3DNow!™

Technology Manual

AMD

Contents

List of Figures

List of Tables

Revision History

Date	Rev	Description
Feb 1998	A	Initial Release
Feb 1998	B	Clarified CPUID usage in "Feature Detection" on page 3.
May 1998	C	Revised description of 3DNow! instructions in "Definitions" on page 8.
May 1998	C	Revised function descriptions in Table 2, "3DNow!™ Floating-Point Instructions," on page 13.
Sept 1998	D	Revised code example for the PFRSQRT instruction on page 48.
Sept 1998	D	Changed exceptions generated for the PREFETCH/PREFETCHW instructions to none, deleted exception table, and revised PREFETCHW description on page 56.
Sept 1998	D	Added PUNPCKLDQ instruction to the division example (24-bit precision) on page 60.

1

3DNow!™ Technology

Introduction

3DNow!™ Technology is a significant innovation to the x86 architecture that drives today's personal computers. 3DNow! technology is a group of new instructions that opens the traditional processing bottlenecks for floating-point-intensive and multimedia applications. With 3DNow! technology, hardware and software applications can implement more powerful solutions to create a more entertaining and productive PC platform. Examples of the type of improvements that 3DNow! technology enables are faster frame rates on high-resolution scenes, much better physical modeling of real-world environments, sharper and more detailed 3D imaging, smoother video playback, and near theater-quality audio.

AMD has taken a leadership role in developing these new instructions that enable exciting new levels of performance and realism. 3DNow! technology was defined and implemented in collaboration with independent software developers, including operating system designers, application developers, and graphics vendors. It is compatible with today's existing x86 software and requires no operating system support, thereby enabling 3DNow! applications to work with all existing operating systems. 3DNow! technology will first appear in the AMD-K6®-2 processor and AMD-K6 processor Model 9.

Key Functionality

The 3DNow! technology instructions are intended to open a major processing bottleneck in a 3D graphics application—floating-point operations. Today's 3D applications are facing limitations due to the fact that only one floating-point execution unit exists in the most advanced x86 processors. The front end of a typical 3D graphics software pipeline performs object physics, geometry transformations, clipping, and lighting calculations. These computations are very floating-point intensive and often limit the features and functionality of a 3D application. The source of performance for the 3DNow! instructions originates from the single instruction multiple data (SIMD) implementation. With SIMD, each instruction not only operates on two single-precision, floating-point operands, but the microarchitecture within the AMD-K6-2 processor can execute up to two 3DNow! instructions per clock through two register execution pipelines, which allows for a total of four floating-point operations per clock. In addition, because the 3DNow! instructions use the same floating-point registers as the MMX™ technology instructions, task switching between MMX and 3DNow! operations is eliminated.

The 3DNow! technology instruction set contains 21 instructions that support SIMD floating-point operations and includes SIMD integer operations, data prefetching, and faster MMX-to-floating-point switching. To improve MPEG decoding, the 3DNow! instructions include a specific SIMD integer instruction created to facilitate pixel-motion compensation. Because media-based software typically operates on large data sets, the processor often needs to wait for this data to be transferred from main memory. The extra time involved with retrieving this data can be avoided by using the new 3DNow! instruction called PREFETCH. This instruction can ensure that data is in the level 1 cache when it is needed. To improve the time it takes to switch between MMX and x87 code, the 3DNow! instructions include the FEMMS (fast entry/exit multimedia state) instruction, which eliminates much of the overhead involved with the switch. The addition of 3DNow! technology expands the capabilities of the AMD-K6 family of processors and enables a new generation of enriched user applications.

Feature Detection

To properly identify and use the 3DNow! instructions, the application program must determine if the processor supports them. The CPUID instruction gives programmers the ability to determine the presence of 3DNow! technology on a processor. Software applications must first test to see if the CPUID instruction is supported. For a detailed description of the CPUID instruction, see the *AMD Processor Recognition Application Note,* order# 20734.

The presence of the CPUID instruction is indicated by the ID bit (21) in the EFLAGS register. If this bit is writable, the CPUID instruction is supported. The following code sample shows how to test for the presence of the CPUID instruction.

```
pushfd                  ; save EFLAGS
pop   eax               ; store EFLAGS in EAX
mov   ebx, eax          ; save in EBX for later testing
xor   eax, 00200000h    ; toggle bit 21
push  eax               ; put to stack
popfd                   ; save changed EAX to EFLAGS
pushfd                  ; push EFLAGS to TOS
pop   eax               ; store EFLAGS in EAX
cmp   eax, ebx          ; see if bit 21 has changed
jz    NO_CPUID          ; if no change, no CPUID
```

Once the software has identified the processor's support for CPUID, it must test for extended functions by executing extended function 0 (EAX=8000_000h). The EAX register returns the largest extended function input value defined for the CPUID instruction on the processor. If the value is greater than 8000_0000h, extended functions are supported.

The next step is for the programmer to determine if the 3DNow! instructions are supported. Extended function 8000_0001h of the CPUID instruction provides this information by returning the extended feature bits in the EDX register. If bit 31 in the EDX register is set to 1, 3DNow! instructions are supported. The following code sample shows how to test for 3DNow! instruction support.

```
mov   eax, 8000_0001h   ; setup extended function 1
CPUID                   ; call the function
test  edx, 8000_0000h   ; test bit 31
jnz   YES_3DNow!        ; 3DNow! technology supported
```

Register Set

The complete multimedia units in the AMD-K6-2 processor combine the existing MMX instructions with the new 3DNow! instructions. In addition, by merging 3DNow! with MMX, it becomes possible to write x86 programs containing both integer, MMX, and floating-point graphics instructions with no performance penalty for switching between the multimedia (integer) and 3DNow! (floating-point) units.

The AMD-K6-2 processor implements eight 64-bit 3DNow!/MMX registers. These registers are mapped onto the floating-point registers. As shown in Figure 1, the 3DNow! and MMX instructions refer to these registers as mm0 to mm7. Mapping the new 3DNow!/MMX registers onto the floating-point register stack enables backwards compatibility for the register saving that must occur as a result of task switching.

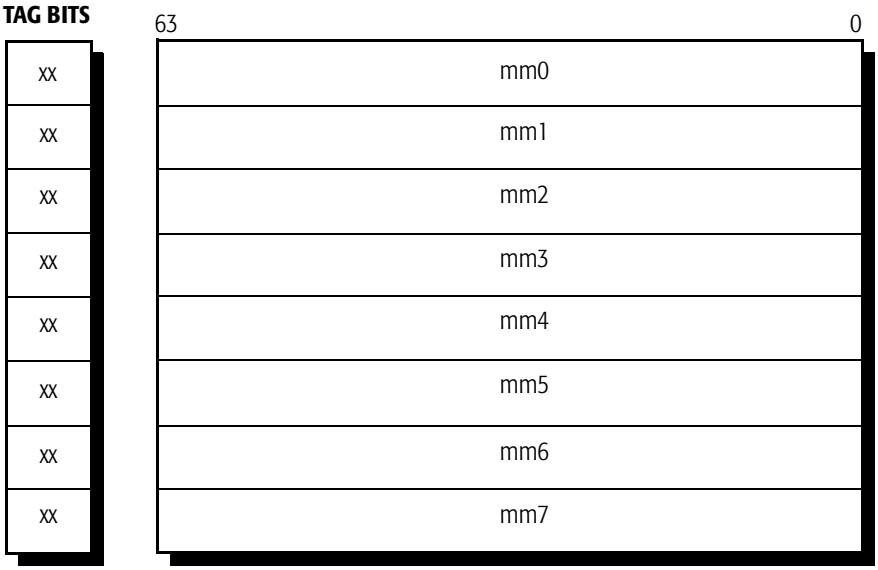


Figure 1. 3DNow!™/MMX™ Registers

Aliasing the 3DNow!/MMX registers onto the floating-point register stack provides a safe method to introduce 3DNow! and MMX technology, because it does not require modifications to existing operating systems. Instead of requiring operating

system modifications, new 3DNow! and MMX technology applications are supported through device drivers, 3DNow! and MMX libraries, or Dynamic Link Library (DLL) files.

Current operating systems have support for floating-point operations and the floating-point register state. Using the floating-point registers for 3DNow! and MMX code is a convenient way of implementing non-intrusive support for 3DNow! and MMX instructions. Every time the processor executes a 3DNow! or MMX instruction, all the floating-point register tag bits are set to zero (00b=valid), except for the FEMMS and EMMS instructions, which set all tag bits to one (11b=empty).

Note: *Executing the PREFETCH instruction does not change the tag bits.*

Data Types

3DNow! technology uses a packed data format. The data is packed in a single, 64-bit 3DNow!/MMX register or a quadword memory operand.

Figure 2 shows the 3DNow! floating-point data type. D0 and D1 each hold an IEEE 32-bit single-precision, floating-point doubleword.

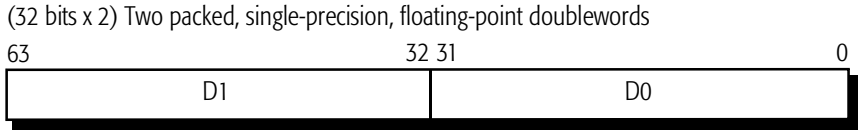
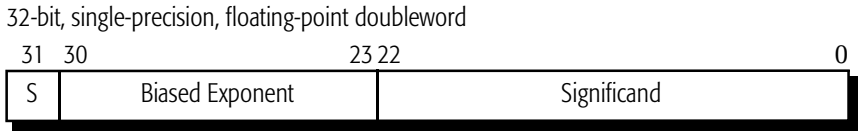


Figure 2. 3DNow!™ Data Type

Figure 3 on page 6 shows the format of the IEEE 32-bit, single-precision, floating-point format.



Value definitions

$1.\ X=(-1)^{S}*0$	Biased Exponent=0
$2.\ X=(-1)^{S}*2^{(\text{Biased Exponent} - 127)}*\text{Significand}$	0<Biased Exponent<FFh
3. X=Undefined	Biased Exponent=FFh

X is the value of the 32-bit, single-precision, floating-point doubleword.

Figure 3. Single-Precision, Floating-Point Data Format

Figure 4 on page 6 shows the formats for the integer data types.

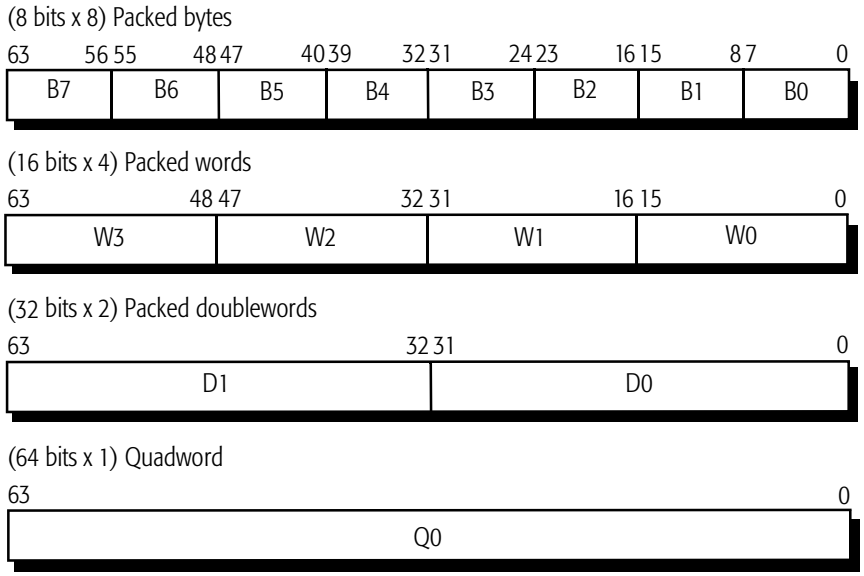


Figure 4. Integer Data Types

3DNow!™ Instruction Formats

The format of 3DNow! instruction encodings is based on the conventional x86 modR/M instruction format and is similar to the format used by MMX instructions. The assembly language syntax used for the 3DNow! instructions is as follows:

3DNow! Mnemonic mmreg1, mmreg2/mem64

The destination and source1 operand (mmreg1) must be an MMX register (mm0–mm7). The source2 operand (mmreg2/mem64) can be either an MMX register or a 64-bit memory value.

The encoding uses the opcode prefix 0Fh followed by a second opcode byte of 0Fh. To differentiate the various 3DNow! instructions, a third instruction suffix byte is used. This suffix byte occupies the same position at the end of a 3DNow! instructions as would an imm8 byte. The opcode format is as follows:

0Fh 0Fh modR/M [sib] [displacement] *3DNow!_suffix*

The specific operands (mmreg1 and mmreg2/mem64) determine the values used in modR/M [sib] [displacement], and follow conventional x86 encodings. The 3DNow! suffix is determined by the actual 3DNow! instruction. The 3DNow! suffixes are defined in Table 2 on page 13.

As an example, the 3DNow! PFMUL instruction can produce the following opcodes, depending on its use:

Opcode	Instruction
0F 0F CA B4	PFMUL mm1, mm2
0F 0F 0B B4	PFMUL mm1, [ebx]
0F 0F 4B 0A B4	PFMUL mm1, [ebx+10]
26 0F 0F 0B B4	PFMUL mm1, es:[ebx]
0F 0F 4C 83 0A B4	PFMUL mm1, [ebx+eax*4+10]

The encoding of the two performance-enhancement instructions (FEMMS and PREFETCH) uses a single opcode prefix 0Fh. The details of the opcodes for these instructions are shown on pages 18 and 56 respectively.

Definitions

3DNow! technology provides 21 additional instructions to support high-performance, 3D graphics and audio processing. 3DNow! instructions are vector instructions that operate on 64-bit registers. 3DNow! instructions are SIMD—each instruction operates on pairs of 32-bit values.

The definitions for the 3DNow! instructions starting on page 17 contain designations classifying each instruction as vectored or scalar. Vector instructions operate in parallel on two sets of 32-bit, single-precision, floating-point words. Instructions that are labeled as scalar instructions operate on a single set of 32-bit operands (from the low halves of the two 64-bit operands).

The 3DNow! single-precision, floating-point format is compatible with the IEEE-754, single-precision format. This format comprises a 1-bit sign, an 8-bit biased exponent, and a 23-bit significand with one hidden integer bit for a total of 24 bits in the significand. The bias of the exponent is 127, consistent with the IEEE single-precision standard. The significands are normalized to be within the range of [1,2).

In contrast to the IEEE standard that dictates four rounding modes, 3DNow! technology supports one rounding mode — either round-to-nearest or round-to-zero (truncation). The hardware implementation of 3DNow! technology determines the rounding mode. The AMD-K6-2 processor implements round-to-nearest mode. Regardless of the rounding mode used, the floating-point-to-integer and integer-to-floating-point conversion instructions, PF2ID and PI2FD, always use the round-to-zero (truncation) mode.

The largest, representable, normal number in magnitude for this precision in hexadecimal has an exponent of FEh and a significand of 7FFFFFh, with a numerical value of $2^{127}(2 - 2^{-23})$. All results that overflow above the maximum-representable positive value are saturated to either this maximum-representable normal number or to positive infinity. Similarly, all results that overflow below the minimum-representable negative value are saturated to either this minimum-representable normal number or to negative infinity.

The implementation of 3DNow! technology determines how arithmetic overflow is handled—either properly signed maximum- or minimum-representable normal numbers or properly signed infinities. The AMD-K6-2 processor generates properly signed maximum- or minimum-representable normal numbers.

Infinities and NaNs are not supported as operands to 3DNow! instructions.

The smallest representable normal number in magnitude for this precision in hexadecimal has an exponent of 01h and a significand of 000000h, with a numerical value of 2^{-126}. Accordingly, all results below this minimum representable value in magnitude are held to zero. Table 1 shows the exponent ranges supported by the 3DNow! technology.

Table 1. 3DNow!™ Technology Exponent Ranges

Biased Exponent	Description
FFh	Unsupported *
00h	Zero
00h<x<FFh	Normal
01h	$2^{(1-127)}$ lowest possible exponent
FEh	$2^{(254-127)}$ largest possible exponent

Note:

* *Unsupported numbers can be used as operands. The results of operations with unsupported numbers are undefined.*

Like MMX instructions, 3DNow! instructions do not generate numeric exceptions nor do they set any status flags. It is the user's responsibility to ensure that in-range data is provided to 3DNow! instructions and that all computations remain within valid ranges (or are held as expected).

Execution Resources

The register operations of all 3DNow! floating-point instructions are executed by either the register X unit or the register Y unit. One operation can be issued to each register unit each clock cycle, for a maximum issue and execution rate of two 3DNow! operations per cycle. All 3DNow! operations

have an execution latency of two clock cycles and are fully pipelined.

Even though 3DNow! execution resources are not duplicated in both register units (for example, there are not two pairs of 3DNow! multipliers, just one shared pair of multipliers), there are no instruction-decode or operation-issue pairing restrictions. When, for example, a 3DNow! multiply operation starts execution in a register unit, that unit grabs and uses the one shared pair of 3DNow! multipliers. Only when actual contention occurs between two 3DNow! operations starting execution at the same time is one of the operations held up for one cycle in its first execution pipe stage while the other proceeds. The delay is never more than one cycle.

For code optimization purposes, 3DNow! operations are grouped into two categories. These categories are based on execution resources and are important when creating properly scheduled code. As long as two 3DNow! operations that start execution simultaneously do not fall into the same category, both operations will start execution without delay.

The first category of instructions contains the operations for the following 3DNow! instructions: PFADD, PFSUB, PFSUBR, PFACC, PFCMPx, PFMIN, PFMAX, PI2FD, PF2ID, PFRCP, and PFRSQRT.

The second category contains the operations for the following 3DNow! instructions: PFMUL, PFRCPIT1, PFRSQIT1, and PFRCPIT2.

Note: *3DNow! add and multiply operations, among other combinations, can execute simultaneously.*

Normally, in high-performance 3DNow! code, all of the 3DNow! instructions are properly scheduled apart from each other so as to avoid delays due to execution resource contentions (as well as taking into account dependencies and execution latencies). For further information regarding code optimization, see the *AMD-K6®-2 Processor Code Optimization Application Note*, order# 21924. This document provides in-depth discussions of code optimization techniques for the AMD-K6-2 processor.

The SIMD 3DNow! instructions are summarized in Table 2 on page 13. The dedicated and shared execution resources of the

register X unit and register Y unit are shown in Figure 5 on page 12. The execution resources for some MMX operations, as well as all 3DNow! operations, are shared between the two register units. For contention-checking purposes, each box represents a category of operations that cannot start execution simultaneously. In addition, the MMX and 3DNow! multiplies use the same hardware, while MMX and 3DNow! adds and subtracts do not.

The two 3DNow! performance-enhancement instructions are summarized in Table 3 on page 13. The FEMMS instruction does not use any specific execution resource or pipeline. The PREFETCH instruction is operated on in the Load unit.

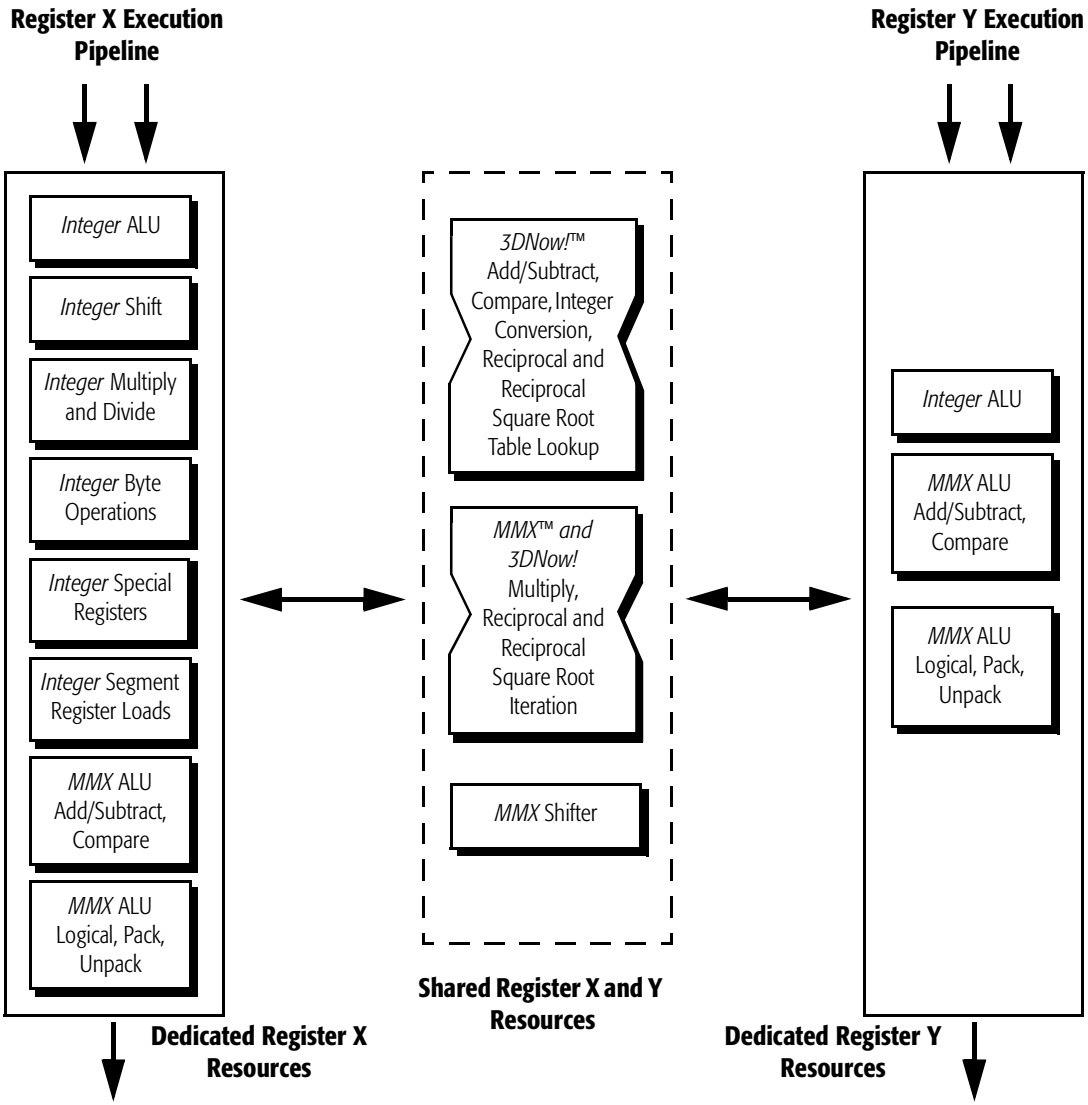


Figure 5. Register X Unit and Register Y Unit Resources

Table 2. 3DNow!™ Floating-Point Instructions

Operation	Function	Opcode Suffix
PAVGUSB	Packed 8-bit Unsigned Integer Averaging	BFh
PFADD	Packed Floating-Point Addition	9Eh
PFSUB	Packed Floating-Point Subtraction	9Ah
PFSUBR	Packed Floating-Point Reverse Subtraction	AAh
PFACC	Packed Floating-Point Accumulate	AEh
PFCMPGE	Packed Floating-Point Comparison, Greater or Equal	90h
PFCMPGT	Packed Floating-Point Comparison, Greater	A0h
PFCMPEQ	Packed Floating-Point Comparison, Equal	B0h
PFMIN	Packed Floating-Point Minimum	94h
PFMAX	Packed Floating-Point Maximum	A4h
PI2FD	Packed 32-bit Integer to Floating-Point Conversion	0Dh
PF2ID	Packed Floating-Point to 32-bit Integer	1Dh
PFRCP	Packed Floating-Point Reciprocal Approximation	96h
PFRSQRT	Packed Floating-Point Reciprocal Square Root Approximation	97h
PFMUL	Packed Floating-Point Multiplication	B4h
PFRCPIT1	Packed Floating-Point Reciprocal First Iteration Step	A6h
PFRSQIT1	Packed Floating-Point Reciprocal Square Root First Iteration Step	A7h
PFRCPIT2	Packed Floating-Point Reciprocal/Reciprocal Square Root Second Iteration Step	B6h
PMULHRW	Packed 16-bit Integer Multiply with rounding	B7h

Table 3. 3DNow!™ Performance-Enhancement Instructions

Operation	Function	Opcode Suffix
FEMMS	Faster entry/exit of the MMX™ or floating-point state	0Eh
PREFETCH	Prefetch at least a 32-byte line into L1 data cache (Dcache)	0Dh

Task Switching

With respect to task switching, treat the 3DNow! instructions exactly the same as MMX instructions. Operating system design must be taken into account when writing a 3DNow! program.

The programmer must know whether the operating system automatically saves the current states when task switching, or if the 3DNow! program has to provide the code to save states.

If a task switch occurs, the Control Register (CR0) Task Switch (TS) bit is set to 1. The processor then generates an interrupt 7 (int 7—Device Not Available) when it encounters the next floating-point, 3DNow!, or MMX instruction, allowing the operating system to save the state of the 3DNow!/MMX/FP registers.

In a multitasking operating system, if there is a task switch when 3DNow!/MMX applications are running with older applications that do not include MMX instructions, the MMX/FP register state is still saved automatically through the int 7 handler.

Exceptions

Table 4 contains a list of exceptions that 3DNow! and MMX instructions can generate.

Table 4. 3DNow!™ and MMX™ Instruction Exceptions

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)	X	X	X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The rules for exceptions are the same for both MMX and 3DNow! instructions. In addition, exception detection and handling is identical for MMX and 3DNow! instructions. None of the exception handlers need modification.

Notes:

1. *An invalid opcode exception (interrupt 6) occurs if a 3DNow! instruction is executed on a processor that does not support 3DNow! instructions.*
2. *If a floating-point exception is pending and the processor encounters a 3DNow! instruction, FERR# is asserted and, if CR0.NE = 1, an interrupt 16 is generated. (This is the same for MMX instructions.)*

Prefixes

The following prefixes can be used with 3DNow! instructions:

- The segment override prefixes (2Eh/CS, 36h/SS, 3Eh/DS, 26h/ES, 64h/FS, and 65h/GS) affect 3DNow! instructions that contain a memory operand.
- The address-size override prefix (67h) affects 3DNow! instructions that contain a memory operand.
- The operand-size override prefix (66h) is ignored.
- The LOCK prefix (F0h) triggers an invalid opcode exception (interrupt 6).
- The REP prefixes (F3h/ REP/ REPE/ REPZ, F2h/ REPNE/ REPNZ) are ignored.

2

3DNow!™ Instruction Set

The following 3DNow! instruction definitions are in alphabetical order according to the instruction mnemonics.

FEMMS

mnemonic	*opcode*	*description*
FEMMS	0F 0Eh	Faster Enter/Exit of the MMX or floating-point state

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.

Like the EMMS instruction, the FEMMS instruction can be used to clear the MMX state following the execution of a block of MMX instructions. Because the MMX registers and tag words are shared with the floating-point unit, it is necessary to clear the state before executing floating-point instructions. Unlike the EMMS instruction, the contents of the MMX/floating-point registers are undefined after a FEMMS instruction is executed. Therefore, the FEMMS instruction offers a faster context switch at the end of an MMX routine where the values in the MMX registers are no longer required. FEMMS can also be used prior to executing MMX instructions where the preceding floating-point register values are no longer required, which facilitates faster context switching.

PAVGUSB

mnemonic	*opcode/suffix*	*description*
PAVGUSB mmreg1, mmreg2/mem64	0F 0Fh / BFh	Average of unsigned packed 8-bit values

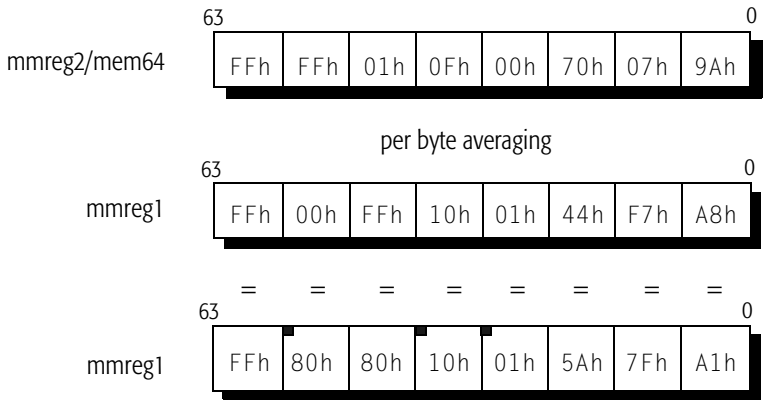
Privilege: None
Registers Affected: MMX
Flags Affected: None
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PAVGUSB instruction produces the rounded averages of the eight unsigned 8-bit integer values in the source operand (an MMX register or a 64-bit memory location) and the eight corresponding unsigned 8-bit integer values in the destination operand (an MMX register). It does so by adding the source and destination byte values and then adding a 001h to the 9-bit intermediate value. The intermediate value is then divided by 2 (shifted right one place) and the eight unsigned 8-bit results are stored in the MMX register specified as the destination operand.

The PAVGUSB instruction can be used for pixel averaging in MPEG-2 motion compensation and video scaling operations.

Functional Illustration of the PAVGUSB Instruction



■ Indicates a value that was rounded-up

The following list explains the functional illustration of the PAVGUSB instruction:

- The rounded byte average of FFh and FFh is FFh.
- The rounded byte average of FFh and 00h is 80h.
- The rounded byte average of 01h and FFh is also 80h.
- The rounded byte average of 0Fh and 10h is 10h.
- The rounded byte average of 00h and 01h is 01h.
- The rounded byte average of 70h and 44h is 5Ah.
- The rounded byte average of 07h and F7h is 7Fh.
- The rounded byte average of 9Ah and A8h is A1h.

The equations for byte averaging with rounding are as follows:

- mmreg1[63:56] = (mmreg1[63:56] + mmreg2/mem64[63:56] + 01h)/2
- mmreg1[55:48] = (mmreg1[55:48] + mmreg2/mem64[55:48] + 01h)/2
- mmreg1[47:40] = (mmreg1[47:40] + mmreg2/mem64[47:40] + 01h)/2
- mmreg1[39:32] = (mmreg1[39:32] + mmreg2/mem64[39:32] + 01h)/2
- mmreg1[31:24] = (mmreg1[31:24] + mmreg2/mem64[31:24] + 01h)/2
- mmreg1[23:16] = (mmreg1[23:16] + mmreg2/mem64[23:16] + 01h)/2
- mmreg1[15:8] = (mmreg1[15:8] + mmreg2/mem64[15:8] + 01h)/2
- mmreg1[7:0] = (mmreg1[7:0] + mmreg2/mem64[7:0] + 01h)/2

PF2ID

mnemonic	*opcode/imm8*	*description*
PF2ID mmreg1, mmreg2/mem64	0Fh 0Fh / 1Dh	Converts packed floating-point operand to packed 32-bit integer

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PF2ID is a vector instruction that converts a vector register containing single-precision, floating-point operands to 32-bit signed integers using truncation. Table 5 on page 22 shows the numerical range of the PF2ID instruction.

The PF2ID instruction performs the following operations:

```
IF (mmreg2/mem64[31:0] >= 2^31)
   THEN mmreg1[31:0] = 7FFF_FFFFh
ELSEIF (mmreg2/mem64[31:0] <= -2^31)
   THEN mmreg1[31:0] = 8000_0000h
ELSE mmreg1[31:0] = int(mmreg2/mem64[31:0])
IF (mmreg2/mem64[63:32] >= 2^31)
   THEN mmreg1[63:32] = 7FFF_FFFFh
ELSEIF (mmreg2/mem64[63:32] <= -2^31)
   THEN mmreg1[63:32] = 8000_0000h
ELSE mmreg1[63:32] = int(mmreg2/mem64[63:32])
```

Table 5. Numerical Range for the PF2ID Instruction

Source 2	Source 1 and Destination
0	0
Normal, abs(Source 1) <1	0
Normal, –2147483648 < Source 1 <= –1	round to zero (Source 1)
Normal, 1 <= Source 1< 2147483648	round to zero (Source 1)
Normal, Source 1 >= 2147483648	7FFF_FFFFh
Normal, Source 1 <= –2147483648	8000_0000h
Unsupported	Undefined

Related Instructions See the PI2FD instruction.

PFACC

mnemonic	*opcode/imm8*	*description*
PFACC mmreg1, mmreg2/mem64	0Fh 0Fh / AEh	Floating-point accumulate

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFACC is a vector instruction that accumulates the two words of the destination operand and the source operand and stores the results in the low and high words of destination operand respectively. Both operands are single-precision, floating-point operands with 24-bit significands. Table 6 on page 24 shows the numerical range of the PFACC instruction.

The PFACC instruction performs the following operations:

```
mmreg1[31:0] = mmreg1[31:0] + mmreg1[63:32]
mmreg1[63:32] = mmreg2/mem64[31:0] + mmreg2/mem64[63:32]
```

Table 6. Numerical Range for the PFACC Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	+/– 0 *	Source 2	Source 2
	Normal	Source 1	Normal, +/– 0 **	Undefined
	Unsupported	Source 1	Undefined	Undefined

Notes:

* *The sign of the result is the logical AND of the signs of the source operands.*

** *If the absolute value of the result is less then 2^{-126}, the result is zero with the sign being the sign of the source operand that is larger in magnitude (if the magnitudes are equal, the sign of source 1 is used). If the absolute value of the result is greater than or equal to 2^{128}, the result is the largest normal number with the sign being the sign of the source operand that is larger in magnitude.*

PFADD

mnemonic	*opcode/imm8*	*description*
PFADD mmreg1, mmreg2/mem64	0Fh 0Fh / 9Eh	Packed, floating-point addition

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFADD is a vector instruction that performs addition of the destination operand and the source operand. Both operands are single-precision, floating-point operands with 24-bit significands. Table 7 on page 26 shows the numerical range of the PFADD instruction.

The PFADD instruction performs the following operations:

```
mmreg1[31:0] = mmreg1[31:0] + mmreg2/mem64[31:0]
mmreg1[63:32] = mmreg1[63:32] + mmreg2/mem64[63:32]
```

Table 7. Numerical Range for the PFADD Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	+/– 0 *	Source 2	Source 2
	Normal	Source 1	Normal, +/– 0 **	Undefined
	Unsupported	Source 1	Undefined	Undefined

Notes:

* *The sign of the result is the logical AND of the signs of the source operands.*

** *If the absolute value of the result is less then 2^{-126}, the result is zero with the sign being the sign of the source operand that is larger in magnitude (if the magnitudes are equal, the sign of source 1 is used). If the absolute value of the result is greater than or equal to 2^{128}, the result is the largest normal number with the sign being the sign of the source operand that is larger in magnitude.*

PFCMPEQ

mnemonic	*opcode/imm8*	*description*
PFCMPEQ mmreg1, mmreg2/mem64	0Fh 0Fh / B0h	Packed floating-point comparison, equal to

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFCMPEQ is a vector instruction that performs a comparison of the destination operand and the source operand and generates all one bits or all zero bits based on the result of the corresponding comparison. Table 8 on page 28 shows the numerical range of the PFCMPEQ instruction.

The PFCMPEQ instruction performs the following operations:

```
IF (mmreg1[31:0] = mmreg2/mem64[31:0])
   THEN mmreg1[31:0] = FFFF_FFFFh
ELSE mmreg1[31:0] = 0000_0000h
IF (mmreg1[63:32] = mmreg2/mem64[63:32]
   THEN mmreg1[63:32] = FFFF_FFFFh
ELSE mmreg1[63:32] = 0000_0000h
```

Table 8. Numerical Range for the PFCMPEQ Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	FFFF_FFFFh *	0000_0000h	0000_0000h
	Normal	0000_0000h	0000_0000h, FFFF_FFFFh **	0000_0000h
	Unsupported	0000_0000h	0000_0000h	Undefined

Notes:

* *Positive zero is equal to negative zero.*

** *The result is FFFF_FFFFh if source 1 and source 2 have identical signs, exponents, and mantissas. Otherwise, the result is 0000_0000h.*

Related Instructions See the PFCMPGE instruction.

See the PFCMPGT instruction.

PFCMPGE

mnemonic	*opcode/imm8*	*description*
PFCMPGE mmreg1, mmreg2/mem64	0Fh 0Fh / 90h	Packed floating-point comparison, greater than or equal to

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFCMPGE is a vector instruction that performs a comparison of the destination operand and the source operand and generates all one bits or all zero bits based on the result of the corresponding comparison. Table 9 on page 30 shows the numerical range of the PFCMPGE instruction.

The PFCMPGE instruction performs the following operations:

```
IF (mmreg1[31:0] >= mmreg2/mem64[31:0])
   THEN mmreg1[31:0] = FFFF_FFFFh
ELSE mmreg1[31:0] = 0000_0000h
IF (mmreg1[63:32] >= mmreg2/mem64[63:32]
   THEN mmreg1[63:32] = FFFF_FFFFh
ELSE mmreg1[63:32] = 0000_0000h
```

Table 9. Numerical Range for the PFCMPGE Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	FFFF_FFFFh *	0000_0000h, FFFF_FFFFh **	Undefined
	Normal	0000_0000h, FFFF_FFFF ***	0000_0000h, FFFF_FFFFh ****	Undefined
	Unsupported	Undefined	Undefined	Undefined

Notes:

* *Positive zero is equal to negative zero.*

** *The result is FFFF_FFFFh, if source 2 is negative. Otherwise, the result is 0000_0000h.*

*** *The result is FFFF_FFFFh, if source 1 is positive. Otherwise, the result is 0000_0000h.*

**** *The result is FFFF_FFFFh, if source 1 is positive and source 2 is negative, or if they are both negative and source 1 is smaller than or equal in magnitude to source 2, or if source 1 and source 2 are both positive and source 1 is greater than or equal in magnitude to source 2. The result is 0000_0000h in all other cases.*

Related Instructions See the PFCMPEQ instruction.

See the PFCMPGT instruction.

PFCMPGT

mnemonic	*opcode/imm8*	*description*
PFCMPGT mmreg1, mmreg2/mem64	0Fh 0Fh / A0h	Packed floating-point comparison, greater than

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFCMPGT is a vector instruction that performs a comparison of the destination operand and the source operand and generates all one bits or all zero bits based on the result of the corresponding comparison. Table 10 on page 32 shows the numerical range of the PFCMPGT instruction.

The PFCMPGT instruction performs the following operations:

```
IF (mmreg1[31:0] > mmreg2/mem64[31:0])
   THEN mmreg1[31:0] = FFFF_FFFFh
ELSE mmreg1[31:0] = 0000_0000h
IF (mmreg1[63:32] > mmreg2/mem64[63:32]
   THEN mmreg1[63:32] = FFFF_FFFFh
ELSE mmreg1[63:32] = 0000_0000h
```

Table 10. Numerical Range for the PFCMPGT Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	0000_0000h	0000_0000h, FFFF_FFFFh *	Undefined
	Normal	0000_0000h, FFFF_FFFF **	0000_0000h, FFFF_FFFFh ***	Undefined
	Unsupported	Undefined	Undefined	Undefined

Notes:

* *The result is FFFF_FFFFh, if source 2 is negative. Otherwise, the result is 0000_0000h.*

** *The result is FFFF_FFFFh, if source 1 is positive. Otherwise, the result is 0000_0000h.*

*** *The result is FFFF_FFFFh, if source 1 is positive and source 2 is negative, or if they are both negative and source 1 is smaller in magnitude than source 2, or if source 1 and source 2 are positive and source 1 is greater in magnitude than source 2. The result is 0000_0000h in all other cases.*

Related Instructions See the PFCMPEQ instruction.

See the PFCMPGE instruction.

PFMAX

mnemonic	*opcode/imm8*	*description*
PFMAX mmreg1, mmreg2/mem64	0Fh 0Fh / A4h	Packed floating-point maximum

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFMAX is a vector instruction that returns the larger of the two single-precision, floating-point operands. Any operation with a zero and a negative number returns positive zero. An operation consisting of two zeros returns positive zero. Table 11 on page 34 shows the numerical range of the PFMAX instruction.

The PFMAX instruction performs the following operations:

```
IF (mmreg1[31:0] > mmreg2/mem64[31:0])
   THEN mmreg1[31:0] = mmreg1[31:0]
ELSE mmreg1[31:0] = mmreg2/mem64[31:0]
IF (mmreg1[63:32] > mmreg2/mem64[63:32])
   THEN mmreg1[63:32] = mmreg1[63:32]
ELSE mmreg1[63:32] = mmreg2/mem64[63:32]
```

Table 11. Numerical Range for the PFMAX Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	+0	Source 2, +0 *	Undefined
	Normal	Source 1, +0 **	Source 1/Source 2 ***	Undefined
	Unsupported	Undefined	Undefined	Undefined

Notes:

* *The result is source 2, if source 2 is positive. Otherwise, the result is positive zero.*

** *The result is source 1, if source 1 is positive. Otherwise, the result is positive zero.*

*** *The result is source 1, if source 1 is positive and source 2 is negative. The result is source 1, if both are positive and source 1 is greater in magnitude than source 2. The result is source 1, if both are negative and source 1 is lesser in magnitude than source 2. The result is source 2 in all other cases.*

Related Instructions See the PFMIN instruction.

PFMIN

mnemonic	*opcode/imm8*	*description*
PFMIN mmreg1, mmreg2/mem64	0Fh 0Fh / 94h	Packed floating-point minimum

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFMIN is a vector instruction that returns the smaller of the two single-precision, floating-point operands. Any operation with a zero and a positive number returns positive zero. An operation consisting of two zeros returns positive zero. Table 12 on page 36 shows the numerical range of the PFMIN instruction.

The PFMIN instruction performs the following operations:

```
IF (mmreg1[31:0] < mmreg2/mem64[31:0])
   THEN mmreg1[31:0] = mmreg1[31:0]
ELSE mmreg1[31:0] = mmreg2/mem64[31:0]
IF (mmreg1[63:32] < mmreg2/mem64[63:32])
   THEN mmreg1[63:32] = mmreg1[63:32]
ELSE mmreg1[63:32] = mmreg2/mem64[63:32]
```

Table 12. Numerical Range for the PFMIN Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	+0	Source 2, +0 *	Undefined
	Normal	Source 1, +0 **	Source 1/Source 2 ***	Undefined
	Unsupported	Undefined	Undefined	Undefined

Notes:

* *The result is source 2, if source 2 is negative. Otherwise, the result is positive zero.*

** *The result is source 1, if source 1 is negative. Otherwise, the result is positive zero.*

*** *The result is source 1, if source 1 is negative and source 2 is positive. The result is source 1, if both are negative and source 1 is greater in magnitude than source 2. The result is source 1, if both are positive and source 1 is lesser in magnitude than source 2. The result is source 2 in all other cases.*

Related Instructions See the PFMAX instruction.

PFMUL

mnemonic	*opcode/imm8*	*description*
PFMUL mmreg1, mmreg2/mem64	0Fh 0Fh / B4h	Packed floating-point multiplication

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFMUL is a vector instruction that performs multiplication of the destination operand and the source operand. Both operands are single-precision, floating-point operands with 24-bit significands. Table 13 on page 38 shows the numerical range of the PFMIN instruction.

The PFMUL instruction performs the following operations:

```
mmreg1[31:0] = mmreg1[31:0] * mmreg2/mem64[31:0]
mmreg1[63:32] = mmreg1[63:32] * mmreg2/mem64[63:32]
```

Table 13. Numerical Range for the PFMUL Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	+/– 0 *	+/– 0 *	+/– 0 *
	Normal	+/– 0 *	Normal, +/– 0 **	Undefined
	Unsupported	+/– 0 *	Undefined	Undefined

Notes:

* *The sign of the result is the exclusive-OR of the signs of the source operands.*

** *If the absolute value of the result is less then 2^{-126}, the result is zero with the sign being the exclusive-OR of the signs of the source operands. If the absolute value of the product is greater than or equal to 2^{128}, the result is the largest normal number with the sign being exclusive-OR of the signs of the source operands.*

PFRCP

mnemonic	*opcode/imm8*	*description*
PFRCP mmreg1, mmreg2/mem64	0Fh 0Fh / 96h	Floating-point reciprocal approximation

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFRCP is a scalar instruction that returns a low-precision estimate of the reciprocal of the source operand. The single result value is duplicated in both high and low halves of this instruction's 64-bit result. The source operand is single-precision with a 24-bit significand, and the result is accurate to 14 bits. Table 14 on page 40 shows the numerical range of the PFRCP instruction.

Increased accuracy (the full 24 bits of a single-precision significand) requires the use of two additional instructions (PFRCPIT1 and PFRCPIT2). The first stage of this increase or refinement in accuracy (PFRCPIT1) requires that the input and output of the already executed PFRCP instruction be used as input to the PFRCPIT1 instruction. Refer to "Appendix A" on page 59 for an application-specific example of how to use this instruction and related instructions.

The PFRCP instruction performs the following operations:

```
mmreg1[31:0] = reciprocal(mmreg2/mem64[31:0])
mmreg1[63:32] = reciprocal(mmreg2/mem64[31:0])
```

In the following code example, the bold line illustrates the PFRCP instruction in a sequence used to compute q = a/b accurate to 24 bits:

```
X0 =    PFRCP(b)
X1 =    PFRCPIT1(b,X0)
X2 =    PFRCPIT2(X1,X0)
q  =    PFMUL(a,X2)
```

Table 14. Numerical Range for the PFRCP Instruction

		Source 1 and Destination
Source 2	0	+/– Maximum Normal*
	Normal	Normal, +/– 0 **
	Unsupported	Undefined

Notes:

* *The result has the same sign as the source operand.*

** *If the absolute value of the result is less then 2^{-126}, the result is zero with the sign being the sign of the source operand. Otherwise, the result is a normal with the sign being the same sign as the source operand.*

Related Instructions See the PFRCPIT1 instruction.

See the PFRCPIT2 instruction.

PFRCPIT1

mnemonic	*opcode/imm8*	*description*
PFRCPIT1 mmreg1, mmreg2/mem64	0Fh 0Fh / A6h	Packed floating-point reciprocal, first iteration step

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFRCPIT1 is a vector instruction that performs the first step in a Newton-Raphson iteration to refine the reciprocal approximation produced by the PFRCP instruction (the second and final step yields a result accurate to 24 bits). Table 15 on page 42 shows the numerical range of the PFRCPIT1 instruction.

The behavior of this instruction is only defined for those combinations of operands such that one source operand was the input to the PFRCP instruction and the other source operand was the output of the same PFRCP instruction. Refer to “Appendix A” on page 59 for an application-specific example of how to use this instruction and related instructions.

In the following code example, the bold line illustrates the PFRCPIT1 instruction in a sequence used to compute q = a/b accurate to 24 bits:

```
X0 =    PFRCP(b)
X1 =    PFRCPIT1(b,X0)
X2 =    PFRCPIT2(X1,X0)
q  =    PFMUL(a,X2)
```

Table 15. Numerical Range for the PFRCPIT1 Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	+/– 0 *	+/– 0 *	+/– 0 *
	Normal	+/– 0 *	Normal **	Undefined
	Unsupported	+/– 0 *	Undefined	Undefined

Notes:

* *The sign of the result is the exclusive-OR of the signs of the source operands.*

** *The sign is positive.*

Related Instructions See the PFRCP instruction.

See the PFRCPIT2 instruction.

PFRCPIT2

mnemonic	*opcode/imm8*	*description*
PFRCPIT2 mmreg1, mmreg2/mem64	0Fh 0Fh / B6h	Packed floating-point reciprocal/reciprocal square root, second iteration step

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFRCPIT2 is a vector instruction that performs the second and final step in a Newton-Raphson iteration to refine the reciprocal or reciprocal square root approximation produced by the PFRCP and PFSQRT instructions, respectively. Table 16 on page 44 shows the numerical range of the PFRCPIT2 instruction.

The behavior of this instruction is only defined for those combinations of operands such that the first source operand (mmreg1) was the output of either the PFRCPIT1 or PFRSQIT1 instructions and the second source operand (mmreg2/mem64) was the output of either the PFRCP or PFRSQRT instructions. Refer to “Appendix A” on page 59 for an application-specific example of how to use this instruction and related instructions.

In the following code example, the bold line illustrates the PFRCPIT2 instruction in a sequence used to compute q = a/b accurate to 24 bits:

```
X0 =     PFRCP(b)
X1 =     PFRCPIT1(b,X0)
X2 =     PFRCPIT2(X1,X0)
q  =     PFMUL(a,X2)
```

Table 16. Numerical Range for the PFRCPIT2 Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	+/– 0 *	+/– 0 *	+/– 0 *
	Normal	+/– 0 *	Normal, +/– 0 **	Undefined
	Unsupported	+/– 0 *	Undefined	Undefined

Notes:

* *The sign of the result is the exclusive-OR of the signs of the source operands.*

** *If the absolute value of the result is less then 2^{-126}, the result is zero with the sign being the exclusive-OR of the signs of the source operands. If the absolute value of the product is greater than or equal to 2^{128}, the result is the largest normal number with the sign being exclusive-OR of the signs of the source operands.*

Related Instructions

See the PFRCPIT1 instruction.

See the PFRSQIT1 instruction.

See the PFRCP instruction.

See the PFRSQRT instruction.

PFRSQIT1

mnemonic	*opcode/imm8*	*description*
PFRSQIT1 mmreg1, mmreg2/mem64	0Fh 0Fh / A7h	Packed floating-point reciprocal square root, first iteration step

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFRSQIT1 is a vector instruction that performs the first step in a Newton-Raphson iteration to refine the reciprocal square root approximation produced by the PFSQRT instruction (the second and final step is accurate to 24 bits). Table 17 on page 46 shows the numerical range of the PFRCPIT2 instruction.

The behavior of this instruction is only defined for those combinations of operands such that one source operand was the input to the PFRSQRT instruction and the other source operand is the square of the output of the same PFRSQRT instruction. Refer to "Appendix A" on page 59 for an application-specific example of how to use this instruction and related instructions.

In the following code example, the bold lines illustrate the PFMUL and PFRSQIT1 instructions in a sequence used to compute a = 1/sqrt (b) accurate to 24 bits:

X_0 = PFRSQRT(b)
X_1 = PFMUL(X_0,X_0)
X_2 = PFRSQIT1(b,X_1)
a = PFRCPIT2(X_2,X_0)

Table 17. Numerical Range for the PFRSQIT1 Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	+/– 0 *	+/– 0 *	+/– 0 *
	Normal	+/– 0 *	Normal **	Undefined
	Unsupported	+/– 0 *	Undefined	Undefined

Notes:
* *The sign of the result is the exclusive-OR of the signs of the source operands.*
** *The sign is 0.*

Related Instructions See the PFRCPIT2 instruction.

See the PFRSQRT instruction.

PFRSQRT

mnemonic	*opcode/imm8*	*description*
PFRSQRT mmreg1, mmreg2/mem64	0Fh 0Fh / 97h	Floating-point reciprocal square root approximation

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFRSQRT is a scalar instruction that returns a low-precision estimate of the reciprocal square root of the source operand. The single result value is duplicated in both high and low halves of this instruction's 64-bit result. The source operand is single-precision with a 24-bit significand, and the result is accurate to 15 bits. Negative operands are treated as positive operands for purposes of reciprocal square root computation, with the sign of the result the same as the sign of the source operand. Table 18 on page 48 shows the numerical range of the PFRSQRT instruction.

Increased accuracy (the full 24 bits of a single-precision significand) requires the use of two additional instructions (PFRSQIT1 and PFRCPIT2). The first stage of this increase or refinement in accuracy (PFRSQIT1) requires that the input and squared output of the already executed PFRSQRT instruction be used as input to the PFRSQIT1 instruction. Refer to "Appendix A" on page 59 for an application-specific example of how to use this instruction and related instructions.

The PFRSQRT instruction performs the following operations:

```
mmreg1[31:0] = reciprocal square root(mmreg2/mem64[31:0])
mmreg1[63:32] = reciprocal square root(mmreg2/mem64[31:0])
```

In the following code example, the bold line illustrates the PFRSQRT instruction in a sequence used to compute a = 1/sqrt (b) accurate to 24 bits:

X_0 = PFRSQRT(b)
X_1 = PFMUL(X_0,X_0)
X_2 = PFRSQIT1(b,X_1)
a = PFRCPIT2(X_2,X_0)

Table 18. Numerical Range for the PFRSQRT Instruction

		Source 1 and Destination
Source 2	0	+/– Maximum Normal*
	Normal	Normal *
	Unsupported	Undefined *

Notes:
* *The result has the same sign as the source operand.*

Related Instructions See the PFRSQIT1 instruction.

See the PFRCPIT2 instruction.

PFSUB

mnemonic	*opcode/imm8*	*description*
PFSUB mmreg1, mmreg2/mem64	0Fh 0Fh / 9Ah	Packed floating-point subtraction

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFSUB is a vector instruction that performs subtraction of the source operand from the destination operand. Both operands are single-precision, floating-point operands with 24-bit significands. Table 19 on page 50 shows the numerical range of the PFSUB instruction.

The PFSUB instruction performs the following operations:

```
mmreg1[31:0] = mmreg1[31:0] - mmreg2/mem64[31:0]
mmreg1[63:32] = mmreg1[63:32] - mmreg2/mem64[63:32]
```

Table 19. Numerical Range for the PFSUB Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	+/– 0 *	Source 2	Source 2
	Normal	Source 1	Normal, +/– 0 **	Undefined
	Unsupported	Source 1	Undefined	Undefined

Notes:

* *The sign of the result is the logical AND of the sign of source 1 and the inverse of the sign of source 2.*

** *If the absolute value of the result is less then 2^{-126}, the result is zero with the sign being the sign of the source operand that is larger in magnitude (if the magnitudes are equal, the sign of source 1 is used). If the absolute value of the result is greater than or equal to 2^{128}, the result is the largest normal number with the sign being the sign of the source operand that is larger in magnitude.*

Related Instructions See the PFSUBR instruction.

PFSUBR

mnemonic	*opcode/imm8*	*description*
PFSUBR mmreg1, mmreg2/mem64	0Fh 0Fh / AAh	Packed floating-point reverse subtraction

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFSUBR is a vector instruction that performs subtraction of the destination operand from the source operand. Both operands are single-precision, floating-point operands with 24-bit significands. Table 20 on page 52 shows the numerical range of the PFSUBR instruction.

The PFSUBR instruction performs the following operations:

mmreg1[31:0] = mmreg2/mem64[31:0] – mmreg1[31:0]
mmreg1[63:32] = mmreg2/mem64[63:32] – mmreg1[63:32]

Table 20. Numerical Range for the PFSUBR Instruction

		Source 2		
		0	**Normal**	**Unsupported**
Source 1 and Destination	**0**	+/– 0 *	Source 2	Source 2
	Normal	Source 1	Normal, +/– 0 **	Undefined
	Unsupported	Source 1	Undefined	Undefined

Notes:

* *The sign of the result is the logical AND of the sign of source 1 and the inverse of the sign of source 2.*

** *If the absolute value of the result is less then 2^{-126}, the result is zero with the sign being the sign of the source operand that is larger in magnitude (if the magnitudes are equal, the sign of source 2 is used). If the absolute value of the result is greater than or equal to 2^{128}, the result is the largest normal number with the sign being the sign of the source operand that is larger in magnitude.*

Related Instructions See the PFSUB instruction.

PI2FD

mnemonic	*opcode/imm8*	*description*
PI2FD mmreg1, mmreg2/mem64	0Fh 0Fh / 0Dh	Packed 32-bit integer to floating-point conversion

Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PI2FD is a vector instruction that converts a vector register containing signed, 32-bit integers to single-precision, floating-point operands. When PI2FD converts an input operand with more significant digits than are available in the output, the output is truncated.

The PI2FD instruction performs the following operations:

```
mmreg1[31:0] = float(mmreg2/mem64[31:0])
mmreg1[63:32] = float(mmreg2/mem64[63:32])
```

Related Instructions See the PF2ID instruction.

PMULHRW

mnemonic	*opcode/suffix*	*description*
PMULHRW mmreg1, mmreg2/mem64	0F 0Fh/B7h	Multiply signed packed 16-bit values with rounding and store the high 16 bits.

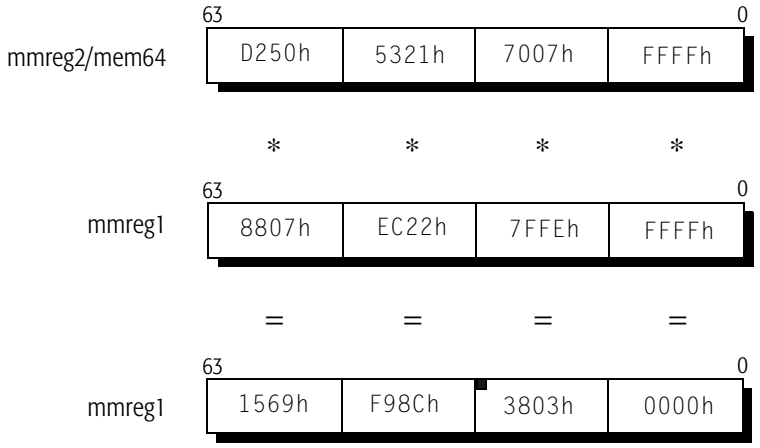
Privilege: None
Registers Affected: MMX
Flags Affected: None
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PMULHRW instruction multiplies the four signed 16-bit integer values in the source operand (an MMX register or a 64-bit memory location) by the four corresponding signed 16-bit integer values in the destination operand (an MMX register). The PMULHRW instruction then adds 8000h to the lower 16 bits of the 32-bit result, which results in the rounding of the high-order, 16-bit result. The high-order 16 bits of the result (including the sign bit) are stored in the destination operand.

The PMULHRW instruction provides a numerically more accurate result than the PMULMH instruction, which truncates the result instead of rounding.

Functional Illustration of the PMULHRW Instruction



■ Indicates a value that was rounded-up

The following list explains the functional illustration of the PMULHRW instruction:

- The signed 16-bit negative value D250h (–2DB0h) is multiplied by the signed 16-bit negative value 8807h (–77F9h) to produce the signed 32-bit positive result of 1569_4030h. 8000h is then added to the lower 16 bits to produce a final result of 1569_C030h. This rounding does not affect the final result of 1569h. The signed high-order 16 bits of the result are stored in the destination operand.
- The signed 16-bit positive value 5321h is multiplied by the signed 16-bit negative value EC22h (–13DEh) to produce the signed 32-bit negative result of F98C_7662h (–0673_899Eh). 8000h is then added to the lower 16 bits, producing a final result of F98C_F662h. This rounding does not affect the final result of F98Ch. The signed high-order 16 bits of the result are stored in the destination operand.
- The signed 16-bit positive value 7007h is multiplied by the signed 16-bit positive value 7FFEh to produce the signed 32-bit positive result of 3802_9FF2h. 8000h is then added to the lower 16 bits to produce a final result of 3803_1FF2h. This result has been rounded up. The signed high-order 16 bits of the result (3803h) are stored in the destination operand.
- The signed 16-bit negative value FFFFh (–1) is multiplied by the signed 16-bit negative value FFFFh (–1) to produce the signed 32-bit positive result of 0000_0001h. 8000h is then added to the lower 16 bits to produce a final result of 0000_8001h. This rounding does not affect the final result of 0000h. The signed high-order 16 bits of the result are stored in the destination operand.

PREFETCH/PREFETCHW

mnemonic	*opcode*	*description*
PREFETCH(W) mem8	0F 0Dh	Prefetch processor cache line into L1 data cache (Dcache)

Privilege:	none
Registers Affected:	none
Flags Affected:	none
Exceptions Generated:	none

The PREFETCH instruction loads a processor cache line into the data cache. The address of this line is specified by the mem8 value. For the AMD-K6-2 processor, the line size is 32 bytes. In all future processors, the size of the line that is loaded by the PREFETCH instruction will be at least 32-bytes. The PREFETCH instruction loads a cache line even if the mem8 address is not aligned with the start of the line (although some implementations, including the AMD-K6 family of processors, may perform the cache fill starting from the cache miss or mem8 address). If a cache hit occurs (the line is already in the Dcache) or a memory fault is detected, no bus cycle is initiated and the instruction is treated as a NOP.

In applications where a large number of data sets must be processed, the PREFETCH instruction can pre-load the next data set into the Dcache while, simultaneously, the processor is operating on the present set of data. This instruction allows the programmer to explicitly code operation concurrency. When the present set of data values is completed, the next set is already available in the Dcache. An example of a concurrent operation is vertices processing in 3D transformations, where the next set of vertices can be prefetched into the data cache while the present set is being transformed.

The PREFETCH instruction format in the AMD-K6-2 processor is defined to allow extensions in future AMD K86™ processors. The instruction mnemonic for the PREFETCH instruction includes the modR/M byte. Only the memory form of modR/M is valid (use of the register form results in an invalid opcode exception). Because there is no destination register, the three destination register field bits of the modR/M byte are used to define the type of prefetch to be performed. The PREFETCH and PREFETCHW instructions are defined by the bit pattern 000b and 001b, respectively. All other bit patterns are reserved for future use.

The PREFETCHW instruction loads the prefetched line and sets the cache line MESI state to modified (in anticipation of subsequent data writes to the line), unlike the PREFETCH instruction, which typically sets the state to exclusive. If the data that is prefetched into the Dcache is to be modified, use of the PREFETCHW instruction

will save the cycle that the PREFETCH instruction requires for modifying the Dcache line state. The PREFETCHW instruction should be used when the programmer expects that the data in the cache line will be modified. Otherwise, the PREFETCH instruction should be used.

Note: *The AMD-K6-2 processor executes the PREFETCHW instruction identically to the PREFETCH instruction. However, future AMD processors that support PREFETCHW as described above will be able to take advantage of the performance benefit provided by this instruction.*

Table 21 summarizes the PREFETCH type options:

Table 21. Summary of PREFETCH Instruction Type Options

Mod R/M	Result
11-xxx-xxx	Invalid Opcode
mm-000-xxx	PREFETCH
mm-001-xxx	PREFETCHW
mm-010-xxx	Reserved
mm-011-xxx	Reserved
mm-100-xxx	Reserved
mm-101-xxx	Reserved
mm-110-xxx	Reserved
mm-111-xxx	Reserved

Note: *The "Reserved" PREFETCH types do not result in an Invalid Opcode Exception if executed. Instead, for forward compatibility with future processors that may implement additional forms of the PREFETCH instruction, all "Reserved" PREFETCH types are implemented as synonyms for the basic PREFETCH type (for example, the PREFETCH instruction with type 000b).*

Appendix A

Division and Square Root

Division

The 3DNow! instructions can be used to compute a very fast, highly accurate reciprocal or quotient.

Consider the quotient q = a/b. An on-chip, ROM-based table lookup can be used to quickly produce a 14–15 bit precision approximation of 1/b (using just one two-cycle latency instruction—PFRCP). A full-precision reciprocal can then quickly be computed from this approximation using a Newton-Raphson algorithm.

The general Newton-Raphson recurrence for the reciprocal is as follows:

$$X_{i+1} = X_i \bullet (2 - b \bullet X_i)$$

Given that the initial approximation X_0 is accurate to at least 14 bits, and that full IEEE single precision contains 24 bits of mantissa, just one Newton-Raphson iteration is required. The following shows the 3DNow! instruction sequence to produce the initial reciprocal approximation, to compute the full-precision reciprocal from this, and lastly, to complete the required division of a/b.

X_0 = PFRCP(b)

X_1 = PFRCPIT1(b, X_0)

X_2 = PFRCPIT2(X_1, X_0)

q = PFMUL(a, X_2)

The 24-bit final reciprocal value is X_2. In the AMD-K6-2 processor implementation, the estimate contains the correct round-to-nearest value for approximately 99% of all arguments. The remaining arguments differ from the correct round-to-nearest value for the reciprocal by 1 unit-in-the-last-place (ulp). The quotient is formed in the last step by multiplying the reciprocal by the dividend a.

Divide Examples

These examples illustrate the use of 3DNow! instructions to perform divides.

(14-Bit Precision)

```
MOVD      MM0, [mem]    ;     0 | w
PFRCP     MM0, MM0      ;   1/w | 1/w     (approx.)
MOVQ      MM2, [mem]    ;     y | x
PFMUL     MM2, MM0      ;   y/w | x/w
```

(24-Bit Precision)

```
MOVD      MM0, [mem]    ;     0 | w
PFRCP     MM1, MM0      ;   1/w | 1/w     (approx.)
PUNPCKLDQ MM0, MM0      ;     w | w       (MMX instruction)
PFRCPIT1  MM0, MM1      ;   1/w | 1/w     (intermed.)
MOVQ      MM2, [mem]    ;     y | x
PFRCPIT2  MM0, MM1      ;   1/w | 1/w     (full prec.)
PFMUL     MM2, MM0      ;   y/w | x/w
```

Note: *For a description of the PUNPCKLDQ instruction, see the AMD-K6® Processor Multimedia Technology Manual, order# 20726.*

Square Root

The 3DNow! instructions can also be used to compute a reciprocal square root or square root with high performance. The general Newton-Raphson reciprocal square root recurrence is as follows:

$$X_{i+1} = 1/2 \bullet X_i \bullet (3 - b \bullet X_i^2)$$

To reduce the number of iterations, X_0 is an initial approximation read from a table. The 3DNow! reciprocal square root approximation is accurate to at least 15 bits. Accordingly, to obtain a single-precision 24-bit reciprocal square root of an input operand b, one Newton-Raphson iteration is required using the following 3DNow! instructions:

1. X_0 = PFRSQRT(b)
2. X_1 = PFMUL(X_0, X_0)
3. X_2 = PFRSQIT1(b, X_1)
4. X_3 = PFRCPIT2(X_2, X_0)
5. X_4 = PFMUL(b, X_3)

The 24-bit final reciprocal square root value is X_3. In the AMD-K6-2 implementation, the estimate contains the correct round-to-nearest value for approximately 87% of all arguments. The remaining arguments differ from the correct round-to-nearest value by 1 ulp. The square root (X_4) is formed in the last step by multiplying by the input operand b.

Square Root Examples

These examples illustrate the use of 3DNow! technology to perform square roots.

(15-Bit Precision)

```
MOVD      MM0, [mem]   ;          0 | a
PFRSQRT   MM1, MM0     ; 1/(sqrt a) | 1/(sqrt a)  (approx.)
PFMUL     MM0, MM1     ;   (sqrt a) | (sqrt a)
```

(24-Bit Precision)

```
MOVD      MM0, [mem]   ;          0 | a
PFRSQRT   MM1, MM0     ; 1/(sqrt a) | 1/(sqrt a)  (approx.)
MOVQ      MM2, MM1     ; 1/(sqrt a) | 1/(sqrt a)  (approx.)
PFMUL     MM1, MM1     ;   (sqrt a) | (sqrt a)    step 1
PFRSQIT1  MM1, MM0     ;    (sqrt) (intermed.)    step 2
PFRCPIT2  MM1, MM2     ;   (sqrt) (full prec.)    step 3
PFMUL     MM0, MM1     ;   (sqrt a) | (sqrt a)
```