AMD

3DNow!™ Instruction Porting Guide

Application Note

Publication # **22621** Rev: **B**
Issue Date: **August 1999**

Contents

Revision History

Date	Rev	Description
August 1999	B	Initial public release.

Application Note

3DNow!™ Instruction Porting Guide

Introduction

This document contains information to assist programmers in creating optimized code for AMD processors with 3DNow!™ technology. Compiler and assembler designers and assembly language programmers writing execution-sensitive code sequences as well as high-level C programmers will also find the guidelines useful. This document assumes that the reader possesses in-depth knowledge of the x86 instruction set, the x86 architecture (registers, programming modes, etc.), and the IBM PC-AT platform.

This document has three sections of guidelines for 3DNow! porting:

- 3DNow!™ Instruction Porting
- Blended Code Guidelines
- General Porting Guidelines

The 3DNow! Instruction Porting section describes the actual process of converting existing code to 3DNow! code. The Blended Code Guidelines section deals specifically with the creation of blended code—3DNow! code that provides high performance on AMD-K6® processors as well as on the AMD Athlon™ processor. New applications should use blended code to ensure optimal performance on current and future platforms. The General Porting Guidelines section describes a number of important issues for 3DNow! code optimization mainly for the family of AMD-K6 processors, but also addressing the AMD Athlon processor.

Detecting 3DNow!™ Technology Support

3DNow! technology is an open standard that has been adopted by multiple processor vendors. Therefore, checking for 3DNow! technology capability should not be limited to AMD processors. All 3DNow! technology licensees have agreed to indicate 3DNow! technology capability through bit 31 of the extended feature flags. Checks for 3DNow! technology support can be made without first checking for the processor vendor. This allows current detection code to also detect future 3DNow! technology licensees.

The basic steps of the 3DNow! technology capability detection are as follows:

1. Test that the processor has the CPUID instruction.
2. Check that CPUID instruction also supports extended function 8000_0001h.
3. Execute CPUID extended function 8000_0001h and retrieve the EDX register.
4. If bit 31 of the EDX register is set, the processor supports the 3DNow! instruction set.

The following assembly language code shows how this can be implemented:

```
;; check whether CPUID is supported
:: (bit 21 of Eflags can be toggled)
pushfd                ;save Eflags
pop   eax             ;transfer Eflags into EAX
mov   edx, eax        ;save original Eflags
xor   eax, 00200000h  ;toggle bit 21
push  eax             ;put new value of stack
popfd                 ;transfer new value to Eflags
pushfd                ;save updated Eflags
pop   eax             ;transfer Eflags to EAX
xor   eax, edx        ;updated Eflags and original differ?
jz    NO_CPUID        ;no diff, bit 21 can’t be toggled
```

```
;;test whether extended function 80000001h is supported
mov   eax, 80000000h   ;call extended function 80000000h
cpuid                  ;reports back highest supported ext.
                       ; function
cmp   eax, 80000000h   ;supports functions > 80000000h?
jbe   NO_EXTENDED      ;no 3DNow! support, either

;;test if function 80000001h indicates 3DNow! support
mov   eax, 80000001h   ;call extended function 80000001h
cpuid                  ;reports back extended feature flags
test  edx, 80000000h   ;bit 31 in extended features
jnz   YES_3DNow!       ;if set, 3DNow! is supported
```

Related Documents

Related documents can be downloaded at the following URL:

`http://www.amd.com/support/techdocdir.html`

Including:

- *AMD-K6® Processor Code Optimization Application Note,* order# 21924
- *3DNow!™ Technology Manual,* order# 21928
- *AMD-K6® Processor Multimedia Technology,* order# 20726
- *Implementation of Write Allocate Application Note,* order# 21326
- *AMD Athlon™ Processor x86 Code Optimization Guide,* order# 22007
- *AMD Extensions to the 3DNow!™ and MMX™ Instruction Sets Manual,* order# 22466
- *AMD Processor Recognition Application Note*, order# 20734

3DNow!™ Instruction Porting

Code Support Considerations

Consider how your software can support several paths through the different code optimized for various processors. Choices include the following methods:

Separate Executables

Build separate executables optimized for each platform. This is probably the highest performance option, but can be impractical due to code distribution issues and other problems.

Separate DLL

Place all performance-sensitive code into a separate DLL, providing several DLLs optimized for each target platform to be supported. This is a high-performance solution as the overhead is typically no more than selecting and loading the DLL version most appropriate for the platform detected at run time. The problem with this approach is that the performance-sensitive code can come from different and unrelated parts of the source tree, but becomes grouped together in a single DLL.

Different Optimized Versions

Provide optimized versions of each performance-critical function for each target platform, and call the functions through pointers that are initialized at run time based on the system processor the software is running on. This has a negative performance impact on AMD-K6® processors because function calls through pointers are slower than regular function calls.

Conditional Code Paths

Inside performance-critical parts of the code, conditionally select code paths based on capability flags. On AMD-K6 processors, this can be faster than the approach using function pointers, because the branches will be well predicted since the capabilities do not change during run time. On the other hand, this approach can make the code less clear and more difficult to maintain.

3DNow!™ Porting Preparations

Perform High-Level Optimizations

Before starting a 3DNow! porting effort, perform all high-level optimizations that can be done at the source-code level. This primarily affects loops, which can be transformed in a variety of ways for better performance—loop unrolling, loop splitting, loop merging, loop inversion, loop switching, and hoisting of loop invariant expressions and conditionals. Function calls can also be optimized by inlining. It is much more difficult to perform high-level transformations once the code has been ported to the assembly-language level.

Profile Existing Code

Before starting the actual porting process, profile the existing code on the target platform to identify the hotspots that merit manual porting work.

Profilers come in various types. Some require source code, some can instrument binaries, others use a sampling approach. All profilers should work on AMD processors. VTUNE works too, but doesn't have event-based profiling or disassemble 3DNow! code capabilities. Some profilers, like Metrowerks™ CATS, have built-in support for 3DNow! instructions and can be easier to use when reprofiling code during the porting process.

Port Major Hotspots

Candidates for 3DNow! porting are hotspots that frequently use x87 floating-point unit (FPU) instructions. AMD-K6 processors incur a penalty called switching overhead whenever the instruction flow changes between the use of x87 instructions and MMX™/3DNow! instructions (or vice versa). For full 3DNow! optimization, port all x87 code down to hotspots that take up only a small percentage (approximately 2%) of the total execution time. Due to switching overhead, porting a few small functions to 3DNow! can often be detrimental to overall performance. The goal is to keep the processor operating on 3DNow!/MMX code for long periods of time, with only occasional use of x87 code.

Some manual porting work can be saved by compiling the code which contains fewer hotspots with a compiler that can generate native 3DNow! code, such as Metrowerks CodeWarrior™ Professional Release 4 and later. At this time, major hotspots require manual porting for optimal performance.

Use Compiler Optimizations

To achieve the best performance from hotspots that are not floating-point intensive and so do not lend themselves to 3DNow! porting, experiment with compiler flags to find which flag settings provide the best code for AMD processors. Most compilers allow processor-specific optimizations based on the capabilities of Intel processors. Since AMD processors are different from Intel processors, the available processor-specific settings are not fully optimal for AMD processors. The microarchitecture of AMD processors most closely resembles the P6, Pentium® II, and Pentium III microarchitecture, and in most cases selecting P6/PII/PIII-specific optimization results in the highest performance for AMD processors (for example, -G6 for Microsoft® Visual C/C++). The Metrowerks CodeWarrior compiler has a specific optimization setting for AMD processors.

Use MASM Code for Critical Code

Use standalone MASM code for performance-critical parts of code that are ported to 3DNow!. This gives the best control over the code (for example, code alignment). To assemble 3DNow! code, use MASM 6.13 or MASM 6.14. Upgrade from an existing installation of MASM 6.11 to 6.13 by downloading ML613.EXE from the following ftp site:

```
ftp://ftp.microsoft.com/Softlib/MSLFILES
```

Apply this patch. To enable MMX instructions, use the .MMX directive. To enable 3DNow! instructions, use the .K3D directive after using the .MMX directive. It is order dependent.

MASM 6.14 supports most of the new 3DNow! and MMX extensions introduced in the AMD Athlon processor. Use the .XMM directive to enable the use of these new extensions. Note that the new instructions, PFNACC and PFPNACC are not yet accessible in MASM 6.14. Also, in order to use the new PSWAPD instruction, users need to define the text macro as follows:

```
pswapd TEXTEQU <pswapw>
```

For some big functions where only a small part of the C code is replaced, use inline assembly. Since Microsoft Visual C 5.0 does not have native inline assembly support for the 3DNow! instruction set, download instruction macros from the AMD web site. The macros are in the amd3d.h file in the 3DNow! SDK, which can be downloaded from the following URL:

```
http://www.amd.com/3dsdk/index2.html
```

To get started on assembly language code, have the C compiler generate an assembly language listing and use that as the initial assembly language version. Make sure to compile with maximum optimizations to have the compiler perform all the high-level optimizations up front. The compiler will convert symbolic constants in the source code to “magic numbers”; however, the programmer may need to set up a mechanism to extract symbolic constants from C code and import them into the assembly code to maintain the assembly code as well as the C code.

Port Code in Blocks

Most functions contain several more-or-less self-contained blocks. Port blocks one-by-one to 3DNow! and surround the 3DNow! code with FEMMS. This block-by-block approach minimizes debug time. After each block is ported, run the code to verify that it is still working. If the code is not working, it’s usually easy to locate errors because they are isolated to a block. With this approach a debugger is only rarely necessary in 3DNow! porting work. The commenting conventions for 3DNow! code show the most significant half of the operand on the left hand side, the least significant half of the operand on the right hand side, with the halves separated by a vertical bar.

3DNow!™ Code versus x87 FPU Code

When porting, most programmers find that 3DNow! code is much easier to write than x87 FPU code because the register file is flat and because with the 3DNow! single instruction multiple data (SIMD) capability twice as many operands can be manipulated. It is often possible to remove local temporary variables.

Maximize the use of SIMD—always try to do useful work on both parts of the operands. It can be advantageous to add overhead to pack and unpack operands in order to use the SIMD arithmetic. Consider modifying existing data structures so the data layout is more conducive to SIMD processing, thereby eliminating the need for additional pack and unpack instructions.

Replace integer code with MMX code. Unroll small loops completely. This can free up integer registers, and branches that do not exist cannot be mispredicted. Due to the large number of global history bits, the AMD-K6 processor does not predict well on many short loops. If possible, use computations to replace branches caused by "if...then...else" constructs acting on 3DNow! data. Branching on 3DNow! data is a bit slower since 3DNow! instructions don't affect the integer flags. Also, branching is disruptive to SIMD code as it is an inherently scalar operation which diminishes the advantages of SIMD processing.

Avoid moving data between the integer and the MMX registers because this is time-consuming on the AMD-K6 and AMD Athlon™ processors. To move data between the integer and the MMX registers, use the MOVD instruction. Write MMX and 3DNow! code in a load/store construction—but do not use load execute instructions such as PFADD MM0, [FOO]. Using a load/store construct enables aggressive scheduling which is essential for good performance. (See Schedule Instructions on page 11.)

Maximize the use of instructions that guarantee high decode bandwidth. These are called short-decode instructions on AMD-K6 family processors and DirectPath for AMD Athlon family processors. The optimization guides for both processors list the short-decode or DirectPath instructions. Maintaining a high decode bandwidth is essential for high performance code. Using short-decoded instructions, the AMD-K6 family processors can decode two instructions per cycle. Using DirectPath instructions, the AMD Athlon family processors can decode three instructions per cycle. On the AMD-K6 family processors, the only 3DNow!/MMX instructions that are not short-decoded are EMMS, FEMMS, and PREFETCH.

Avoid indirect calls and jumps, as the AMD-K6 processors do not apply branch prediction to these control-transfer instructions. At the source code level, this affects functions called through a function pointer (such as entry points into DLLs). The latency of a JMP DWORD PTR is eight cycles, and the latency of a CALL DWORD PTR is seven cycles. Note that AMD-K6 processors use the return stack on indirect calls, so the return from an indirectly called routine is still accelerated. The AMD Athlon processor applies branch prediction to indirect calls and jumps.

Optimize Register Allocation

After porting a complete function, optimize register allocation across the function. Keep as much data as possible in registers to reduce overall memory traffic. Make sure all data is aligned to natural boundaries—QWORDs on QWORD boundaries, DWORDs on DWORD boundaries. Note that data that is accessed by 3DNow! code as QWORDs is not necessarily declared as QWORDs in the program, and therefore can not be properly aligned even if compiler switches are used to force data alignment to natural boundaries. Ensuring alignment can require slight changes or padding to data structures outside the ported code, and can require manual QWORD alignment of pointers returned by dynamic memory allocation routines such as malloc(), calloc(), etc. Use the /zp8 switch on Microsoft Visual C to pad and align structs to QWORD boundaries. Note however that /zp8 doesn't always do a perfect job, so a small amount of manual padding may still be needed.

Schedule Instructions

Schedule the code according to instruction latencies. Scheduling is important for AMD-K6-2 and AMD-K6-III processors because their scheduler is six deep and four wide, and it holds 24 OPs. OPs are pushed into the scheduler four OPs (an op-quad) at a time. As new OPs come in at the top, the previous lines shift down. When a line reaches the bottom of the scheduler and the OPs haven't all completed yet, the scheduler stalls—no new OPs can be pushed in at the top. If all OPs have completed and the line is at the bottom of the scheduler, the results of the OPs are committed to architectural state (retired) and the op-quad is discarded from the scheduler, allowing the following lines to shift down.

In the best possible case, the decoders push in a new op-quad every cycle. The OPs must complete after six cycles or else the processor loses performance. The 24 OPs are equivalent to 12 short-decoded x86 instructions. So, the out-of-order window is not very big, and an instruction that doesn't get its source operands right away can get to the bottom of the scheduler without having completed, this prevents the scheduler from shifting.

There are a few basic scheduling rules. All 3DNow! instructions in the AMD-K6-2 and AMD-K6-III processors have two-cycle latency. All MMX instructions have one-cycle latency, except MMX multiplies which are two cycles. Loads have two-cycle latency. To guarantee smooth flow of code through the machine, group instructions into pairs that can decode together, issue together, and retire together. To achieve this, observe the following rules:

- No dependencies between instructions in a decode pair
- No resource conflicts between instructions in a decode pair

Per cycle, the AMD-K6-2 and AMD-K6-III processors can perform the following:

- One load
- One store
- Two integer ALU operations
- One integer shift
- Two MMX ALU operations
- One MMX shift
- One 3DNow! add pipe op
- One 3DNow! mul pipe op
- One branch

LEA counts as a store op. PUNPCK* instructions are MMX ALU instructions.

One scheduling method is to first group the code following the above rules, marking the empty slots with <> and then move instructions to fill the slots. For example:

```
movd       mm1, [foo_var]      ;0 | v[3],v[2],v[1],v[0]
<>
<>
<>
punpcklbw  mm1, mm0 ; 0,v[3],0,v[2] | 0,v[1],0,v[0]
<>
movq       mm2, mm1 ; 0,v[3],0,v[2] | 0,v[1],0,v[0]
punpcklwd  mm1, mm0 ;    0,0,0,v[1] | 0,0,0,v[0]
punpckhwd  mm2, mm0 ;    0,0,0,v[3] | 0,0,0,v[2]
pi2fd      mm1, mm1 ;   float(v[1]) | float(v[0])
pi2fd      mm2, mm2 ;   float(v[3]) | float(v[2])
```

3DNow!™ Code Debugging

To debug 3DNow! code, it is best to have a debugger that supports both the disassembly of 3DNow! instructions, and allows the MMX registers to be viewed as pairs of single-precision floating-point values. NuMega SoftICE version 3.24 and later has both these capabilities. Microsoft Visual C/C++ 6.0 can also disassemble 3DNow! instructions; however, it does not provide a convenient way of viewing the MMX registers as pairs of floating-point numbers.

Decode Degradation Checking

After code has been scheduled and thoroughly tested, the last stage of tweaking is to make sure there is no decode degradation. All AMD-K6 processors use a technique called pre-decode to speed up decoding. In certain instances, the pre-decode information can be degraded, resulting in decode of only one instruction per cycle (long decode) or even one instruction per two cycles (vector decode), even though the instruction itself is listed as short decoded. Use the following guidelines for AMD-K6 family processors:

[ESI] Inhibits Short Decode

Use of [ESI] addressing mode inhibits short decode. Note that [ESI+disp], [ESI+reg] etc. is acceptable. Also, note that specifying [ESI+0] is optimized by most assemblers to [ESI].

Instructions Longer Than Seven Bytes

If the length of an instruction exceeds seven bytes, short decode is inhibited, and the instruction can never be short decoded.

Crossing Cache Line Boundary

If an instruction crosses a cache line boundary and the opcode byte and modR/M byte are not in the same cache line, short decode is inhibited. Instruction cache lines are 32-bytes long in the AMD-K6 family processor. If the code segment is only paragraph (16-byte) aligned, check all 16-byte boundaries for the occurrence of this case. Bad cases can be remedied as follows:

- Swap instructions in a decode pair.
- Choose alternative instructions to move code.
  (For example, use CMP EAX, 0 instead of TEST EAX, EAX)
- Insert filler instructions like NOPs. Since an instruction degraded to vector decode takes up two cycles, it's better to add an additional instruction and have both be short decoded.
- Hand code an instruction to add a zero displacement or to make a displacement 32 bits instead of 8 bits.

Instruction Length Determination

Short-decode is inhibited if more than three instruction bytes are required to determine the length of an instruction. This happens for certain SIB addressing modes where the decoder needs to look at the SIB byte to determine instruction length, but 0Fh, opcode, and modR/M already make up the maximum of three bytes. Avoid these SIB addressing modes. For more information, see the *AMD-K6 Processor Code Optimization Application Note*, order# 21924. AMD-K6-2 processors with the CXT core (CPUIDs of 588h to 58Fh) and AMD-K6-III processors eliminate this particular form of degraded predecode.

Align Loops on 32-Byte Boundary

Align important loops on a 32-byte cache line boundary. At the minimum, make sure that after the start of the loop there are at least two instructions before the next 32-byte boundary.

Blended Code Guidelines

Introduction

Blended code is 3DNow!™ optimized code that runs well on both the AMD-K6® and AMD Athlon™ processor platforms. The basic approach to blended code optimization is to address the AMD-K6 processor requirements first, and then to look for specific AMD Athlon processor improvements and issues which do not adversely affect AMD-K6 processor performance.

With much larger buffers and a much larger out-of-order instruction window than other x86 processors, the AMD Athlon processor is good at automatically extracting performance out of existing executables, even if they are specifically optimized for a different processor. Of course, the best AMD Athlon performance can be achieved by optimizing code to exploit the specific strengths of the AMD Athlon processor.

To learn more about AMD Athlon code optimization, refer to the *AMD Athlon™ Processor x86 Code Optimization Guide*, order# 22007.

Data Alignment

Data alignment is very important for both AMD-K6 and AMD Athlon processor performance. Standard processor designs will work to their full potential if data is aligned. Alignment is specially important for data that is written by one instruction and subsequently read by another instruction. Three typical areas to watch for data alignment are:

- Alignment of structures and structure components
- Alignment of dynamically allocated memory
- Alignment of stack data

Alignment of Structures

With regard to alignment of structures, many compilers offer switches to automatically pad and align structures. These switches do not always work perfectly. It is best to check the alignment and to pad manually if necessary.

Alignment of Structure Components

Arranging structure components in order of decreasing size may help. For example, declare components with larger base type (e.g., DWORD) ahead of components with smaller base types (e.g., BYTE).

Alignment of Dynamically Allocated Memory

With regard to alignment of dynamically allocated memory, if your programming environment does not guarantee pointers returned by dynamic memory allocators, such as malloc(), to be suitably aligned, allocate a slightly larger chunk of memory and align the pointer manually. For example, a QWORD alignment should be:

```
p=(QWORD *)malloc(sizeof(QWORD)*number_of_qwords)+7L);
np=(QWORD *)((((long)(p))+7L) & (-8L));
```

Alignment of Stack Data

Alignment of stack data is hard to control unless the complete functions are written in assembly language. In this case, use code like the following example to keep local 3DNow! data QWORD aligned.

```
Prolog:
PUSH    EBP
MOV     EBP, ESP
AND     ESP, -8
SUB     ESP, size_of_local_variable
```

Note: *Use EBP to access arguments, use ESP to access local variables.*

```
Epilog:
MOV     ESP, EBP
POP     EBP
RET
```

Maximize SIMD Processing

Maximize the amount of SIMD processing in your code. If the programmer exploits the SIMD nature of the 3DNow! instructions aggressively, using 3DNow! instructions in the code can provide significant performance benefits as compared to x87 code.

Using PUNPCK instructions to combine scalar data for SIMD processing can create significant overhead and should be avoided where possible. It is best to rearrange computations and data structures in the source such that the amount of SIMD computation can be maximized.

Example 1 (Avoid):

```
float  Xscale, Xoffset, Yscale, Yoffset;
xnew = x*Xscale+Xoffset;
ynew = y*Yscale+Yoffset;
```

Example 2 (Better):

```
float  Xscale, Yscale, Xoffset, Yoffset;
xnew = x*Xscale+Xoffset;
ynew = y*Yscale+Yoffset;
```

The second example can now be efficiently implemented using 3DNow! instructions:

```
MOVQ  mm0, x          ;y | x
MOVQ  mm1, Xscale     ;Yscale | Xscale
MOVQ  mm2, Xoffset    ;Yoffset | Xoffset
PFMUL mm0, mm1        ;y*Yscale | x*Xscale
PFADD mm0, mm2        ;y*Yscale+Yoffset | x*Xscale+Xoffset
MOVQ  xnew, mm0       ;store ynew | xnew
```

As a rough goal, strive to use 90% or more of the available computational slots provided by the SIMD instructions.

Use PREFETCH and PREFETCHW Instructions

Use PREFETCH and PREFETCHW as aggressively as possible. On the AMD-K6-2 processor, use of PREFETCH results in only a small performance improvements, because the prefetches share the frontside bus (FSB) bandwidth. However, due to the high FSB utilization at high core-clock multipliers, the prefetches often get bumped because they are a low priority memory access. This situation improves with the AMD-K6-III processor, where L2 traffic is redirected to a separate backside bus, which frees up FSB bandwidth.

The AMD Athlon processor has large amounts of FSB bandwidth available, and application-level improvements of up to 20% have been observed using PREFETCH(W) aggressively. Examine code carefully to find opportunities for using PREFETCH(W). Good use of PREFETCH requires that essentially all of the prefetched data is actually used, and it therefore works best if data is accessed with unit stride and in ascending order. Sometimes algorithms can be rewritten to create such a data access pattern. On the AMD-K6 processor, PREFETCH creates a small overhead, since it is a vector decode instruction. On the AMD Athlon processor, PREFETCH is DirectPath.

Use PREFETCH as aggressively as possible without decreasing AMD-K6 processor performance due to the overhead of the PREFETCH instruction. This is possible in almost all cases. PREFETCH on the AMD Athlon processor brings in 64 bytes per PREFETCH due to the cache line length having doubled over the AMD-K6 processor (32 bytes versus 64 bytes), but it is acceptable to have overlapping (on the AMD Athlon processor) prefetches to account for the shorter 32-byte cache lines of the AMD-K6 processor. Make sure to prefetch to addresses at least 64 bytes apart from the target address of any stores in the vicinity of a PREFETCH(W) instruction. Also, for best AMD Athlon performance, prefetch about three cache lines (192 bytes) ahead of current loads. For a more detailed formula, see the PREFETCH usage guideline in the *AMD Athlon™ Processor Code Optimization Guide,* order# 22007.

Take Advantage of Write Combining

Make sure to take advantage of the write-combining mechanisms provided by the hardware. For the AMD-K6 processor, the best performance is achieved by using software write combining. (See “Software Write Combining” on page 34.) Also enable the write-combining features provided by the hardware on the AMD-K6-2 processor with the CXT core and on the AMD-K6-III processor. Aggressive software write combining can often do a better job than the AMD-K6 processor’s hardware write-combining mechanism, but enabling the hardware write-combining mechanism provides the additional benefit of shorter latency writes to non-cacheable memory areas.

The AMD Athlon processor has a very powerful write-combining mechanism that achieves even better acceleration of writes to non-cacheable space than is possible with write combining on the AMD-K6 processor. Specifically, the AMD Athlon write-combining buffer is 64 bytes and can combine writes of any size. The programming of the write-combining hardware is through model-specific registers (MSRs), which have been implemented compatibly with the Intel Pentium II processor. In addition to accelerating writes to write-combining (WC) regions, the AMD Athlon write combining can also accelerate writes to write-through (WT) memory areas *if they occur in strictly ascending order.* (Writes to WC areas can be combined regardless of the order of the writes.) See the Write Combining chapter for the *AMD Athlon™ Processor Code Optimization Guide,* order# 22007 for more details.

Use FEMMS Instruction

The AMD Athlon processor does not have any switching overhead when switching between 3DNow!/MMX instructions and x87 instructions. Also, the FEMMS and EMMS instructions are essentially free because they execute with apparent zero-cycle latency. However, for blended code it is important to avoid frequent switching between 3DNow!/MMX and x87 code blocks and to use FEMMS before entering and after leaving a block of

3DNow!/MMX code. Otherwise, AMD-K6 processor performance can suffer significantly.

Load-Execute Instruction Usage

The AMD Athlon processor performs well when load-execute instructions (i.e., instructions that have a register and a memory source, where the result goes to a register) are used. In fact, use of load-execute instructions is recommended for the AMD Athlon processor because they improve code density. However, for blended code, do not use load-execute instructions in 3DNow!/MMX code to enable proper scheduling of loads and to avoid potential problems with load-execute instructions (degradation to vector decode due to instruction length) on AMD-K6 family processors. The AMD Athlon processor has a built-in mechanism that enables a sequence of a load and a dependent 3DNow!/MMX instruction to execute just as quickly as a load-execute instruction. Avoiding load-execute instructions does not cause performance degradation on the AMD Athlon processor but can help the AMD-K6 processor.

Scheduling Instructions

Schedule instructions for the AMD-K6 processor. (See "Schedule Instructions" on page 11.) Due to the relatively small instruction re-order buffer in the AMD-K6 processor, instruction scheduling is important for maximizing performance on AMD-K6 processors. However, the AMD Athlon processor is a very aggressive out-of-order machine with a huge instruction re-order buffer. Therefore, instruction scheduling on the AMD Athlon processor is of minor importance, because the CPU can extract the available parallelism automatically. Scheduling code for the AMD-K6 processor has no adverse side effects on AMD Athlon performance.

Instruction and Addressing Mode Selection

As far as instruction selection is concerned, only a few issues require attention. On the AMD Athlon processor, transferring data between integer and MMX registers is somewhat slower than on the AMD-K6 processor. Therefore, such transfers should be minimized. Usually, this is not difficult to do.

Among the integer instructions, avoid the LOOP instruction. While very fast on the AMD-K6 processor, it is somewhat slower on the AMD Athlon processor. It should be replaced with the sequence DEC ECX;JNZ. This will, in most cases, not reduce AMD-K6 performance, and if so, only to a very limited amount.

The AMD Athlon processor uses a different instruction pre-decode scheme than the AMD-K6 processor. It therefore has no sub-optimal addressing modes. However, since this is a real performance issue on the AMD-K6 processor, addressing modes considered sub-optimal for the AMD-K6 processor should be avoided in blended code. Sub-optimal addressing modes are described in "Addressing Modes on the AMD-K6®-2 and AMD-K6®-III Processors" on page 36.

General Porting Guidelines

Minimize AMD-K6®-2 Processor Switching Overhead

Minimize the FPU to 3DNow!™ and MMX™ switching overhead by porting all hotspots containing x87 code to 3DNow! code. Even if FEMMS is used, switching incurs about 25 cycles in each direction—50 cycles round-trip. Always use FEMMS, and not EMMS, as the switching overhead with EMMS is about 100 cycles round-trip.

Always bracket 3DNow! code with FEMMS to ensure proper operation and minimize switching overhead. If there are function calls to functions that can contain FPU code, bracket the function call with FEMMS.

It is also beneficial to simply minimize the number of FEMMS. One technique to use if there are multiple calls to a DLL (where the functions are _stdcall), is to perform the following in order:

- Push all the arguments first
- Execute a FEMMS
- Call all the functions (which unload the stack)
- Execute another FEMMS

Since FEMMS is a three-cycle vector path instruction, functions should not be made very small to avoid adding significant overhead (functions have been observed in OpenGL that consist of just five x86 instructions).

Note: *For the AMD Athlon processor, it is important that CALLs not be spaced too closely together. No more than two CALLs for every 16 bytes of code are recommended.*

The switching overhead occurs on the first floating-point unit instruction after a piece of 3DNow!/MMX code, and it occurs on the first MMX or 3DNow! instruction after a piece of x87 code. FEMMS and EMMS are 3DNow!/MMX instructions. Thus, looking at the following sample code:

```
code                           cycles
<FPU instructions>
FEMMS                          3 + switching overhead
<MMX/3DNow! instructions>
FEMMS                          3
<1st FPU instruction>          x + switching overhead
```

Note that PREFETCH(W), although introduced as part of the 3DNow! instruction set extension, is treated like an ordinary integer instruction and therefore never incurs switching overhead. PREFETCH(W) can be used to accelerate integer, x87, MMX, or 3DNow! code.

Using PREFETCH

Use PREFETCH judiciously. PREFETCH on the AMD-K6®-2 and AMD-K6®-III processors is microcoded, so it adds some overhead. Also on the AMD-K6-2 processor, all cache and memory accesses have to flow through the same frontside bus. Do not waste bandwidth on the frontside bus by executing useless prefetching.

Opportunities for using PREFETCH are typically inside loops that process large amounts of data. If the loop goes through less than a cache line of data per iteration, partially unroll the loop. Make sure that close to 100% of the prefetched data is actually being used. This usually requires unit stride access—all accesses are to contiguous memory locations.

PREFETCH on the AMD-K6® Processor

The usefulness of PREFETCH on AMD-K6-III processors is limited by hardware constraints, the most important is that the AMD-K6-III processor allows only one load miss to be outstanding at any time.

The cases where PREFETCH can most likely provide benefits are characterized as follows:

- The bandwidth requirements of the code are moderate—there is a relatively large amount of computation and relatively few memory accesses. An example of moderate bandwidth requirements would be code that consumes about 250 Mbytes per second worth of data when running out of the L1 cache on a 400-MHz processor.
- Stores in the code that access cacheable memory write to a small area of memory only—the working sets for stores is small or empty. Due to the write-allocate feature of the AMD-K6-2 and AMD-K6-III processors, stores bring lines into the cache which are subsequently dirtied and must be written back from the cache when the cache line is replaced with data brought in by PREFETCH. Cache writebacks use up bandwidth on the front-side bus.
- PREFETCHes do not overlap—no two PREFETCH instructions try to bring in the same data.
- The number of distinct memory regions being prefetched is small, preferably only one region—if there are multiple memory regions being prefetched (like multiple source arrays), the density of the loads must be low compared to the amount of computation, such that the computation can be overlapped with each PREFETCH. The PREFETCH instructions should be scheduled separately in such cases to allow each to overlap with computation, and to avoid the first PREFETCH blocking subsequent PREFETCHes due to the limit of one load miss in the machine at any time.

PREFETCH on the AMD Athlon™ Processor

PREFETCH on the AMD Athlon™ processor is a very powerful tool both because of the much larger available bandwidth that it can exploit and because of the ability to have multiple outstanding load misses.

PREFETCHW Usage

Code that intends to modify the cache line brought in through prefetching should use the PREFETCHW instruction. While PREFETCHW works the same as a PREFETCH on the AMD-K6-2 and AMD-K6-III processors, PREFETCHW gives a hint to the AMD Athlon processor of an intent to modify the cache line. The AMD Athlon processor will mark the cache line being brought in by PREFETCHW as modified. Using PREFETCHW can save an additional 15-25 cycles compared to a PREFETCH and the subsequent cache state change caused by a write to the prefetched cache line.

Multiple Prefetches

Programmers can initiate multiple outstanding prefetches on the AMD Athlon processor. While the AMD-K6-2 and AMD-K6-III processors can have only one outstanding prefetch, the AMD Athlon processor can have up to six outstanding prefetches. For example, when traversing more than one array, the programmer should initiate multiple prefetches.

Example (Multiple Prefetches):

```
double a[A_REALLY_LARGE_NUMBER];
double b[A_REALLY_LARGE_NUMBER];
double c[A_REALLY_LARGE_NUMBER];

for (i=0; i<A_REALLY_LARGE_NUMBER/4; i++) {
    prefetchw (a[i*4+64]); // will be modifying a
    prefetch (b[i*4+64]);
    prefetch (c[i*4+64]);
    a[i*4]   = b[i*4]   * c[i*4];
    a[i*4+1] = b[i*4+1] * c[i*4+1];
    a[i*4+2] = b[i*4+2] * c[i*4+2];
    a[i*4+3] = b[i*4+3] * c[i*4+3];
  }
```

Determining Prefetch Distance

To make sure code with PREFETCH works well on the AMD Athlon processor, prefetch several cache lines ahead of the current loads. A good heuristic is to fetch three AMD Athlon cache lines (at 64 bytes each), or 192 bytes ahead of current loads. That is, if the code is currently operating on data at address X, prefetch at X+192.

Given the latency of a typical AMD Athlon processor system and expected processor speeds, the following formula should be used to determine the prefetch distance in bytes:

Prefetch Distance = 200 ($^{DS}/_{C}$) bytes

- Round up to the nearest 64-byte cache line.

- The number 200 is a constant that is based upon expected AMD Athlon processor clock frequencies and typical system memory latencies.
- DS is the data stride in bytes per loop iteration.
- C is the number of cycles for one loop to execute entirely from the L1 cache.

Prefetch at Least 64 Bytes Away from Surrounding Stores

The PREFETCH and PREFETCHW instructions can suffer from false dependencies on stores. If there is a store to an address that matches a request on bits 14–6, that request (the PREFETCH or PREFETCHW instruction) is blocked until the store is written to the cache. Therefore, code should prefetch data that is located at least 64 bytes away from any surrounding store's data address.

If PREFETCH helps on a piece of code, but doesn't affect the AMD-K6-III processors, keep the PREFETCH code anyway. There is a good chance that it will help on the AMD Athlon processor, because the AMD Athlon processor's implementation of PREFETCH is very aggressive. If an AMD Athlon processor is available, check that it benefits from the PREFETCH, and then make sure that the PREFETCH doesn't hurt the AMD-K6-III processor.

Use PFSUBR Instruction When Needed

Note that there is a PFSUBR instruction, so in a subtraction the programmer can choose which operand to destroy.

Using PAND and PXOR

Use PAND and PXOR to perform FABS and FCHS work on 3DNow! operands. For example:

```
mabs DQ 07fffffff7fffffffh
sgn  DQ 08000000080000000h
movq mm0, [mabs]
movq mm1, [sgn]
pxor mm2, mm1    ;change sign
pand mm2, mm0    ;absolute value
```

Use a `PXOR MMreg, MMreg` instruction to clear the bits in an MMX register.

Use a `PCMPEQD MMreg, MMreg` instruction to set the bits in an MMX register.

Swapping MMX™ Registers Halves

To swap the two register halves of an MMX register (which should be avoided) use the following:

```
;mm1 = swapd (mm0), mm0 destroyed
movq      mm1, mm0     ;y | x
punpckldq mm0, mm0     ;x | x
punpckhdq mm1, mm0     ;x | y

;mm1 - swapd (mm0), mm0 preserved
movq       mm1, mm0    ;y | x
punpckhdq  mm1, mm1    ;y | y
punpckldq  mm1, mm0    ;x | y
```

For code being used only on AMD Athlon family processors, use the new PSWAPD instructions. See the *AMD Extensions to the 3DNow!™ and MMX Instruction Sets Manual,* order# 22466 for the instruction usage.

PUNPCKL* and PUNPCKH* Instructions

PUNPCKL* and PUNPCKH* are essential facilities for manipulating MMX and 3DNow! operands. Besides MOVQ/MOVD, these are the most frequently used MMX instructions in 3DNow! code. For example, converting a stream of unsigned bytes into 3DNow! floating-point operands:

```
; outside loop:
pxor       mm0, mm0

;inside loop:
movd       mm1, [foo_var]   ;0 | v[3],v[2],v[1],v[0]
punpcklbw  mm1, mm0         ;0,v[3],0,v[2] | 0,v[1],0,v[0]
movq       mm2, mm1         ;0,v[3],0,v[2] | 0,v[1],0,v[0]
punpcklwd  mm1, mm0         ;0,0,0,v[1] | 0,0,0,v[0]
punpckhwd  mm2, mm0         ;0,0,0,v[3] | 0,0,0,v[2]
pi2fd      mm1, mm1         ;float(v[1]) | float(v[0])
pi2fd      mm2, mm2         ; float(v[3]) | float(v[2])
```

Storing the Upper 32 Bits of an MMX™ Register

To store the upper 32 bits of an MMX register using MOVD, one can use either a PSRLQ or a PUNPCKHDQ instruction to move the high-order 32 bits of the register to the low-order 32 bits of the register. In this situation, it is optimal to use the PUNPCKHDQ instruction. The AMD-K6-III processor has only one MMX shifter (which can execute a PSRLQ), but two MMX ALUs (which can execute a PUNPCKHDQ). Using PUNPCHDQ therefore maximizes the likelihood of an execution unit being available.

PFMIN and PFMAX

Use PFMIN and PFMAX where possible. They are much faster than the equivalent code using MMX and 3DNow! instructions. PFMIN and PFMAX can be used for clamping. They can also be used in SIMD code that avoids branching by replacing it with computation. For example:

```
float x,z;
z = abs(x);
if (z >= 1) {
   z = 1/z;
}
```

can be coded using branchless SIMD code as follows:

```
;;in:  mm0 = x
;;out: mm0 = z
movq      mm5, mabs  ;0x7fffffff
movq      mm6, one   ;1.0
pand      mm0, mm5   ;z=abs(x)
pcmpgtd   mm6, mm0   ;z < 1 ? 0xffffffff : 0
pfrcp     mm2, mm0   ;1/z approx
movq      mm1, mm0   ;save z
pfrcpit1  mm0, mm2   ;1/z step
pfrcpit2  mm0, mm2   ;1/z final
pfmin     mm0, mm1   ;z = z < 1 ? z : 1/z
```

Another example. The following code:

```
#define PI 3.14159265358979323f
float x,z,r,res;
z = abs(x)
if (z < 1) {
   res = r;
}
else {
   res = PI/2-r;
}
```

This can be code as branchless SIMD code as follows:

```
;;in:   mm0 = x
;;      mm1 = r
;;out:  mm1 = res
movq      mm5, mabs  ;0x7fffffff
movq      mm6, one   ;1.0
pand      mm0, mm5   ;z=abs(x)
pcmpgtd   mm6, mm0   ;z < 1 ? 0xffffffff : 0
movq      mm4, pio2  ;pi/2
pfsub     mm4, mm1   ;pi/2-r
pandn     mm6, mm4   ;z < 1 ? 0 : pi/2-r
pfmax     mm1, mm6   ;res = z < 1 ? r : pi/2-r
```

Precision Considerations

Carefully consider whether to use reciprocals, divides, square roots, and reciprocal square roots to full precision. If full precision is not required, accelerate code by using just the approximations returned by PFRCP (14 bits accuracy), and PFRSQRT (15 bits accuracy) instead of coding the reciprocal or reciprocal square root sequence with the Newton-Raphson step instructions. For lighting computations, the accuracy of the approximation instructions often suffices, but geometry transforms typically require full precision.

Moving Data Between MMX™ and Integer Registers

For the AMD Athlon processor, avoid moving data between MMX and integer registers or vice versa. If this cannot be avoided, use the MOVD instruction to accomplish the transfer, and do not pass the data manually through memory (except where the store can be scheduled at least 15 instructions ahead of the load).

Store-to-Load Forwarding

Avoid any store-to-load forwarding (store feeding into load) that does not have address and size matches. The only exception is a wide store feeding into a small load where the addresses match:

```
movq   [foo], mm0
mov    eax, [foo]
```

Here are some cases to avoid:

```
mov    [foo], eax
mov    [foo+4], edx
movq   mm0, [foo]
movq   [foo], mm0
mov    eax, [foo+4]
movq   [foo], mm0
movq   [foo+8], mm1
movq   mm2, [foo+4]
```

Block Copies

For memory block copies on the AMD-K6-III processor, most code will have very similar performance for large blocks, because it is limited by the bus interface. For the AMD-K6-2 processor, this was verified by creating multiple block copy types and discovering that there were insignificant performance differences. This is also true for block copies inside L2 (for off-chip L2). However, in L1-to-L1 block copies there can be a big difference.

The following are measurements performed with an AMD-K6-2/300 on an Epox motherboard with VIA MVP3 chipset and PC100 DRAM. Data blocks are QWORD aligned.

```
MSV 5.0      memcpy()          aggressive MOVQ loop
L1-to-L1     985 MB/s               1718 MB/s
L2-to-L2     122 MB/s                124 MB/s
mem-to-mem    71 MB/s                 72 MB/s
```

The L2-to-L2 and mem-to-mem throughput increases with the AMD-K6-III processor and further increases on the AMD Athlon processor.

The aggressive MOVQ loop performs at a minimum as well as a memcpy(), and does much better for L1-to-L1 transfers. It is also preferable for copies to non-cacheable areas on the AMD-K6-III processor due to the doubled chunk size over the REP MOVSD inside the memcpy() function. For this reason, consider using it for all block copies. The code is as follows:

```
_asm { mov   eax, [src]
mov   edx, [dst]
mov   ecx, (SIZE >> 6)

xfer:
movq  mm0, [eax]
add   edx, 64

movq  mm1, [eax+8]
add   eax, 64

movq  mm2, [eax-48]
movq  [edx-64], mm0

movq  mm3, [eax-40]
movq  [edx-56], mm1

movq  mm4, [eax-32]
movq  [edx-48], mm2

movq  mm5, [eax-24]
movq  [edx-40], mm3

movq  mm6, [eax-16]
movq  [edx-32], mm4

movq  mm7, [eax-8]
movq  [edx-24], mm5

movq  [edx-16], mm6
dec   ecx

movq  [edx-8], mm7
jnz   xfer    }
```

Care should be taken to make the label *xfer:* 32-byte-aligned for maximum performance. As a side note, the Microsoft Visual C 5.0 *without* Service Pack 3 appears to ignores align directives in inline assembly. This problem may not occur after applying Service Pack 3 to Microsoft Visual C 5.0.

Instruction Cache and Branch Prediction Effects

Try different function ordering to see how it affects performance, there are sometimes interesting differences of several FPS (frames per second—as a measure of the performance of graphics applications) based on that. Instruction cache thrashing is one suspect in this. The other one is the branch prediction which has a global history component where branches can influence the prediction of other branches. Most of the time this helps. (Two branches might be closely correlated—if one is taken the other one is always not taken.) But it can also hurt, like all heuristic algorithms.

In order to reduce the potential for instruction cache thrashing, group all the program's hotspots close together. For example extract all the performance-critical functions into a single file.

Use the Linker

There is another way to affect function ordering that may be more desirable. The linker allows the programmer to specify the exact order of every function in a DLL/executable as follows:

1. All source code must be compiled with the /Gy switch. This creates packaged functions—a COMDAT record is emitted into the object file for each function.
2. At link time, use the /ORDER:@filename switch to fix the order of functions in the DLL/executable. The term *filename*, refers to a file that lists all function names in the order to be emitted, one function name per line. For C code it's simply the function name as it appears in the source (no pre-pended underscore, no @xx suffix for Pascal calling convention).
3. This does not work for object files produced by MASM. MASM doesn't have a switch to create packaged functions, and it does not allow the user to create a COMDAT entry manually by putting *COMDAT func* into your source.

To reduce potential problems due to branch prediction, eliminate as many branches as possible. The AMD-K6-III and

AMD Athlon processors have large instruction caches, and aggressive loop unrolling (which increases the code size) helps.

It is also worthwhile to eliminate branches which have small amounts of code, replacing the branches with in-line computation.

Code Alignment

To get 32-byte alignment in MASM 6.13, forgo the convenience of new-style segment declarations, and use something like the following:

```
_TEXT SEGMENT PAGE PUBLIC USE32 'CODE'
   ASSUME  CS:FLAT, DS:FLAT, SS:FLAT, ES:FLAT
   ALIGN 32
_TEXT ENDS
```

MASM may not allow ALIGN to be more restrictive than the SEGMENT alignment. If .CODE is used, the result is a PARA aligned segment—a 16-byte aligned segment.

For inline assembly in Microsoft® Visual C, the best alignment is 16-byte alignment by using *align 16* in the inline assembly code. Microsoft Visual C 5.0 without SP3 ignores this directive, so check whether the alignment is actually there. Microsoft Visual C 4.2 seems to work in this regard. At present, correct operation of *align* under Microsoft Visual C 5.0 with SP3 has not been verified.

For inline assembly in Metrowerks™ CodeWarrior Pro 4, *align 32* is accepted and works. See the specific vendor for more information.

Software Write Combining

The writes-to-non-cacheable space is an important issue for low-level drivers. Processors communicate with graphics chips through a command buffer on the graphics card which is mapped to non-cacheable PCI (or AGP) space. On a Pentium® II, this can be made high-performance by setting up that space as

UCWC (non-cacheable write-combining), in which case the Pentium II does write-combining and even bursting to that space.

AMD-K6-2 processors that predate the CXT core (CPUID less than 588h) do not support a UCWC memory type, and they neither perform write combining, nor do they burst to UC memory. AMD-K6-2 processors with the CXT core (CPUID 588h to 58Fh) and AMD-K6-III processors support write combining to non-cacheable space, but they are not able to use burst transfers when writing to non-cacheable memory areas.

Also, AMD-K6-2 processors predating the CXT core do not pipeline writes to non-cacheable space well. This can create a bottleneck when a lot of data needs to be transferred to the graphics card, which for 3D graphics drivers happens predominantly in texture download and triangle download code. (These two can cover about 99% of all writes.) Therefore, for good performance with the millions of existing AMD-K6-2 processors and even for AMD-K6-2 processors with the CXT core and AMD-K6-III processors, software needs to organize the PCI or AGP writes carefully to achieve around a 20% performance gain in the process.

This technique is called software write combining. The basic technique is to collect all writes to non-cacheable space into aligned QWORDs as much as possible. This is accomplished by using an MMX register as a write buffer and collecting DWORD writes using PUNPCK. Then store data out using aligned MOVQ stores. The following two basic approaches can align the QWORD writes:

1. If there is a NOP command consisting of a single DWORD, which takes no processing time in the graphics chip, issue the NOP command if the buffer pointer is not QWORD aligned, then continue writing out QWORDs. This works if the DWORDs in the command buffer are at least DWORD aligned. It has the drawback of wasting some bandwidth for the NOP commands.
2. We can split the code into two code streams. If the buffer pointer is not QWORD aligned, take path one and write the first chunk as a DWORD, then continue writing QWORD. If the buffer pointer is aligned, take path two and start writing out QWORDs immediately.

In both cases there is the end case where we need to flush the write buffer (MMX register) at the end of the write loop.

Option 2 is recommended for highest possible performance, but option 1 is often easier to implement and often provides similar performance.

For AMD-K6-2 processors with the CXT core and for AMD-K6-III processors, use both software write combining *and* enable the hardware write-combining features of these processors.

Addressing Modes on the AMD-K6®-2 and AMD-K6®-III Processors

The addressing modes listed below are sub-optimal for all instructions. They degrade short-decoded instructions to vector decode (degrade to long-decode in the case of 3DNow! instructions). This is due to the lack of on-the-fly corrections to the instruction length that is computed during predecode.

- 16-bit addressing: [SI], [SI+disp8], [SI+disp16], [DI]
- 32-bit addressing: [ESI]

The following addressing modes are sub-optimal for all instructions with 0Fh prefix (including all MMX/3DNow! instructions). Again, it degrades short-decoded instructions to vector (long decode in the case of the 3DNow! instruction set). This is due to the inability to determine the instruction length from the first three bytes (0F-prefix, opcode, ModR/M). Note: This category has been eliminated in AMD-K6-2 processors with the CXT core and AMD-K6-III processor. However millions of existing AMD-K6-2 processors are affected by this issue, so it is highly recommended to avoid these addressing modes.

1. ModR/M = 00_xxx_100b is the only ModR/M encoding that requires the SIB value to determine instruction length. For this ModR/M, the processor doesn't know whether there is a disp32 or not until it looks at the SIB (which predecode cannot do in the case of MMX/3DNow!). For ModR/M = 01_xxx_100b there is always a disp8, and for ModR/M =

10_xxx_100b there is always a disp32, so the length can be determined from looking at ModR/M without looking at the SIB byte.

2. This ModR/M encoding is encountered with the following *source-level* addressing modes:
   - [base+scale×index]
   - [scale×index+disp]
   - [scale×index]
   - [base+index]

The following example demonstrates the ModR/M byte and SIB byte resulting from several addressing modes; note that the MOV instruction is not affected by the issue described here.

```
opc mod sib disp
8B  04  F2                 mov eax, [edx+8*esi]
8B  04  B3                 mov eax, [4*esi+ebx]
8B  04  D5  00000000       mov eax, [8*edx]
8B  04  13                 mov eax, [edx+ebx]
```

Note that the third mode is actually identical to the second as far as the actual encoding is concerned (basically it's encoded as [scale×index+0]).

Also, there is a length restriction. Any instruction longer than seven bytes cannot be short decoded. For MMX instructions, avoid addressing modes with SIB and 32-bit displacement. For 3DNow! instructions, avoid all addressing modes with 32-bit displacement.