessed from the V pipe instruction.</td><td>This rule does not apply here because of the Pentium II's out-of-order execution.</td></tr>
</table>

22.2 Instruction and Data Caches

It is important to note the differences in cache architecture between the Pentium II processor and previous processors. As we mentioned earlier, the Pentium II processor doubled the size of the L1 caches (to 16K each) and moved the L2 cache off the chip (running at one half or one third the speed of the core). You might expect that moving the cache off the chip at a fraction of the speed could have a huge negative impact on application performance. Fortunately, doubling the size of the L1 cache positively outweighs the negative effect of moving the L2 cache off the chip.

Except for write misses, the cache behavior of the Pentium II processor is similar to that of the Pentium processor. On a write miss, the Pentium II processor first loads the cache line where the write miss occurred into the

L1 cache, and then it writes the data to the L1 cache. The Pentium processor, in contrast, writes the data through to the L2 cache or main memory without preallocation into the L1 cache.

One of the major enhancements of the Pentium II processor, over the Pentium processor, is that the read operations are nonblocking. As we mentioned in the Pentium chapter, the Pentium processor stalls completely when two back-to-back read misses occur—that is, it stalls until an entire cache line is brought into the L1 cache. The Pentium II processor, on the other hand, allows other micro-ops to execute while it's waiting for data to be brought in to the L1 cache—this improvement is made possible by the out-of-order execution model.

22.2.1 Operational Overview

The L1 cache is on-chip static memory that satisfies internal read/write requests more quickly than an external bus cycle to memory can. In addition, the L1 cache reduces the processor usage of the external bus, thus allowing other devices to move data on the bus—the DMA, bus maters, and so forth.

Similar to the Pentium with MMX technology processor, the Pentium II processor has two independent L1 caches (16K each): one satisfies data accesses, and the other satisfies instruction fetches. The two caches exist on two separate internal buses (each bus is 64 bits wide), which allows the processor to load instructions and data, simultaneously, in the same clock cycle. In contrast, the Intel 486 can only load either data or instructions, not both, at any given moment because both instructions and data have to share the L1 cache.

Both the instruction and data L1 caches are divided into 32-byte cache lines; this is the minimum granularity of the L1 cache. When the processor transfers any amount of data between the L1 cache and the external bus (main memory or the L2 cache), it transfers a minimum of one cache line at a time.

The read behavior of the Pentium II processor is identical to that of the Pentium processor. On a read or write hit, the L1 cache satisfies the request in 1 clock cycle. On a read miss, the processor transfers an entire cache line into the L1 cache. If a multi-byte read crosses a cache line boundary, the next consecutive cache line is also brought into the L1 cache.

But the write miss behavior of the Pentium II processor is different from that of the Pentium processor. On a write miss, the Pentium II processor first loads the entire cache line where the write miss occurred into the L1 cache and then writes the data to the L1 cache. This behavior is useful for applications that exhibit spatial data locality and access more than one element in a cache line—such as applications that involve sequential access of an array or access of local function variables.

22.2.2 Performance Considerations

To put it simply, "Reuse it while it's in the L1 cache." If you have already brought in code or data from main memory to the L1 cache, make sure that you use it while it's still there—before it gets flushed out. Here are a few suggestions on how to get good performance on the Pentium II processor.

- *Keep the size of your inner loops below 16K.* If your most executed loop does not fit in the L1 code cache, the L1 cache will thrash continuously. To fix this problem, you can break the task at hand into smaller tasks with smaller loops that fit within the L1 cache. To find out the size of your loop, you can either look into the map file generated by the linker or use VTune's static analyzer. You should also watch out for in-line macros and functions that, if used often, could bloat your code.
- *Reuse data while it's in the L1 cache.* If possible, operate on the data while it's in the L1 cache—before it gets flushed out. Since multimedia data does not typically fit in the L1 cache, you can operate on some part of the data at one time rather than the full set. For example, instead of decoding the entire video frame in one loop, you can decode the top half of the frame from start to finish and then the bottom half—or whatever part of the frame fits in the L1 cache.

22.3 Instruction Fetch Unit

22.3.1 Operational Overview

The Pentium II processor (Figure 22-2) has an aggressive prefetcher with two 32-byte prefetch buffers that operate in conjunction with the branch target buffer (BTB) to fetch raw opcodes from the L1 cache, L2 cache, or main memory (see section 19.4 for more information about the operation of the BTB).

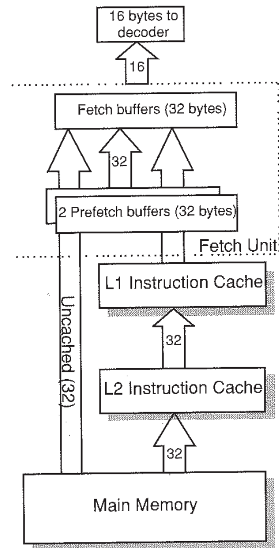


FIGURE 22-2 Pentium II fetch unit.

22.3.2 Performance Considerations

Typically, you do not have to worry about the performance of the fetch unit because the Pentium II processor uses an aggressive prefetcher, deep branch prediction, and has a large L1 instruction cache. The combination of prefetching and branch prediction allows the processor to determine the correct execution path and have instructions ready for execution ahead of time. The larger L1 cache improves the chance of a cache hit when the processor fetches raw opcodes from the L1 cache, which can deliver 32 bytes in 1 clock cycle.

Nonetheless, we've listed a few guidelines that could help you attain optimal performance from the fetch unit point of view:

- *Keep the size of inner loops less than 16K.* If the size of inner loops does not fit in the L1 cache, the cache will thrash. As a result, fetches are satisfied from the L2 cache or main memory, both of which are much slower than the L1 cache. To fix the problem, you can break the task at hand into smaller tasks with smaller loops that fit within the L1 cache. To find out the size of your inner loop, you can either look into the map file generated by the linker or use VTune's static analyzer.
- *Align heavily executed loops and branch and function labels on the 16-byte boundary.* By labels we're referring to the address of the branch when the branch instruction is taken. The idea here is to fill up the execution pipeline quickly after a branch is taken. By aligning the beginning of an exe-

cution block on 16-byte boundaries, you guarantee that there will be enough opcodes to feed the three parallel decoders and, hence, quickly fill up the ROB with micro-ops for the execution unit to work on.

- *Avoid interleaving code with data such as jump tables.* Because of aggressive prefetching, the processor could end up decoding data unnecessarily if it is mixed with code.
- *Reduce the number of mispredicted branches.* Mispredicted branches can have a drastic effect on the Pentium II processor because of the deep pipelining architecture: it will take more clocks to propagate new micro-ops to the execution unit. The delay is even worse if the branch target is *not* in the L1 cache—since it takes longer to fetch raw opcodes and thus takes longer to feed the pipeline.

  Depending on the state of the processor, the effect of mispredicted branches on the fetch unit could be hidden if the branch target is in the L1 cache. On a mispredicted branch, the entire processor core becomes busy trying to recover from the false branch prediction. If the branch target is in the L1 cache, the fetch unit typically has enough time to fetch the branch target instructions while the rest of the units are busy recovering. But if the target branch is out of the L1 cache, the fetch unit cannot fetch the correct instructions in time to satisfy the other stages of the pipeline, so they just stall.

22.3.3 Fetch Performance with Event Counters

You can use the processor's internal event counters[3] to measure the efficiency of the fetch unit as shown in Table 22-2 and Figure 22-3. In the figure you can see where each of the counters is sampled by the processor. Notice that all instruction fetches or misses represent a 32-byte quantity. For example, the IFU_Fetch counter increments by one every time the fetch unit loads 32 bytes of instructions from anywhere.

You can use these counters to determine how well your critical loops fit in the L1 and L2 caches. The following equations may give you some insight into where the fetch unit is getting its instructions.

$$\text{\% External Fetches (L2 and uncached)} = \frac{\text{IFU_IFetchMiss}}{\text{IFU_IFetch}}$$

This percentage gives you an indication of the actual instruction fetches that missed the prefetch buffer and the L1 cache. These unexpected fetches are probably caused by a branch misprediction or an interrupt.

3. See Chapter 21 for more about using VTune or PMonitor for event counter measurement.

TABLE 22-2 Pentium II Instruction Fetch Unit Performance Event Counters

Event Counter	Usage
IFU_Fetch	Number of all fetches including cached and uncached fetches.
IFU_Ifetch_Miss	Number of fetch misses that miss the prefetch buffer and the L1 cache. This number also includes uncached fetches.
L2_Ifetch	Number of cached fetches that miss the L1 cache. So this is the number of L2 cache fetches.
BUS_Tran_IFetch	Number of cached fetches that miss the L2 cache. It does not include uncached fetches that always go to the bus.
IFU_Mem_Stall	Number of cycles that the instruction fetch unit is stalled for any reason.

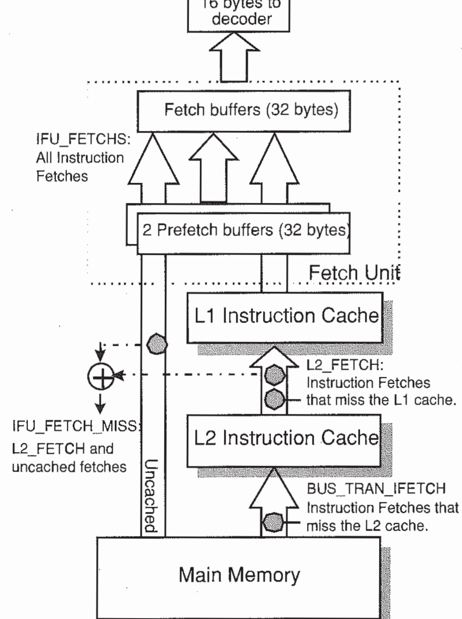


FIGURE 22-3 Sampling of the event counters by the Pentium II processor.

$$\text{\% External Fetches from L2} = \frac{\text{L2_IFetch}}{\text{IFU_IFetch}}$$

This percentage gives you an indication of the number of "demand fetches" that could not be satisfied from the prefetch buffers or the L1 cache. Notice that this number does not indicate all fetches from L2, only the demand fetches. The fetch unit has a stream buffer that continuously fetches

instructions from the L2 cache or main memory to keep the IFU fed properly; this happens during normal operations. Since these fetches do not hinder application performance (they are actually good fetches), they are not counted by the L2_IFetch event counter. This counter only counts instruction fetches that miss the prefetch buffer and the L1 cache, because these fetches affect application performance.

$$\%\text{ External Fetches from System Memory} = \frac{\text{BUS_TranIFetch}}{\text{IFU_IFetch}}$$

This equation indicates the percentage of fetches that came from the system bus. In this situation the instructions could not be found in either cache or the prefetch buffer. This typically happens with applications executing from uncached memory.

22.4 Branch Prediction and the Branch Target Buffer

22.4.1 Operational Overview

The Pentium II processor features a deep branch prediction mechanism that enables the processor to better predict the outcome of branch instructions. This mechanism employs a Branch Target Buffer (BTB) that can hold up to 512 branch addresses of previously mispredicted branch instructions. (You can find a detailed discussion about branch prediction and the BTB in section 19.4.)

The Pentium II processor has a static prediction algorithm similar to the Pentium processor's with one exception: backward branch instructions that are not in the BTB are predicted as taken in the Pentium II processor; the Pentium processor assumes that all branch instructions not in the BTB are not taken.

22.4.2 Performance Considerations

Branch misprediction is one of the first issues that you should consider when you are optimizing for the Pentium II processor. When a branch is mispredicted, the Pentium II processor has to flush the entire pipeline and start fetching the correct instructions. With a deep pipelining architecture, twelve stages, the new instructions take more clock cycles to propagate from the fetch unit to the execution unit, making branch misprediction more costly than with the Pentium processor, a five-stage processor.

The Pentium exhibits a different behavior for backward branches not found in BTB.

You can determine how long it takes the processor to execute branch instructions, assuming that instruction opcodes are already in the L1 cache. With the exception of backward branches, all branch instructions that are not in the BTB are predicted not taken, *including unconditional branch instructions.*[4] However, backward branch instructions that are not in the BTB are predicted taken. Use Table 22-3 to determine, on average, how many clocks it takes to execute a branch instruction. Notice that the table assumes that the instructions of the correct branch address are already in the L1 code cache. If they aren't, it takes much longer to fetch the instructions from the L2 cache or main memory.

TABLE 22-3 Pentium II Processor Branch Behavior

Predicted	Unconditional (clock cycles)	Conditional (clock cycles)
Correctly	1	1
Incorrectly	6	9–15

Now that you know how the branch prediction unit and the BTB operate, we'll leave you with a few suggestions that could help you minimize branch mispredictions in your code:

- *Minimize branch misprediction in your code.* You can either use VTune's dynamic analyzer or the performance event counters to pinpoint portions of your code that are highly affected by branch mispredictions. You can then rearrange your code to achieve better branch prediction behavior.
- *Use Conditional Move `CMOVXX`, `FCMOVXX` instructions.* If possible, use these instructions to eliminate some of the branches in your application. For example, you can use `CMOVZ` to eliminate a branch as follows:

```
   mov eax, 0          mov eax, 0
   dec ecx             dec ecx
   jnz  Continue       CMOVZ eax, 1
   mov eax, 1
Continue:
```

- *Try to fit code with high branch misprediction within the L1 cache.* If the correct target branch instructions reside in the L1 cache, the fetch, decode, and RAT units can typically recover from the branch misprediction while the execution unit is still recovering from the misprediction.

4. Refer to section 19.4 for more details about branches.

22.4.3 Branch Performance with Event Counters

You can use the Pentium II event counters to determine the behavior of branch instructions within your code. Table 22-4 lists the important event counters for branch prediction.

TABLE 22-4 Event Counters for Fetch Unit Instructions on the Pentium Pro

Event Counter	Usage
BR_Inst_Decoded	Number of branch instruction decoded.
BR_Inst_Retired	Number of branch instructions retired.
BR_BTB_Misses	Number of branch instructions encountered with no history of the branch target address in the BTB.
BR_MissPred_Retired:	Number of branch mispredicted branch instructions that eventually executed and retired.

You can measure the percentage of mispredicted branches in your code as follows:

mispredicted branches = BR_MissPred_Retired / BR_Inst_Retired

You will also want to measure the percentage of branch instructions within your code as follows:

branch instructions = BR_Inst_Retired / Inst_Retired

To get an accurate assessment of your branch misprediction, you must have both values. If the percentage of branch instructions within the code is very small, then the percentage of mispredicted branches is insignificant regardless of how high it is.

22.5 Instruction Decoders

22.5.1 Operational Overview

The Pentium II processor features three parallel instruction decoders that can decode up to three instructions generating up to six micro-ops in 1 clock cycle. The complex decoder processes instructions of four or less micro-ops, and the simple decoders process only instructions of one micro-op (4:1:1 decoder template described below). The micro-code sequencer handles instructions greater than four micro-ops in length. The generated micro-ops are forwarded to the Register Alias Table (RAT) for further processing.