AMD-K6™

MMX™ Enhanced Processor

Multimedia Technology

AMD

Contents

Revision History

Date	Rev	Description
July 1996	A	Initial release.
March 1997	B	Removed paragraph from “Mixing MMX™ and Floating-Point Instructions” on page 14 that contained inaccuracies pertaining to floating-point tag words.
June 1997	C	Revised stack exception entry in Table 1, “MMX™ Instruction Exceptions,” on page 13 to include Real mode and Virtual-8086 mode.
June 1997	C	Revised note 2 on page 13 regarding floating-point exceptions.
June 1997	C	Replaced overbar with # to indicate active-Low signals.
June 1997	C	Revised document to comply with MMX trademark.
June 1997	C	Revised description of EMMS instruction on page 18.
Jan 2000	D	Changed mem64 to mem32 for PUNPCKLBW, PUNPCKLWD, and PUNPCKLDQ.

1

AMD-K6™ Processor Multimedia Technology

Introduction

Next generation PC performance requirements are being driven by emerging multimedia and communications software. 3D graphics, video, audio, and telephony capabilities are evolving across education, entertainment, and internet applications. As multimedia applications continue to proliferate in the marketplace, PC systems suppliers are being challenged to deliver multimedia-enabled PC solutions covering all mainstream price/performance points.

In response to the growing need to provide improved PC multimedia capabilities, the AMD-K6™ MMX™ enhanced processor is the first member in the AMD family of processors to incorporate a robust multimedia technology that is fully software compatible with the MMX™ technology as defined by Intel. This multimedia technology enables scaleable multimedia capabilities across a broad range of PC system price/performance points.

The AMD-K6 processor features a decode-decoupled superscalar microarchitecture and state-of-the-art design techniques to deliver true sixth-generation performance while maintaining full x86 binary software compatibility. An x86 binary-compatible processor implements the industry-standard x86 instruction set by decoding and executing the x86 instruction set as its native mode of operation. Only this native mode enables delivery of maximum performance when running PC software.

The AMD-K6 processor delivers leading-edge performance to mainstream PC systems running industry-standard x86 software. The AMD-K6 processor implements advanced design techniques like instruction pre-decoding, dual x86 opcode decoding, single-cycle internal RISC operations, parallel execution units, out-of-order execution, data forwarding, register renaming, and dynamic branch prediction. In other words, the AMD-K6 is capable of issuing, executing, and retiring multiple x86 instructions per cycle, resulting in superior scaleable performance.

This document describes the multimedia technology of the AMD-K6 processor, including data types, instructions, and programming considerations.

Multimedia Technology Architecture

The multimedia technology in the AMD-K6 MMX enhanced processor is designed to accelerate media and communication applications. Specialized applications that use music synthesis, speech synthesis, speech recognition, audio and video compression and decompression, full motion video, 2D and 3D graphics, and video conferencing, can take advantage of the AMD-K6 processor multimedia technology. The multimedia technology implements new instructions, new data types, and powerful parallel processing (Single Instruction Multiple Data, SIMD) techniques that can significantly increase the performance of these applications.

Key Functionality

At the lowest levels, multimedia applications (audio, video, 3D graphics, and telephony, etc.) contain many similar functions. When these functions are performed on a processor that does not have MMX capability, the processor is heavily burdened by the computational requirements of this information. Processors executing the MMX instructions increase the performance of multimedia applications. This performance increase is a direct result of the increased multimedia bandwidth of the processor.

Multimedia applications must process large amounts of data. Parallel data computing is exemplified by applications that manipulate screen pixel information. Instead of acting on one pixel at a time, multimedia technology enables the system to act on multiple pixels simultaneously. This Single Instruction Multiple Data (SIMD) model is a key feature of MMX technology.

The AMD-K6 processor multimedia technology architecture includes four new MMX data types, 57 new MMX instructions, eight new 64-bit MMX registers, and an SIMD processing pipeline. The multimedia technology is compatible with existing x86 applications.

The 57 new MMX instructions include arithmetic functions, packing and unpacking functions, logical operations, and moves. These are the basic functions that are most commonly used in repetitive computational multimedia programs.

Multimedia applications often use smaller operands—8-bit data is commonly used for pixel information and 16-bit data is used for audio samples. The new MMX registers allow data to be packed into 64-bit operands. For example, 8-bit data (1 byte) can be packed in sets of eight in a single 64-bit register, and all eight bytes can be operated on simultaneously by a single MMX instruction.

For 256-color video modes, this translates to computing eight pixels per instruction. When an entire screen is being re-drawn, these pixel manipulation routines often use highly repetitive loops. Parallel processing of eight pieces of data can reduce the processing time of a code loop by up to a factor of eight.

Multimedia applications frequently multiply and accumulate data. The multimedia technology provides instructions that add, multiply, and even combine these operations. For example, the PMADDWD instruction can multiply and then add words of data in a single instruction that uses far less processor cycles than the equivalent x86 operations.

Executing MMX™ Instructions

A programmer must approach the use of MMX instructions differently, based on whether the code being developed is at the system level or at the application level. The details of these differences are discussed in “Programming Considerations” on page 9.

Before using the MMX instructions, the programmer must use the CPUID instruction to determine if the processor supports multimedia technology. See the *AMD Processor Recognition Application Note*, order# 20734, for more information.

Function 1 (EAX=1) of the AMD-K6 processor CPUID instruction returns the processor feature bits in the EDX register. Software can then test bit 23 of the feature bits to determine if the processor supports the multimedia technology. If bit 23 is set to 1, MMX instructions are supported. All AMD-K6 processors have bit 23 set. Once it is determined that multimedia technology is supported, subsequent code can use the MMX instructions. Alternatively, the AMD 8000_0001h extended CPUID function can be used to test whether the processor supports multimedia technology.

After a module of MMX code has executed, the programmer must empty the MMX state by executing the EMMS command. Because the MMX registers share the floating-point registers, an instruction is needed to prevent MMX code from interfering with floating-point. The EMMS command clears the multimedia state and resets all the floating-point tag bits. Emptying the MMX state sets the floating-point tag bits to empty (all ones), which marks the MMX/FP registers as invalid and available.

Register Set

The AMD-K6 processor implements eight new 64-bit MMX registers. These registers are mapped on the floating-point registers. As shown in Figure 1 on page 5, the new MMX instructions refer to these registers as mmreg0 to mmreg7. Mapping the new MMX registers on the floating-point stack enables backwards compatibility for the register saving that must occur as a result of task switching.

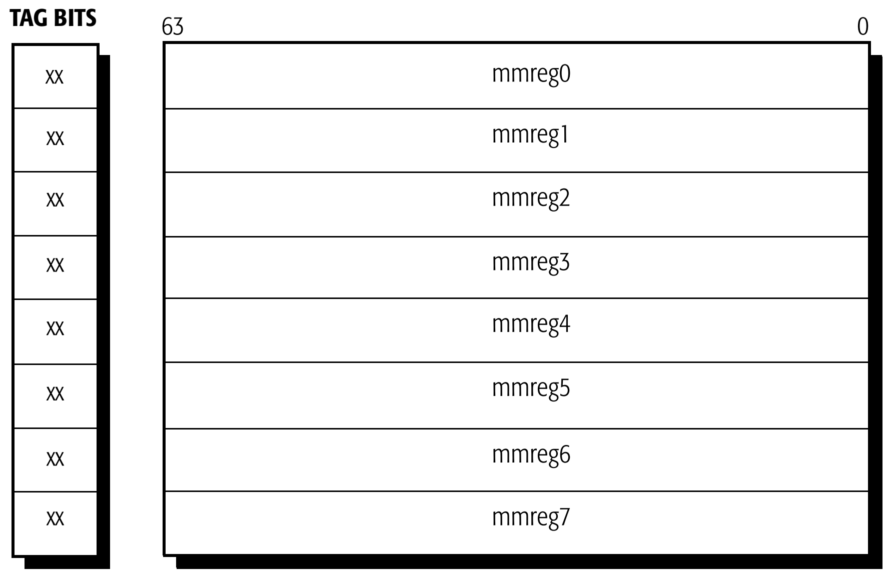


Figure 1. MMX™ Registers

Aliasing the MMX registers onto the floating-point stack registers provides a safe way to introduce this new technology. Instead of needing to modify operating systems, new MMX applications can be supported through device drivers, MMX libraries, or DLL files. See the *Programming Considerations* section of this document for more information.

Current operating systems have support for floating-point operations. Using the floating-point registers for MMX code is an ingenious way of implementing automatic support for MMX instructions. Every time the processor executes an MMX instruction, all the floating-point register tag bits are set to zero (00b=valid). Setting the tag bits after every MMX instruction prevents the processor from having to perform extra tasks. These extra tasks are normally executed on floating-point registers when the Tag field is something other than 00b.

If a task switch occurs during an MMX or floating-point instruction, 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 or MMX instruction, allowing the operating system to save the state of the MMX/FP registers.

If there is a task switch when 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.

Data Types

The AMD-K6 processor multimedia technology uses a packed data format. The data is packed in a single, 64-bit MMX register or memory operand as eight bytes, four words, or two double words. Each byte, word, doubleword, or quadword is an integer data type.

The form of an instruction determines the data type. For example, the MOV instruction comes in two different forms—MOVD moves 32 bits of data and MOVQ moves 64 bits of data.

The four new data types are defined as follows:

Packed byte	Eight 8-bit bytes packed into 64 bits Signed integer range(-2^7 to 2^7-1) Unsigned integer range(0 to 2^8-1)
Packed word	Four 16-bit words packed into 64-bits Signed integer range(-2^{15} to $2^{15}-1$) Unsigned integer range(0 to $2^{16}-1$)
Packed doubleword	Two 32-bit doublewords packed into 64 bits Signed integer range(-2^{31} to $2^{31}-1$) Unsigned integer range(0 to $2^{32}-1$)
Quadword	One 64-bit quadword Signed integer range(-2^{63} to $2^{63}-1$) Unsigned integer range(0 to $2^{64}-1$)

Figure 2 on page 7 shows the four new data types.

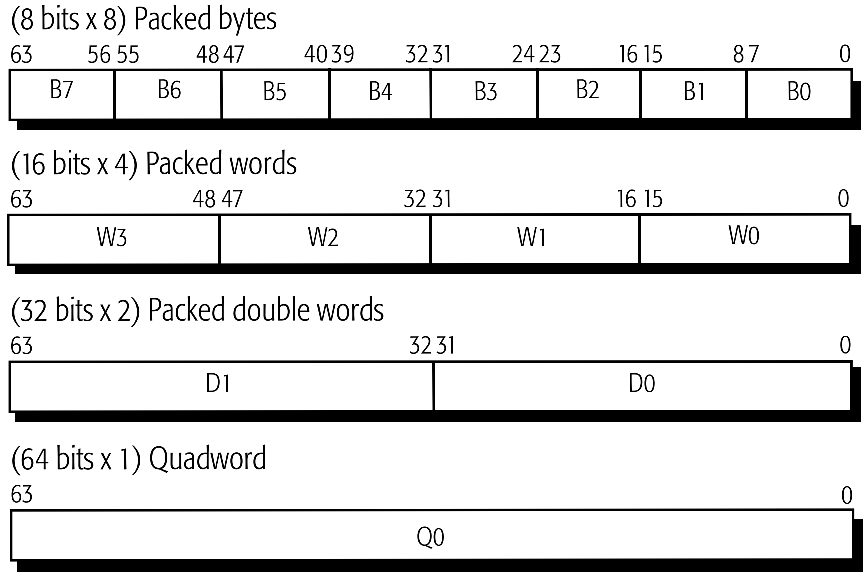


Figure 2. MMX™ Data Types

Instructions

The AMD-K6 processor multimedia technology includes 57 new MMX instructions. These new instructions are organized into the following groups:

- Arithmetic
- Empty MMX registers
- Compare
- Convert (pack/unpack)
- Logical
- Move
- Shift

The following mnemonics are used in the instructions:

- **P**—Packed data
- **B**—Byte
- **W**—Word
- **D**—Doubleword
- **Q**—Quadword
- **S**—Signed

- **U—**Unsigned
- **SS—**Signed Saturation
- **US—**Unsigned Saturation

For example, the mnemonic for the PACK instruction that packs four words into eight unsigned bytes is PACKUSWB. In this mnemonic, the US designates an unsigned result with saturation, and the WB means that the source is packed words and the result is packed bytes.

The term *saturation* is commonly used in multimedia applications. Saturation allows mathematical limits to be placed on the data elements. If a result exceeds the boundary of that data type, the result is set to the defined limit for that instruction. A common use of saturation is to prevent color wraparound.

Instruction Formats

All MMX instructions, except the EMMS instruction that uses no operands, are formatted as follows:

```
INSTRUCTION mmreg1, mmreg2/mem64
```

The source operand (mmreg2/mem64) can be either an MMX register or a memory location. The destination operand (mmreg1) can only be an MMX register.

The MOVD and MOVQ instructions also have the following acceptable formats:

```
MOVD        mmreg1, mreg32/mem32
MOVD        mreg32/mem32, mmreg1
MOVQ        mem64, mmreg1
```

In the first example, the source operand (mreg32/mem32) can be either an integer register or a 32-bit memory address. The destination operand (mmreg1) can only be an MMX register. The second example has the source operand as an MMX register. The destination operand (mreg32/mem32) can be either an integer register or a 32-bit memory address. The third example has the source operand as an MMX register and the destination operand as a 64-bit memory location

The SHIFT instructions can also utilize an immediate source operand. It is designated as *imm8*.

```
PSRLW       mmreg1, imm8
```

2

Programming Considerations

This chapter describes considerations for programmers writing operating systems, compilers, and applications that utilize MMX instructions as implemented in the AMD-K6 MMX enhanced processor.

Feature Detection

To use the AMD-K6 processor multimedia technology, the programmer must determine if the processor supports them. The CPUID instruction gives programmers the ability to determine the presence of multimedia technology on the processor. Software 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
```

If the processor supports the CPUID instruction, the programmer must execute the standard function, EAX=0. The CPUID function returns a 12-character string that identifies the processor's vendor. For AMD processors, standard function 0 returns a vendor string of "Authentic AMD". This string requires the software to follow the AMD definitions for subsequent CPUID functions and the values returned for those functions.

The next step is for the programmer to determine if MMX instructions are supported. Function 1 of the CPUID instruction provides this information. Function 1 (EAX=1) of the AMD CPUID instruction returns the feature bits in the EDX register. If bit 23 in the EDX register is set to 1, MMX instructions are supported. The following code sample shows how to test for MMX instruction support.

```
mov   eax,1             ; setup function 1
CPUID                   ; call the function
test  edx, 800000       ; test 23rd bit
jnz   YES_MM            ; multimedia technology supported
```

Alternatively, the extended function 1 (EAX=8000_0001h) can be used to determine if MMX instructions are supported.

```
mov   eax,8000_0001h    ; setup extended function 1
CPUID                   ; call the function
test  edx, 800000       ; test 23rd bit
jnz   YES_MM            ; multimedia technology supported
```

Task Switching

A task switch is an event that occurs within operating systems that allows multiple programs to be executed in parallel. Most modern operating systems utilizing task switching, are called multitasking operating systems.

There are two types of multitasking operating systems—cooperative and preemptive.

Cooperative Multitasking

In cooperative multitasking operating systems, applications do not care about other tasks that may be running. Each task assumes that it owns the machine state (processor, registers, I/O, memory, etc.). In addition, these tasks must take care of saving their own information (i.e., registers, stacks, states) in their own memory areas. The cooperative multitasking operating system does not save operating state information for the applications.

There are different types of cooperative multitasking operating systems. Some of these operating systems perform some level of state saves, but this state saving is not always reliable. All software engineers programming for a cooperative multitasking environment must save the MMX or floating-point states before relinquishing control to another task or to the operating system. The FSAVE and FRSTOR commands are used to perform this task. Figure 4 illustrates this task switching process.

Note: *Some cooperative operating systems may have API calls to perform these tasks for the application.*

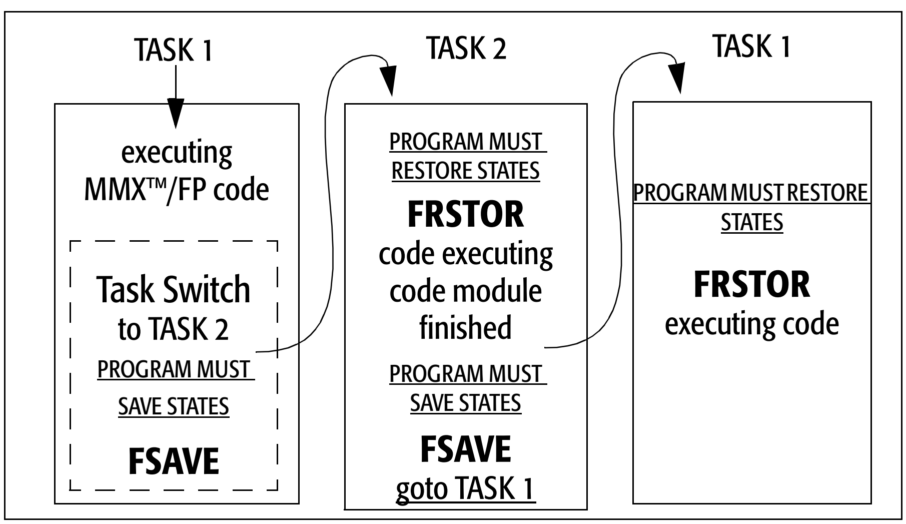


Figure 3. Cooperative Task Switching

Preemptive Multitasking

In preemptive multitasking operating systems like OS/2, Windows NT™, and UNIX, the operating system handles all state and register saves. The application programmer does not need to save states when programming within a preemptive multitasking environment. The preemptive multitasking operating system sets aside a save area for each task.

In a preemptive multitasking operating system, if a task switch occurs, the operating system sets the Control Register 0 (CR0) Task Switch (TS) bit to 1. If the new task encounters a floating-point or MMX instruction, an interrupt 7 (int 7, Device Not Available) is generated. The int7 handler saves the state of the first task and restores the state of the second task. The int7 handler sets the CR0.TS to 0 and returns to the original floating-point or MMX instruction in the second task. Figure 4 illustrates this task switching process.

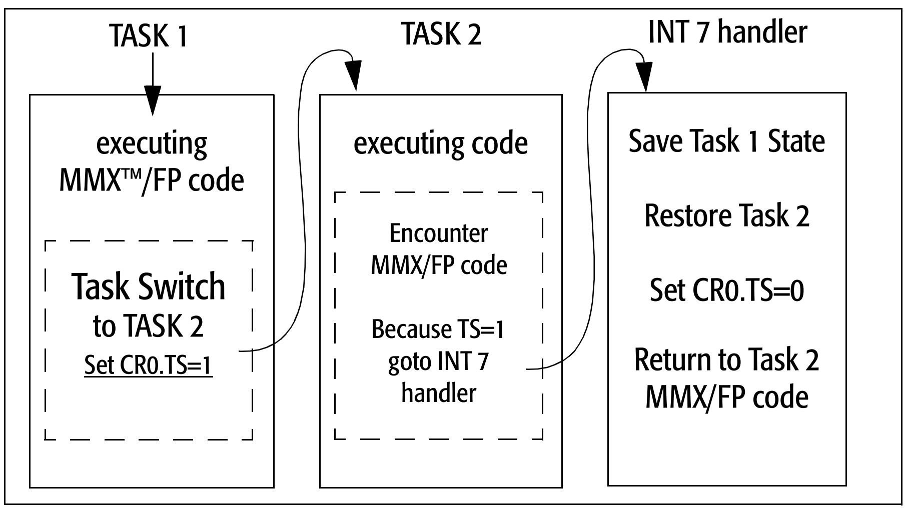


Figure 4. Preemptive Task Switching

Exceptions

Table 1 contains a list of exceptions that MMX instructions can generate.

Table 1. MMX™ Instruction Exceptions

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.
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 have not changed in the implementation of MMX instructions. None of the exception handlers need to be modified.

Note:

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

Mixing MMX™ and Floating-Point Instructions

The programmer must take care when writing code that contains both MMX and floating-point instructions. The MMX code modules should be separated from the floating-point code modules. All code of one type (MMX or floating-point code) should be grouped together as often as possible. To obtain the highest performance, routines should not contain any conditional branches at the end of loops that jump to code of a different type than the code that is currently being executed.

In certain multimedia environments, floating-point and MMX instructions may be mixed. For example, if a programmer wants to change the viewing perspective of a three-dimensional scene, the perspective can be changed through transformation matrices using floating-point registers. The picture/pixel information is integer-based and requires MMX instructions to manipulate this information. Both MMX and floating-point instructions are required to perform this task.

The software must clean up after itself at the end of an MMX code module. The EMMS instruction must be used at the end of an MMX code module to mark all floating-point registers as empty (11=empty/invalid). In cooperative multitasking operating systems, the EMMS instruction must be used when switching between tasks.

Note: *In some situations, experienced programmers can utilize the MMX registers to pass information between tasks. In these situations, the EMMS instruction is not required.*

The tag bits are affected by every MMX and floating-point instruction. After every MMX instruction except EMMS, all the tag bits in the floating-point tag word are set to 0. When the EMMS instruction is executed, all the tag bits in the tag word are set to 1.

Prefixes

All instructions in the x86 architecture translate to a binary value or opcode. This 1 or 2 byte opcode value is different for each instruction. If an instruction is two bytes long, the second byte is called the Mod R/M byte. The Mod R/M byte is used to further describe the type of instruction that is used.

The x86 opcode and the Mod R/M byte can also be followed by an SIB byte. This byte is used to describe the Scale, Index and Base forms of 32-bit addressing.

The format of the x86 instruction allows for certain prefixes to be placed before each instruction. These prefixes indicate different types of command overrides.

The MMX instructions follow these rules just like all the current existing instructions. This allows for an easy implementation into the x86 architecture. All of the rules that apply to the x86 architecture apply to MMX instructions, including accessing registers, memory, and I/O.

Most opcode prefixes can be utilized while using MMX instructions. The following prefixes can be used with MMX instructions:

- The Segment Override prefixes (2Eh/CS, 36h/SS, 3Eh/DS, 26h/ES, 64h/FS, and 65h/GS) affect MMX instructions that contain a memory operand.
- The LOCK prefix (F0h) triggers an invalid opcode exception (interrupt 6).
- The Address Size Override prefix (67h) affects MMX instructions that contain a memory operand.

3

MMX™ Instruction Set

The following MMX instruction definitions are in alphabetical order according to the instruction mnemonics.

EMMS

mnemonic	*opcode*	*description*
EMMS	0F 77h	Clear the MMX 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.

The EMMS instruction is used to clear the MMX state following the execution of a block of code using 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 code that includes floating-point instructions.

MOVD

mnemonic	*opcode*	*description*
MOVD mmreg1, reg32/mem32	0F 6Eh	Copy a 32-bit value from the general purpose register or memory location into the MMX register
MOVD reg32/mem32, mmreg1	0F 7Eh	Copy a 32-bit value from the MMX register into the general purpose register or memory location

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.
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 MOVD instruction moves a 32-bit data value from an MMX register to a general purpose register or memory, or it moves the 32-bit data from a general purpose register or memory into an MMX register. If the 32-bit data to be moved is provided by an MMX register, the instruction moves bits 31–0 of the MMX register into the specified register or memory location. If the 32-bit data is being moved into an MMX register, the instruction moves the 32-bits of data into bits 31–0 of the MMX register and fills bits 63–32 with zeros.

Related Instructions See the MOVQ instruction.

MOVQ

mnemonic	*opcode*	*description*
MOVQ mmreg1, mmreg2/mem64	0F 6Fh	Copy a 64-bit value from an MMX register or memory location into an MMX register
MOVQ mmreg2/mem64, mmreg1	0F 7Fh	Copy a 64-bit value from an MMX register into an MMX register or memory location

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.
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 MOVQ instruction moves a 64-bit data value from one MMX register to another MMX register or memory, or it moves the 64-bit data from one MMX register or memory to another MMX register. Copying data from one memory location to another memory location cannot be accomplished with the MOVQ instruction.

Related Instructions See the MOVD instruction.

PACKSSDW

mnemonic	*opcode*	*description*
PACKSSDW mmreg1, mmreg2/mem64	0F 6Bh	Pack with saturation signed 32-bit operands into signed 16-bit results

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

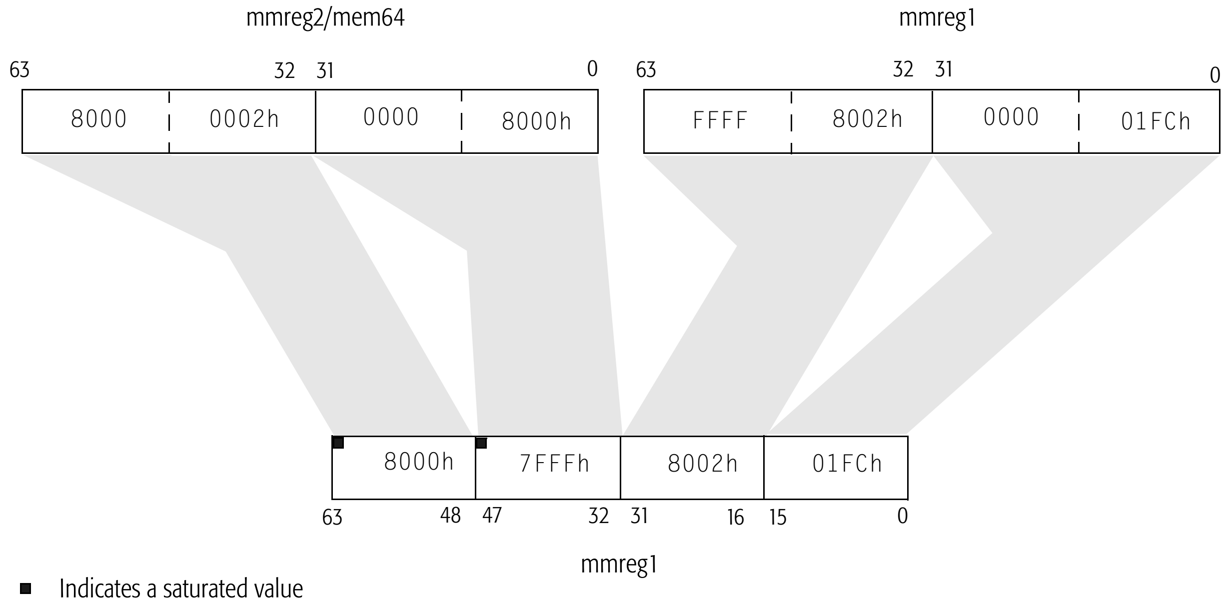
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (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 PACKSSDW instruction performs a pack and saturate operation on two signed 32-bit values in the first operand and two signed 32-bit values in the second operand. The four signed 16-bit results are placed in the specified MMX register.

The pack operation is a data conversion. The PACKSSDW instruction converts or packs the four signed 32-bit values into four signed 16-bit values, applying saturating arithmetic. If the signed 32-bit value is less than –32768 (8000h), it saturates to –32768 (8000h). If the signed 32-bit value is greater than 32767 (7FFFh), it saturates to 32767 (7FFFh). All values between –32768 and 32767 are represented with their signed 16-bit value.

The first operand must be an MMX register. In addition to providing the first operand, this MMX register is the location where the result of the pack and saturate operation is stored. The second operand can be an MMX register or a 64-bit memory location.

Functional Illustration of the PACKSSDW Instruction



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

- Bits 63–32 of the source operand (mmreg2/mem64) are packed into bits 63–48 of the destination operand (mmreg1). The result is saturated to the largest possible 16-bit negative number because the 32-bit negative source operand (8000_0002h) exceeds the capacity of the signed 16-bit destination operand.
- Bits 31–0 of the source operand are packed into bits 47–32 of the destination operand. The result is saturated to the largest possible 16-bit positive number because the 32-bit positive source operand (0000_8000h) exceeds the capacity of the 16-bit destination operand.
- Bits 63–32 of the destination operand are packed into bits 31–16 of the destination operand. The results are not saturated because the 32-bit negative source operand (FFFF_8002h) does not exceed the capacity of the 16-bit destination operand.
- Bits 31–0 of the destination operand are packed into bits 15–0 of the destination operand. The results are not saturated because the 32-bit positive source operand (0000_01FCh) does not exceed the capacity of the 16-bit destination operand.

Related Instructions See the PACKSSWB instruction.

See the PACKUSWB instruction.

See the PUNPCKHWD instruction.

See the PUNPCKLWD instruction.

PACKSSWB

mnemonic	*opcode*	*description*
PACKSSWB mmreg1, mmreg2/mem64	0F 63h	Pack with saturation signed 16-bit operands into signed 8-bit results

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

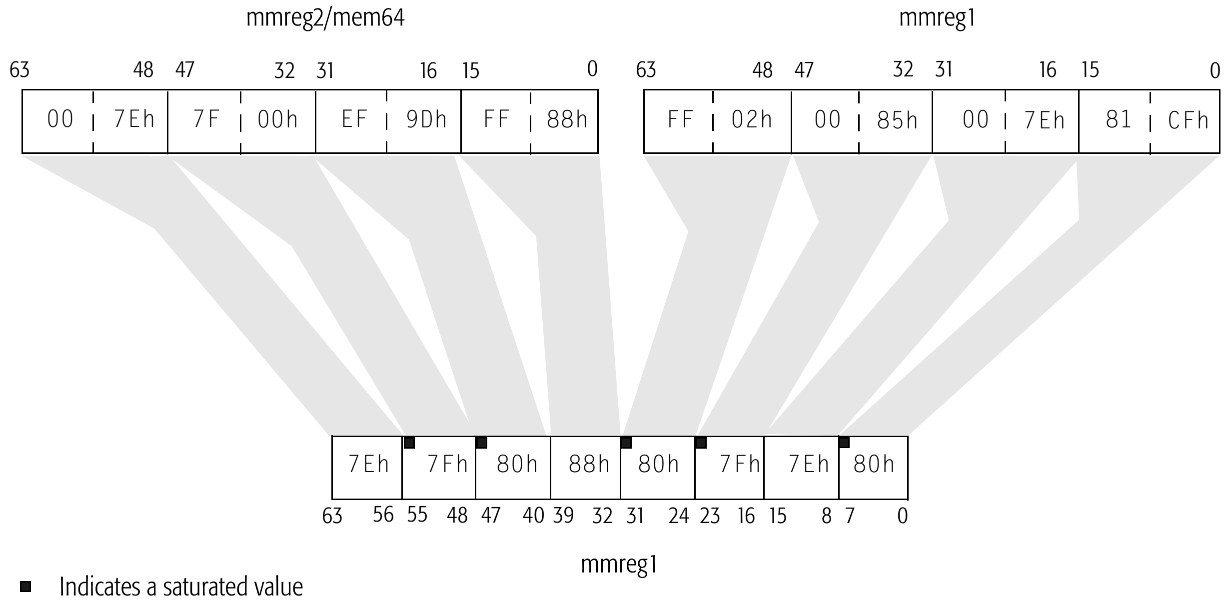
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (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 PACKSSWB instruction performs a pack and saturate operation on four signed 16-bit values in the first operand and four signed 16-bit values in the second operand. The eight signed 8-bit results are placed in the specified MMX register.

The pack operation is a data conversion. The PACKSSWB instruction converts or packs the eight signed 16-bit values into eight signed 8-bit values, applying saturating arithmetic. If the signed 16-bit value is less than –128 (80h), it saturates to –128 (80h). If the signed 16-bit value is greater than 127 (7Fh), it saturates to 127 (7Fh). All values between –128 and 127 are represented by their signed 8-bit value.

The first operand must be an MMX register. In addition to providing the first operand, this MMX register is the location where the result of the pack and saturate operation is stored. The second operand can be an MMX register or a 64-bit memory location.

Functional Illustration of the PACKSSWB Instruction



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

- Bits 63–48 of the source operand (mmreg2/mem64) are packed into bits 63–56 of the destination operand (mmreg1). The result is not saturated because the 16-bit positive source operand (007Eh) does not exceed the capacity of a signed 8-bit destination operand.
- Bits 47–32 of the source operand are packed into bits 55–48 of the destination operand. The result is saturated to the largest possible 8-bit positive number because the 16-bit positive source operand (7F00h) exceeds the capacity of a signed 8-bit destination operand.
- Bits 31–16 of the source operand are packed into bits 47–40 of the destination operand. The result is saturated to the largest possible 8-bit negative number because the 16-bit negative source operand (EF9Dh) exceeds the capacity of a signed 8-bit destination operand.
- Bits 15–0 of the source operand are packed into bits 39–32 of the destination operand. The result is not saturated because the 16-bit negative source operand (FF88h) does not exceed the capacity of the 8-bit destination operand.
- Bits 63–48 of the destination operand are packed into bits 31–24 of the destination operand. The result is saturated to the largest possible 8-bit negative number because the 16-bit negative source operand (FF02h) exceeds the capacity of a signed 8-bit destination operand.

- Bits 47–32 of the destination operand are packed into bits 23–16 of the destination operand. The result is saturated to the largest possible 8-bit positive number because the 16-bit positive source operand (0085h) exceeds the capacity of a signed 8-bit destination operand.
- Bits 31–16 of the destination operand are packed into bits 15–8 of the destination operand. The result is not saturated because the 16-bit positive source operand (007Eh) does not exceed the capacity of a signed 8-bit destination operand.
- Bits 15–0 of the destination operand are packed into bits 7–0 of the destination operand. The result is saturated to the largest possible 8-bit negative number because the 16-bit negative source operand (81CFh) exceeds the capacity of a signed 8-bit destination operand.

Related Instructions

See the PACKSSDW instruction.

See the PACKUSWB instruction.

See the PUNPCKHBW instruction.

See the PUNPCKLBW instruction.

PACKUSWB

mnemonic	*opcode*	*description*
PACKUSWB mmreg1, mmreg2/mem64	0F 67h	Pack with saturation signed16-bit operands into unsigned 8-bit results

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

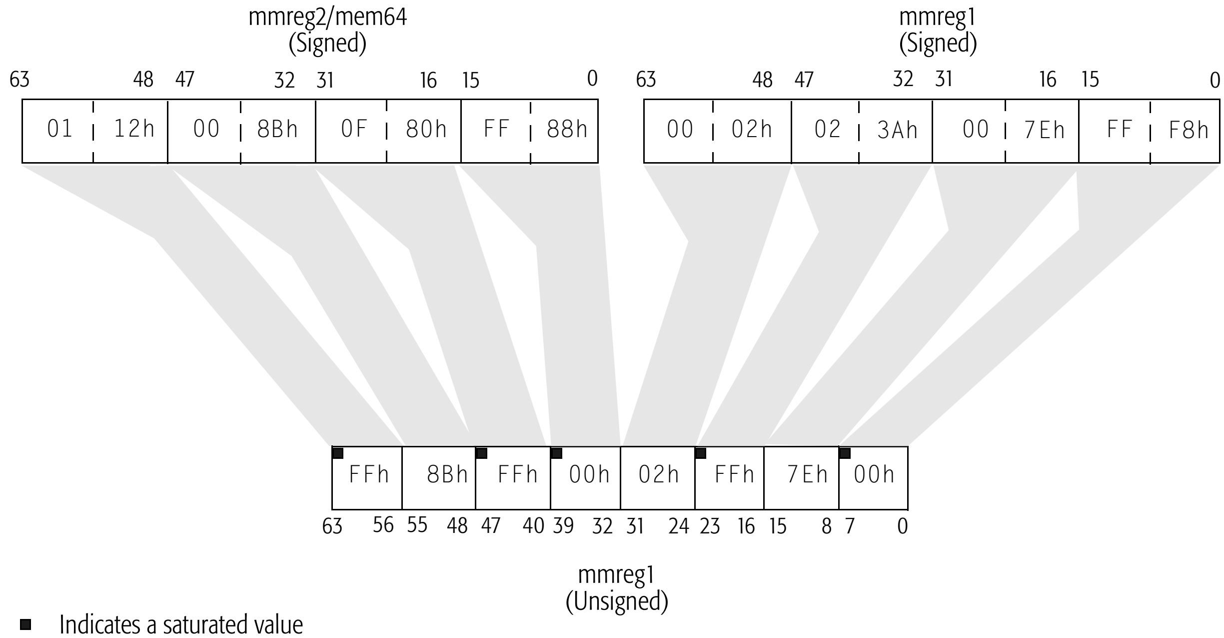
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (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 PACKUSWB instruction performs a pack and saturate operation on four signed 16-bit values in the first operand and four signed 16-bit values in the second operand. The eight unsigned 8-bit results are placed in the specified MMX register.

The pack operation is a data conversion. The PACKUSWB instruction converts or packs the eight signed 16-bit values into eight unsigned 8-bit values, applying saturating arithmetic. If the signed 16-bit value is a negative number, it saturates to 0 (00h). If the signed 16-bit value is greater than 255 (FFh), it saturates to 255 (FFh). All values between 0 and 255 are represented with their unsigned 8-bit value.

The first operand must be an MMX register. In addition to providing the first operand, this MMX register is the location where the result of the pack and saturate operation is stored. The second operand can be an MMX register or a 64-bit memory location.

Functional Illustration of the PACKUSWB Instruction



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

- Bits 63–48 of the source operand (mmreg2/mem64) are packed into bits 63–56 of the destination operand (mmreg1). The result is saturated to the largest possible 8-bit positive number because the 16-bit positive source operand (0112h) exceeds the capacity of an unsigned 8-bit destination operand.
- Bits 47–32 of the source operand are packed into bits 55–48 of the destination operand. The result is not saturated because the 16-bit positive source operand (008Bh) does not exceed the capacity of an unsigned 8-bit destination operand.
- Bits 31–16 of the source operand are packed into bits 47–40 of the destination operand. The result is saturated to the largest possible 8-bit positive number because the 16-bit positive source operand exceeds the capacity of an unsigned 8-bit destination operand.
- Bits 15–0 of the source operand are packed into bits 39–32 of the destination operand. The result is saturated to 00h because the source operand (FF88h) is a negative value.
- Bits 63–48 of the destination operand are packed into bits 31–24 of the destination operand (mmreg1). The result is not saturated because the 16-bit positive source operand (0002h) does not exceed the capacity of an unsigned 8-bit destination operand.
- Bits 47–32 of the destination operand are packed into bits 23–16 of the destination operand. The result is saturated to the largest possible 8-bit positive number

because the 16-bit positive source operand (023Ah) exceeds the capacity of an unsigned 8-bit destination operand.

- Bits 31–16 of the destination operand are packed into bits 15–8 of the destination operand. The result is not saturated because the 16-bit positive source operand (007Eh) does not exceed the capacity of an unsigned 8-bit destination operand.
- Bits 15–0 of the destination operand are packed into bits 7–0 of the destination operand. The result is saturated to 00h because the source operand (FFF8h) is a negative value.

Related Instructions

See the PACKSSDW instruction.

See the PACKSSWB instruction.

See the PUNPCKHBW instruction.

See the PUNPCKLBW instruction.

PADDB

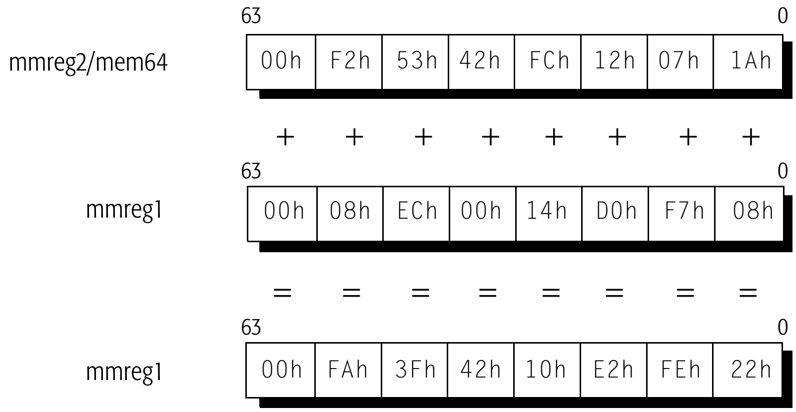
mnemonic	*opcode*	*description*
PADDB mmreg1, mmreg2/mem64	0F FCh	Add unsigned packed 8-bit values

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

Exception	**Real**	**Virtual 8086**	**Protected**	**Description**
Invalid opcode (6)	X	X	X	The emulate 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.
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 PADDB instruction adds eight unsigned 8-bit values from the source operand (an MMX register or a 64-bit memory location) to the eight corresponding unsigned 8-bit values in the destination operand (an MMX register). If any of the eight results is greater than the capacity of its 8-bit destination, the value wraps around with no carry into the next location. The eight 8-bit results are stored in the MMX register that is specified as the destination operand.

Functional Illustration of the PADDB Instruction



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

- The value 53h is added to ECh and wraps around to 3Fh.
- The value FCh is added to 14h and wraps around to 10h.
- The remaining addition operations are simple unsigned operations with no wraparound.

Related Instructions

See the PADDD instruction.

See the PADDW instruction.

See the PADDSB instruction.

See the PADDSW instruction.

See the PADDUSB instruction.

See the PADDUSW instruction.

PADDD

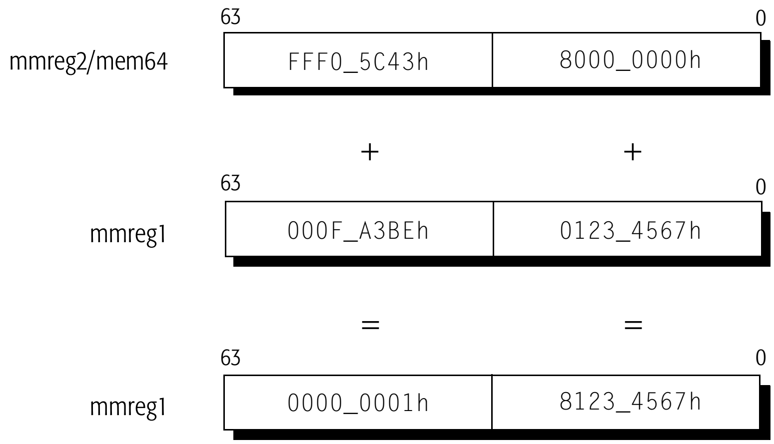
mnemonic	*opcode*	*description*
PADDD mmreg1, mmreg2/mem64	0F FEh	Add unsigned packed 32-bit values

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

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate 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.
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 PADDD instruction adds two unsigned 32-bit values from the source operand (an MMX register or a 64-bit memory location) to the two corresponding unsigned 32-bit values in the destination operand (an MMX register). If any of the two results is greater than the capacity of its 32-bit destination, the value wraps around with no carry into the next location. The two 32-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PADDD Instruction



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

- The value FFF0_5C43h is added to 000F_A3BEh and wraps around to 0000_0001h.
- The second addition is a simple unsigned add operation with no wraparound.

Related Instructions See the PADDB instruction.
See the PADDW instruction.
See the PADDSB instruction.
See the PADDSW instruction.

PADDSB

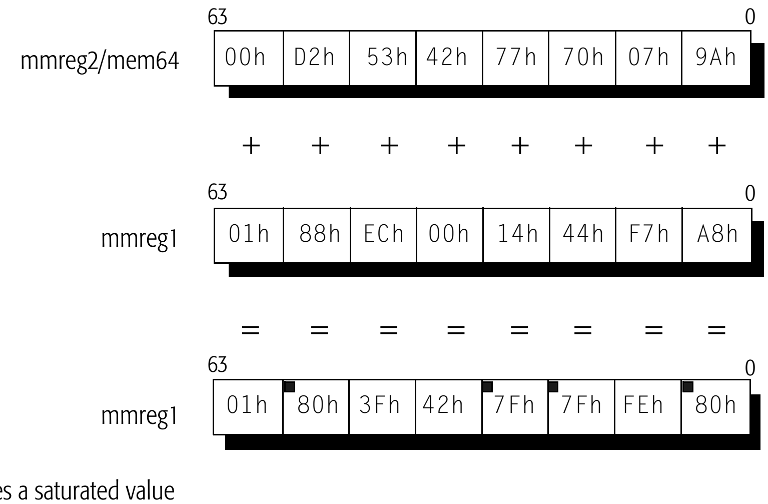
mnemonic	*opcode*	*description*
PADDSB mmreg1, mmreg2/mem64	0F ECh	Add signed packed 8-bit values and saturate

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.
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 PADDSB instruction adds eight signed 8-bit values from the source operand (an MMX register or a 64-bit memory location) to the eight corresponding signed 8-bit values in the destination operand (an MMX register). If the sum of any two 8-bit values is less than –128 (80h), it saturates to –128 (80h). If the sum of any two 8-bit values is greater than 127 (7Fh), it saturates to 127 (7Fh). The eight signed 8-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PADDSB Instruction



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

- The signed 8-bit positive value 00h is added to the signed 8-bit positive value 01h with a signed 8-bit positive result of 01h.
- The signed 8-bit negative value D2h (–46) is added to the signed 8-bit negative value 88h (–120) and saturates to 80h (–128), the largest possible signed 8-bit negative value.
- The signed 8-bit positive value 53h (+83) is added to the signed 8-bit negative value ECh (–20) with a signed 8-bit positive result of 3Fh (+63).
- The signed 8-bit positive value 42h is added to the signed 8-bit positive value 00h with a signed 8-bit positive result of 42h.
- The signed 8-bit positive value 77h (+119) is added to the signed 8-bit positive value 14h (+20) and saturates to 7Fh (+127), the largest possible positive value.
- The signed 8-bit positive value 70h (+112) is added to the signed 8-bit positive value 44h (+68) and saturates to 7Fh (+127), the largest possible positive value.
- The signed 8-bit positive value 07h (+7) is added to the signed 8-bit negative value F7h (–9) with a signed 8-bit negative result of FEh (–2).
- The signed 8-bit negative value 9Ah (–102) is added to the signed 8-bit negative value A8h (–88) and saturates to 80h (–128), the largest possible signed 8-bit negative value.

Related Instructions See the PADDB instruction.

See the PADDD instruction.

See the PADDW instruction.

See the PADDSW instruction.

PADDSW

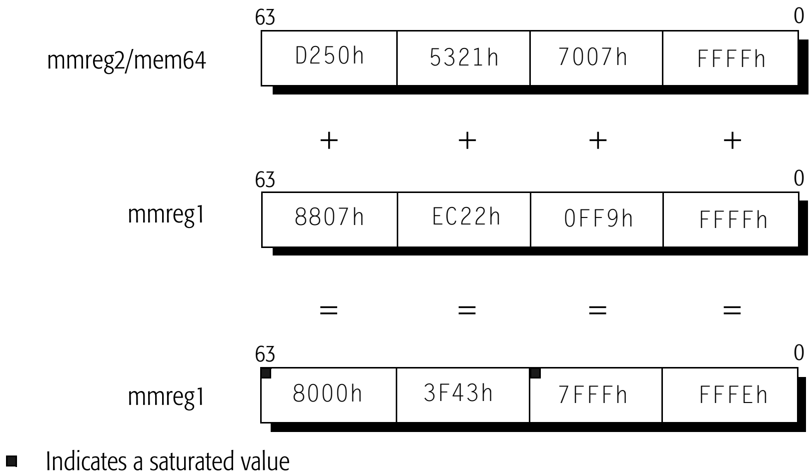
mnemonic	*opcode*	*description*
PADDSW mmreg1, mmreg2/mem64	0F EDh	Add signed packed 16-bit values and saturate

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.
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 PADDSW instruction adds four signed 16-bit values from the source operand (an MMX register or a 64-bit memory location) to the four corresponding signed 16-bit values in the destination operand (an MMX register). If the sum of any two 16-bit values is less than –32768 (8000h), it saturates to –32768 (8000h). If the sum of any two 16-bit values is greater than 32767 (7FFFh), it saturates to 32767 (7FFFh). The four signed 16-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PADDSW Instruction



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

- The signed 16-bit negative value D250h (–11696) is added to the signed 16-bit negative value 8807h (–30713) and saturates to 8000h (–32768), the largest possible signed 16-bit negative value.
- The signed 16-bit positive value 5321h (+21281) is added to the signed 16-bit negative value EC22h (–5086) with a signed 16-bit positive result of 3F43h (+16195).
- The signed 16-bit positive value 7007h (+28679) is added to the signed 16-bit positive value 0FF9h (+4089) and saturates to 7FFFh (+32767), the largest possible positive value.
- The signed 16-bit negative value FFFFh (–1) is added to the signed 16-bit negative value FFFFh (–1) with the negative 16-bit result of FFFEh (–2).

Related Instructions

See the PADDB instruction.

See the PADDD instruction.

See the PADDW instruction.

See the PADDSB instruction.

See the PADDUSB instruction.

See the PADDUSW instruction.

PADDUSB

mnemonic	*opcode*	*description*
PADDUSB mmreg1, mmreg2/mem64	0F DCh	Add unsigned packed 8-bit values and saturate

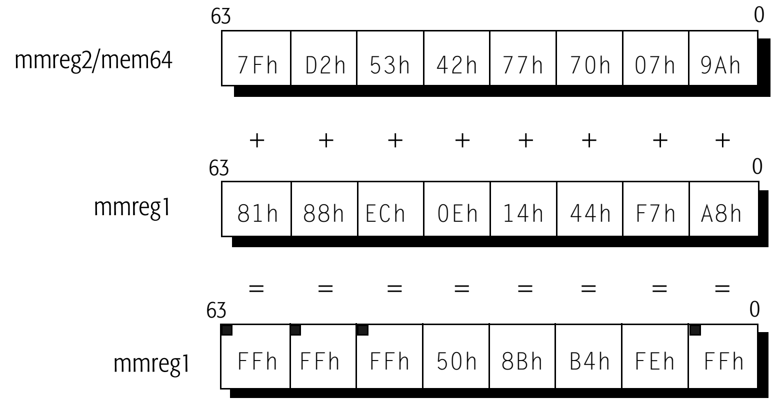
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

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.
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 PADDUSB instruction adds eight unsigned 8-bit values from the source operand (an MMX register or a 64-bit memory location) to the eight corresponding unsigned 8-bit values in the destination operand (an MMX register). The eight unsigned 8-bit results are stored in the MMX register specified as the destination operand.

If the sum of any two unsigned 8-bit values is greater than 255 (FFh), it saturates to 255 (FFh).

Functional Illustration of the PADDUSB Instruction



■ Indicates a saturated value

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

- The sum of 7Fh and 81h is 100h. This value is greater than FFh, so the result saturates to FFh.
- The sum of D2h and 88h is 15Ah. This value is greater than FFh, so the result saturates to FFh.
- The sum of 53h and ECh is 13Fh. This value is greater than FFh, so the result saturates to FFh.
- The sum of 42h and 0Eh is 50h. This value is not greater than FFh, so the result does not saturate.
- The sum of 77h and 14h is 8Bh. This value is not greater than FFh, so the result does not saturate.
- The sum of 70h and 44h is B4h. This value is not greater than FFh, so the result does not saturate.
- The sum of 07h and F7h is FEh. This value is not greater than FFh, so the result does not saturate.
- The sum of 9Ah and A8h is 142h. This value is greater than FFh, so the result saturates to FFh.

Related Instructions

See the PADDB instruction.

See the PADDD instruction.

See the PADDW instruction.

See the PADDSB instruction.

See the PADDSW instruction.

See the PADDUSW instruction.

PADDUSW

mnemonic	*opcode*	*description*
PADDUSW mmreg1, mmreg2/mem64	0F DDh	Add unsigned packed 16-bit values and saturate

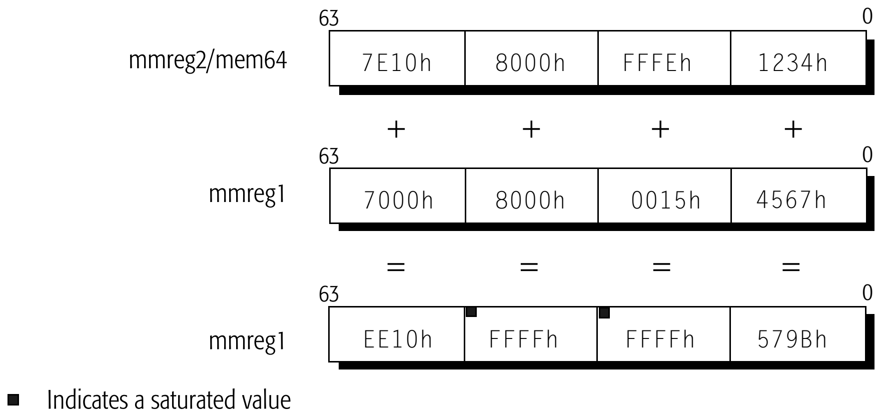
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

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.
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 PADDUSW instruction adds four unsigned 16-bit values from the source operand (an MMX register or a 64-bit memory location) to the four corresponding unsigned 16-bit values in the destination operand (an MMX register). The four unsigned 16-bit results are stored in the MMX register specified as the destination operand.

If the sum of any two unsigned 16-bit values is greater than 65,535 (FFFFh), it saturates to 65,535 (FFFFh).

Functional Illustration of the PADDUSW Instruction



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

- The sum of 7E10h and 7000h is EE10h. This value is not greater than FFFFh, so the result does not saturate.
- The sum of 8000h and 8000h is 10000h. This value is greater than FFFFh, so the result saturates to FFFFh.
- The sum of FFFEh and 0015h is 10013h. This value is greater than FFFFh, so the result saturates to FFFFh.
- The sum of 1234h and 4567h is 579Bh. This value is not greater than FFFFh, so the result does not saturate.

Related Instructions

See the PADDB instruction.

See the PADDD instruction.

See the PADDW instruction.

See the PADDSB instruction.

See the PADDSW instruction.

See the PADDUSB instruction.

PADDW

mnemonic	*opcode*	*description*
PADDW mmreg1, mmreg2/mem64	0F FDh	Add unsigned packed 16-bit values

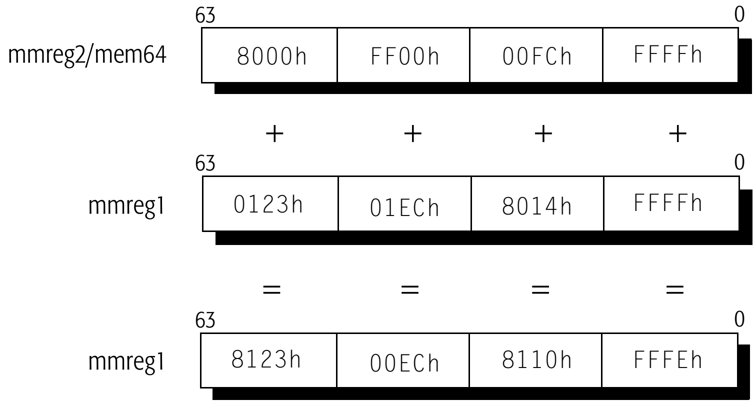
Privilege:	none
Registers Affected:	MMX
Flags Affected:	none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate 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.
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 PADDW instruction adds four unsigned 16-bit values from the source operand (an MMX register or a 64-bit memory location) to the four corresponding unsigned 16-bit values in the destination operand (an MMX register). If any of the four results is greater than the capacity of its 16-bit destination, the value wraps around with no carry into the next location. The four 16-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PADDW Instruction



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

- The value 8000h is added to 0123h with a normal unsigned result of 8123h.
- The value FF00h is added to 01ECh and wraps around to 00ECh.
- The value 00FCh is added to 8014h with a normal signed result of 8110h.
- The value FFFFh is added to FFFFh and wraps around to FFFEh.

Related Instructions

See the PADDB instruction.
See the PADDD instruction.
See the PADDSB instruction.
See the PADDSW instruction.
See the PADDUSB instruction.
See the PADDUSW instruction.

PAND

mnemonic	*opcode*	*description*
PAND mmreg1, mmreg2/mem64	0F DBh	AND 64-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 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.
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 PAND instruction operates on the 64-bit source and destination operands to complete a bitwise logical AND. The results are stored in the destination operand. If the corresponding bits in the source and destination operands both equal 1, the resulting bit is 1 in the destination. If either bit in the source or destination operands equals 0, the resulting bit is 0 in the destination.

The PAND instruction can be used to extract operands from packed fields based on the masks that are produced by the compare instructions—PCMPEQ and PCMPGT. This technique can eliminate branch prediction overhead in MMX routines.

Functional Illustration of the PAND Instruction

mmreg1

63 … 48	47 … 32	31 … 16	15 … 0
1010_1111_0000_1101	0000_1111_0000_1111	1100_0001_0011_0001	1000_1100_1101_0011

Logical AND	Logical AND	Logical AND	Logical AND

mmreg2/mem64

63 … 48	47 … 32	31 … 16	15 … 0
0101_1100_1100_0011	1100_1101_0100_1110	1011_0001_0011_1001	0110_0011_0101_1001

Result

mmreg1

63 … 48	47 … 32	31 … 16	15 … 0
0000_1100_0000_0001	0000_1101_0000_1110	1000_0001_0011_0001	0000_0000_0101_0001

Related Instructions

See the PANDN instruction.
See the POR instruction.
See the PXOR instruction.

PANDN

mnemonic	*opcode*	*description*
PANDN mmreg1, mmreg2/mem64	0F DFh	Invert a 64-bit value, then AND the inverted value and a 64-bit value in memory or an MMX register

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

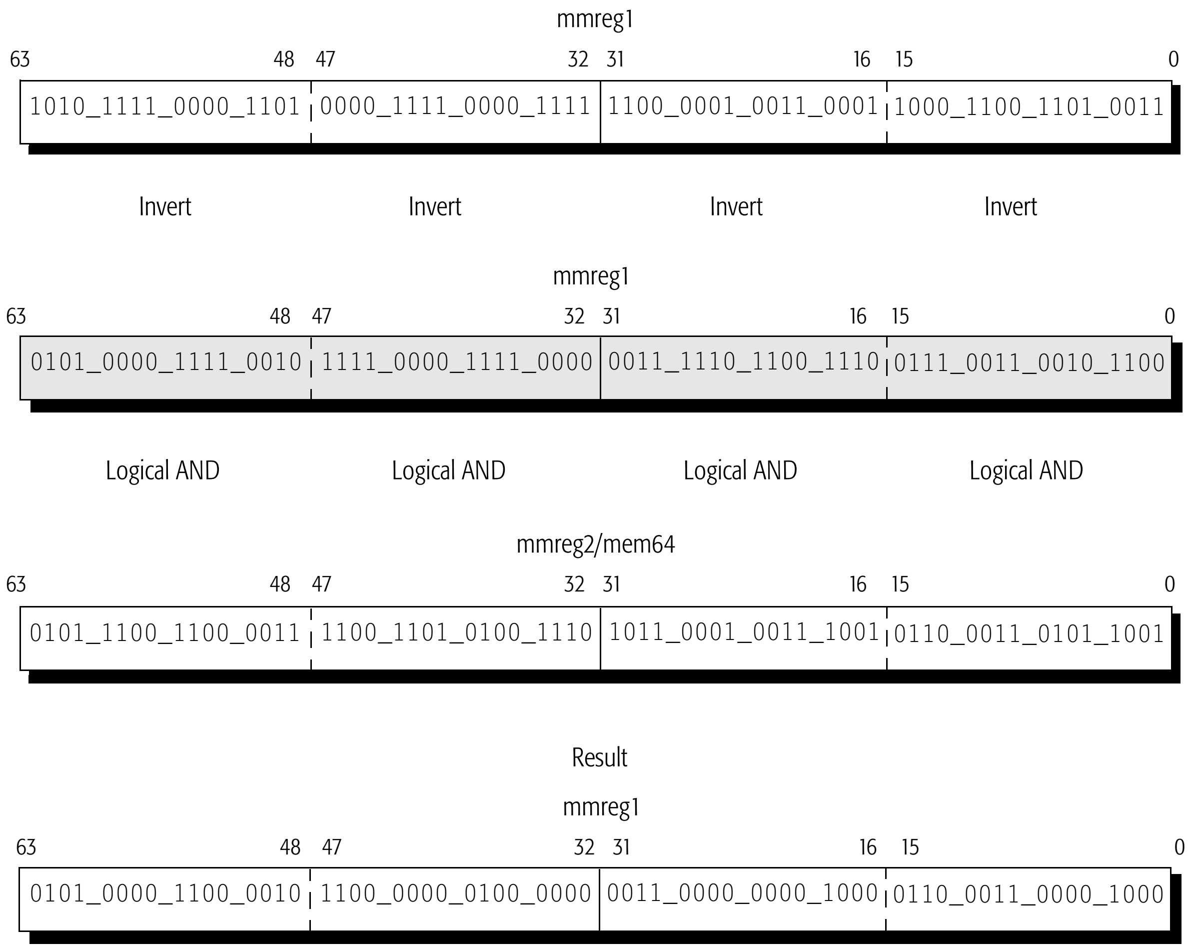
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (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 PANDN instruction first operates on the 64-bit destination operand (an MMX register) to complete a bitwise logical NOT, inverting each bit. This operation changes 1 bits to 0 bits and 0 bits to 1 bits, storing the results in the destination operand. The inverted 64-bit destination operand is then logically AND'd with the 64-bit source operand (an MMX register or a 64-bit memory operand) to complete the PANDN operation.

If corresponding bits in the source operand and the inverted destination operand are both 1, the resulting bit is 1 in the destination. If either bit in the source operand or the inverted destination operand is 0, the resulting bit is 0 in the destination.

The PANDN instruction can be used to extract alternate operands from packed fields based on the inverse of the masks that are produced by the compare instructions—PCMPEQ and PCMPGT. This technique can eliminate branch prediction overhead in MMX routines.

Functional Illustration of the PANDN Instruction



Related Instructions See the PAND instruction.
See the POR instruction.
See the PXOR instruction.

PCMPEQB

mnemonic	*opcode*	*description*
PCMPEQB mmreg1, mmreg2/mem64	0F 74h	Compare packed 8-bit values for equality

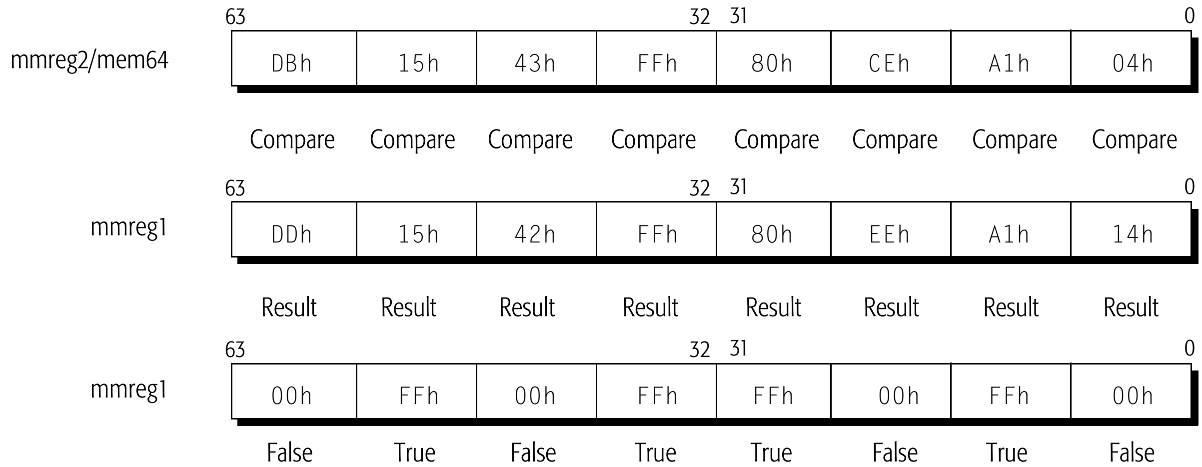
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

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.
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 PCMPEQB instruction operates on 8-bit data values. The instruction compares two 8-bit values to determine if they are equal.

If the corresponding bits in the two operands are equal, all the bits in that 8 bits of the destination operand are set to 1. If any of the corresponding bits in the two operands are not equal, all the bits in that 8 bits of the destination operand are set to 0.

Functional Illustration of the PCMPEQB Instruction



Related Instructions

See the PCMPEQD instruction.

See the PCMPEQW instruction.

See the PCMPGTB instruction.

See the PCMPGTD instruction.

See the PCMPGTW instruction.

PCMPEQD

mnemonic	*opcode*	*description*
PCMPEQD mmreg1, mmreg2/mem64	0F 76h	Compare packed 32-bit values for equality

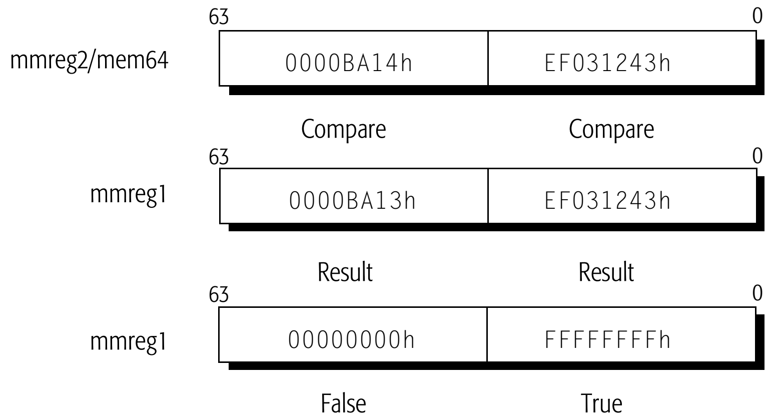
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

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.
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 PCMPEQD instruction operates on 32-bit data values. The instruction compares two 32-bit values to determine if they are equal.

If the corresponding bits in the two operands are equal, all the bits in that 32 bits of the destination operand are set to 1. If any of the corresponding bits in the two operands are not equal, all the bits in that 32 bits of the destination operand are set to 0.

Functional Illustration of the PCMPEQD Instruction



Related Instructions

See the PCMPEQB instruction.

See the PCMPEQW instruction.

See the PCMPGTB instruction.

See the PCMPGTD instruction.

See the PCMPGTW instruction.

PCMPEQW

mnemonic	*opcode*	*description*
PCMPEQW mmreg1, mmreg2/mem64	0F 75h	Compare packed 16-bit values for equality

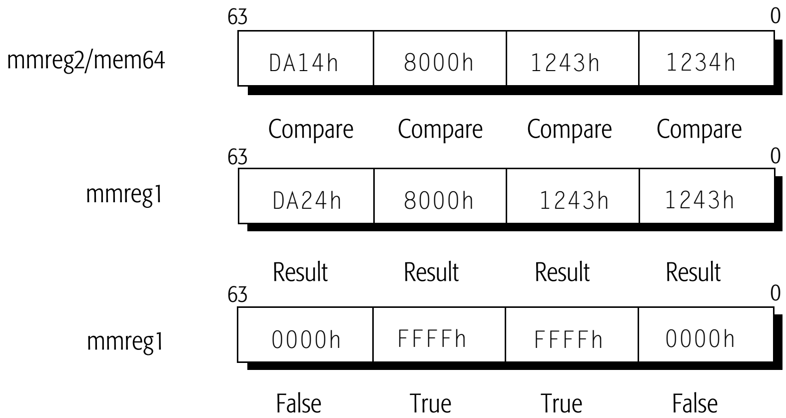
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated

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.
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 PCMPEQW instruction operates on 16-bit data values. The instruction compares two 16-bit values to determine if they are equal.

If the corresponding bits in the two operands are equal, all the bits in that 16 bits of the destination operand are set to 1. If any of the corresponding bits in the two operands are not equal, all the bits in that 16 bits of the destination operand are set to 0.

Functional Illustration of the PCMPEQW Instruction



Related Instructions

See the PCMPEQB instruction.

See the PCMPEQD instruction.

See the PCMPGTB instruction.

See the PCMPGTD instruction.

See the PCMPGTW instruction.

PCMPGTB

mnemonic	*opcode*	*description*
PCMPGTB mmreg1, mmreg2/mem64	0F 64h	Compare signed packed 8-bit values for magnitude

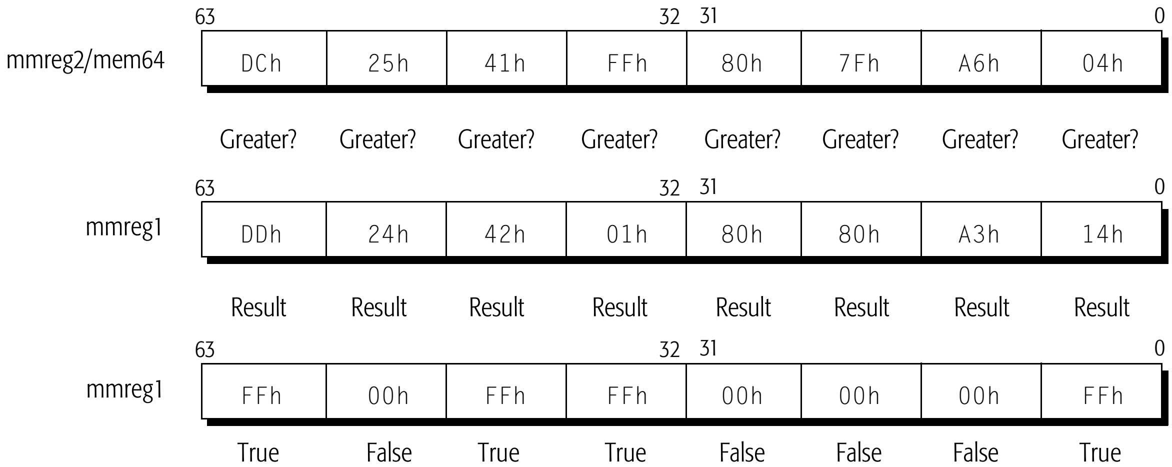
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

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.
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 PCMPGTB instruction operates on signed 8-bit data values. The instruction compares two signed 8-bit values to determine if the value in the destination operand is greater than the corresponding signed 8-bit data value in the source operand.

If the value in the destination operand is greater than the value in the source operand, all the bits in that 8 bits of the destination operand are set to 1. If the value in the destination operand is equal to or less than the value in the source operand, all the bits in that 8 bits of the destination operand are set to 0.

Functional Illustration of the PCMPGTB Instruction



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

- The negative value DDh (–35) is greater than the negative value DCh (–36), so the result is true (FFh).
- The positive value 24h (+36) is not greater than the positive value 25h (+37), so the result is false (00h).
- The positive value 42h (+66) is greater than the positive value 41h (+65), so the result is true (FFh).
- The positive value 01h (+1) is greater than the negative value FFh (–1), so the result is true (FFh).
- The negative value 80h (–128) is not greater than the negative value 80h (–128), so the result is false (00h).
- The negative value 80h (–128) is not greater than the positive value 7Fh (+127), so the result is false (00h).
- The negative value A3h (–93) is not greater than the negative value A6h (–90), so the result is false (00h).
- The positive value 14h (+20) is greater than the positive value 04h (+4), so the result is true (FFh).

Related Instructions

See the PCMPEQB instruction.

See the PCMPEQD instruction.

See the PCMPEQW instruction.

See the PCMPGTD instruction.

See the PCMPGTW instruction.

PCMPGTD

mnemonic	*opcode*	*description*
PCMPGTD mmreg1, mmreg2/mem64	0F 66h	Compare signed packed 32-bit values for magnitude

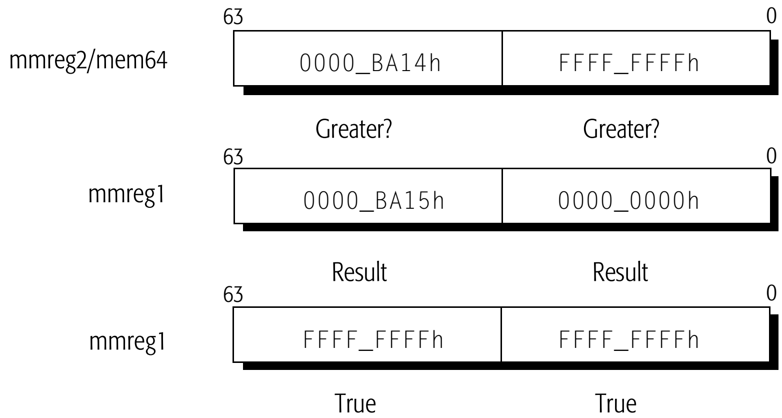
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated

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.
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 PCMPGTB instruction operates on signed 32-bit data values. The instruction compares two signed 32-bit values to determine if the value in the destination operand is greater than the corresponding signed 32-bit data value in the source operand.

If the value in the destination operand is greater than the value in the source operand, all the bits in that 32 bits of the destination operand are set to 1. If the value in the destination operand is equal to or less than the value in the source operand, all the bits in that 32 bits of the destination operand are set to 0.

Functional Illustration of the PCMPGTD Instruction



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

- The positive value 0000_BA15h (+47637) is greater than the positive value 0000_BA14h (+47636), so the result is true (FFFF_FFFFh).
- The positive value 0000_0001h (+1) is greater than the negative value FFFF_FFFFh (–1), so the result is true (FFFF_FFFFh).

Related Instructions See the PCMPEQB instruction.

See the PCMPEQD instruction.

See the PCMPEQW instruction.

See the PCMPGTB instruction.

See the PCMPGTW instruction.

PCMPGTW

mnemonic	*opcode*	*description*
PCMPGTW mmreg1, mmreg2/mem64	0F 65h	Compare signed packed 16-bit values for magnitude

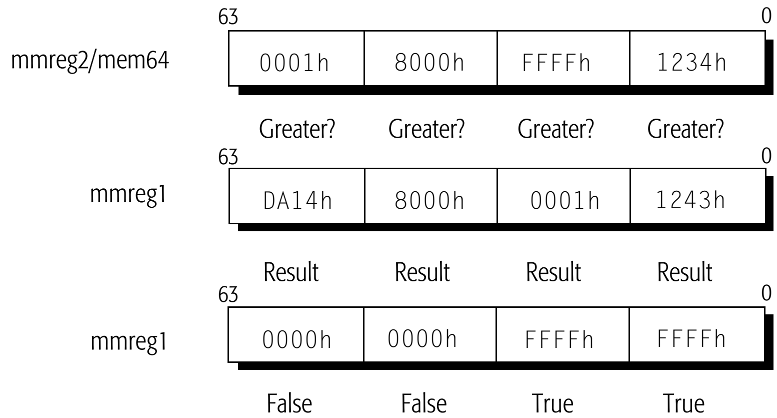
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

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.
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 PCMPGTW instruction operates on signed 16-bit data values. The instruction compares two signed 16-bit values to determine if the value in the destination operand is greater than the corresponding signed 16-bit data value in the source operand.

If the value in the destination operand is greater than the value in the source operand, all the bits in that 16 bits of the destination operand are set to 1. If the value in the destination operand is equal to or less than the value in the source operand, all the bits in that 16 bits of the destination operand are set to 0.

Functional Illustration of the PCMPGTW Instruction



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

- The negative value DA14h (–9708) is not greater than the positive value 0001h (+1), so the result is false (0000h).
- The negative value 8000h (–32768) is not greater than the negative value 8000h (–32768), so the result is false (0000h).
- The positive value 0001h (+1) is greater than the negative value FFFFh (–1), so the result is true (FFFFh).
- The positive value 1243h (+4675) is greater than the positive value 1234h (+4660), so the result is true (FFFFh).

Related Instructions See the PCMPEQB instruction.

See the PCMPEQD instruction.

See the PCMPEQW instruction.

See the PCMPGTB instruction.

See the PCMPGTD instruction.

PMADDWD

mnemonic	*opcode*	*description*
PMADDWD mmreg1, mmreg2/mem64	0F F5h	Multiply signed packed 16-bit values and add the 32-bit results

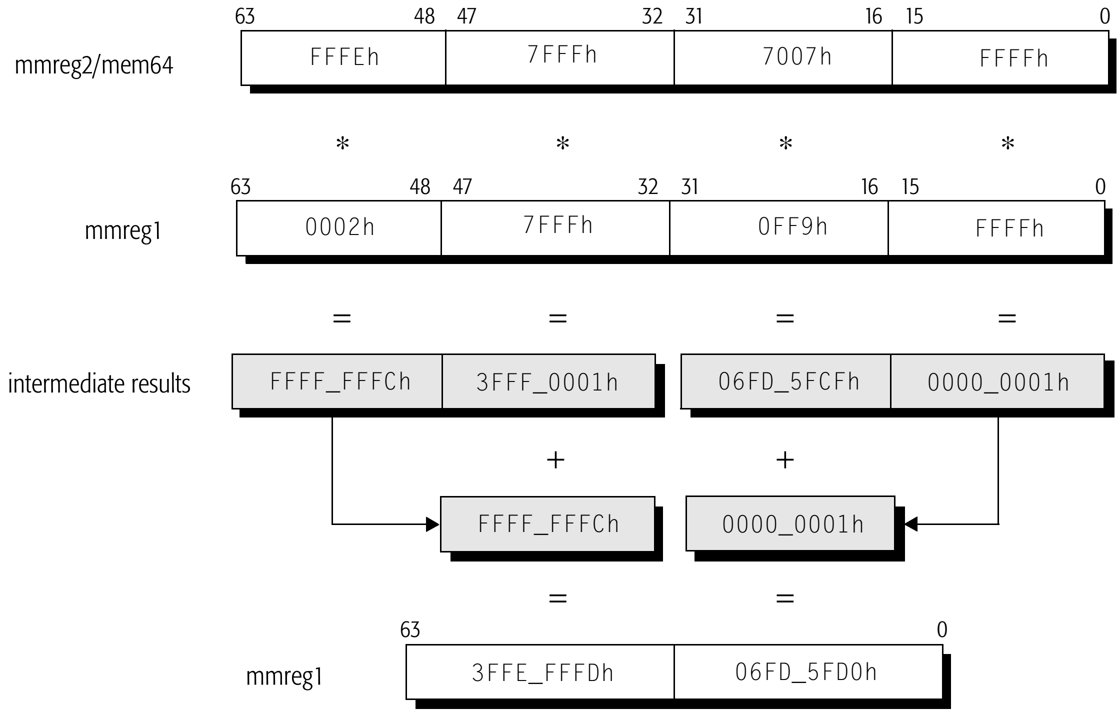
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

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.
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 PMADDWD instruction multiplies signed 16-bit values from the source operand (an MMX register or a 64-bit memory location) by the corresponding signed 16-bit values in the destination operand (an MMX register), adds the resulting 32-bit values from the left and right halves of the 64-bit work space, and stores the 32-bit sums in the MMX destination register.

Note: *If all four of the 16-bit operands are 8000h, the result wraps around to 8000_0000h because the maximum negative 16-bit value of 8000h multiplied by itself equals 4000_0000h, and 4000_0000h added to 4000_0000h equals 8000_0000h. The result of multiplying two negative numbers should be a positive number, but 8000_0000h is the maximum possible 32-bit negative number rather than a positive number. This is the only instance of wraparound that can occur as a result of the PMADDWD instruction.*

Functional Illustration of the PMADDWD Instruction



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

- The signed 16-bit negative value FFFEh (–2) is multiplied by the signed 16-bit positive value 0002h to produce a signed 32-bit negative intermediate result of FFFF_FFFCh (–4).
- The signed 16-bit positive value 7FFFh is multiplied by the signed 16-bit positive value 7FFFh to produce a signed 32-bit positive intermediate result of 3FFF_0001h.
- The two 32-bit intermediate results are added together to produce the final signed 32-bit positive result of 3FFE_FFFDh.
- The signed 16-bit positive value 7007h is multiplied by the signed 16-bit positive value 0FF9h to produce a signed 32-bit intermediate result of 06FD_5FCFh.
- The signed 16-bit negative value FFFFh (–1) is multiplied by the signed 16-bit negative value FFFFh (–1) to produce a signed 32-bit positive intermediate result of 0000_0001h.
- The two 32-bit intermediate results are added together to produce the final signed 32-bit positive result of 06FD_5FD0h.

Related Instructions See the PMULHW instruction.

See the PMULLW instruction.

PMULHW

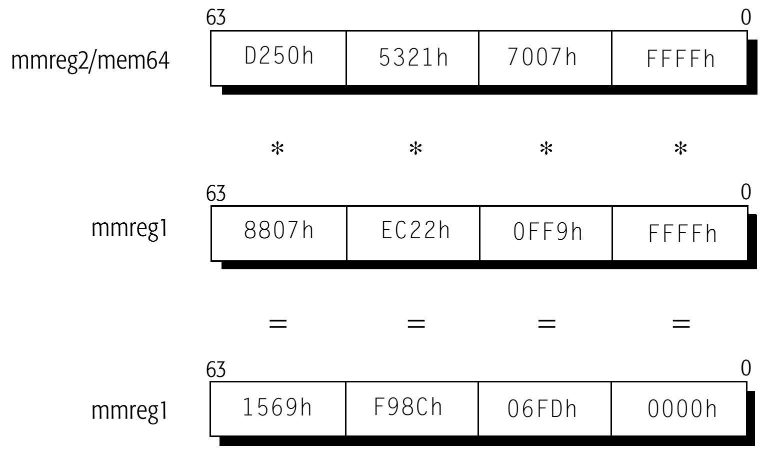
mnemonic	*opcode*	*description*
PMULHW mmreg1, mmreg2/mem64	0F E5h	Multiply signed packed 16-bit values and store the high 16 bits

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

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate 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.
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 PMULHW instruction multiplies four signed 16-bit values from the source operand (an MMX register or a 64-bit memory location) by the four corresponding signed 16-bit values in the destination operand (an MMX register) and then stores the high-order 16 bits of the result (including the sign bit) in the destination operand.

Functional Illustration of the PMULHW Instruction



The following list explains the functional illustration of the PMULHW 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. 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). 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 0FF9h to produce the signed 32-bit positive result of 06FD_5FCFh. The signed high-order 16-bits of the result 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. The signed high-order 16-bits of the result are stored in the destination operand.

Related Instructions See the PMADDWD instruction.

See the PMULLW instruction.

See the PUNPCKHWD instruction.

See the PUNPCKLWD instruction.

PMULLW

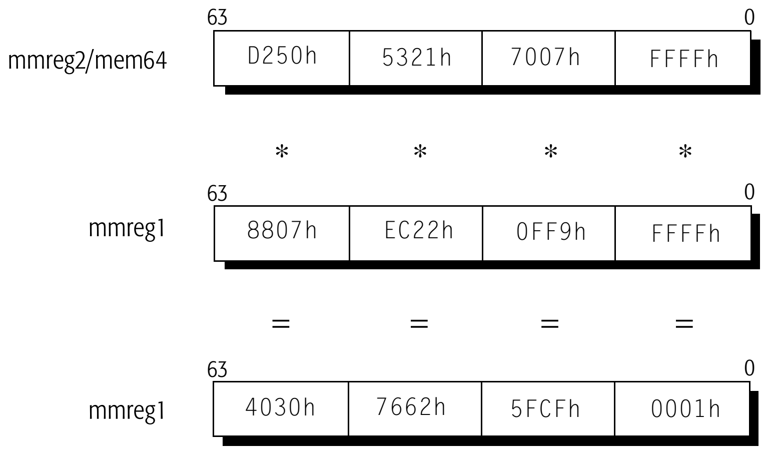
mnemonic	*opcode*	*description*
PMULLW mmreg1, mmreg2/mem64	0F D5h	Multiply signed packed 16-bit values and store the low 16 bits

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

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate 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.
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 PMULLW instruction multiplies four signed 16-bit values from the source operand (an MMX register or a 64-bit memory location) by the four corresponding signed 16-bit values in the destination operand (an MMX register) and then stores the low-order 16 bits of the result (unsigned) in the destination operand.

Functional Illustration of the PMULLW Instruction



The following list explains the functional illustration of the PMULLW 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. The unsigned low-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). The unsigned low-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 0FF9h to produce the signed 32-bit positive result of 06FD_5FCFh. The unsigned low-order 16-bits of the result 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. The unsigned low-order 16-bits of the result are stored in the destination operand.

Related Instructions

See the PMADDWD instruction.

See the PMULHW instruction.

See the PUNPCKHWD instruction.

See the PUNPCKLWD instruction.

POR

mnemonic	*opcode*	*description*
POR mmreg1, mmreg2/mem64	0F EBh	OR 64-bit values

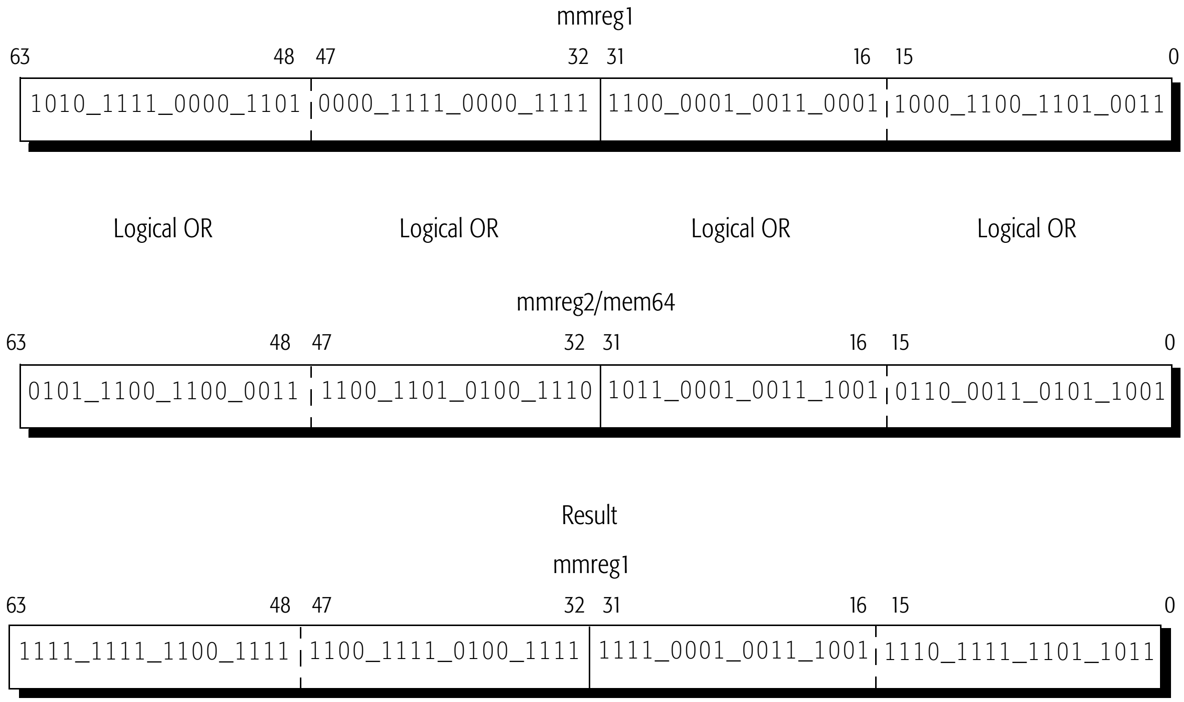
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate 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.
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 POR instruction logically ORs the 64 bits of the source operand (an MMX register or a 64-bit memory location) with the 64 bits of the destination operand (an MMX register) and stores the result in the destination register.

A logical OR produces a 1 bit if either or both input bits is a 1. If both input bits are 0, a logical OR produces a 0 bit.

Functional Illustration of the POR Instruction



In the functional illustration of the POR instruction, the 64-bit source value is logically OR'd to the 64-bit destination value, and the result is stored in the destination register.

Related Instructions

See the PAND instruction.

See the PANDN instruction.

See the PXOR instruction.

PSLLD

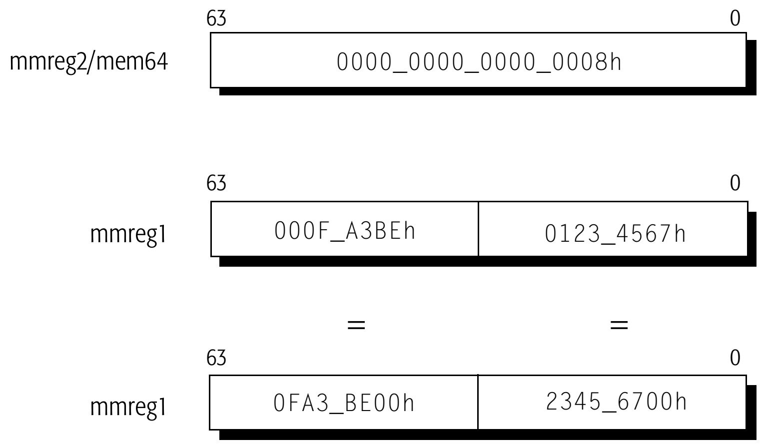
mnemonic	*opcode*	*description*
PSLLD mmreg1, mmreg2/mem64	0F F2h	Shift left logical packed 32-bit values in mmreg1 the number of positions in mmreg2/mem64 with zero fill from the right
PSLLD mmreg1, imm8	0F 72h /6	Shift left logical packed 32-bit values in mmreg1 the number of positions in imm8 with zero fill from the right

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.
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 PSLLD instruction shifts the two 32-bit operands in the destination operand (an MMX register) to the left by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted values are zero filled from the right. The two 32-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PSLLD Instruction



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

- The value 0000_0000_0000_0008h in mmreg2/mem64 indicates a shift of 8 bit positions to the left.
- The 32-bit value 000F_A3BEh in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as 0FA3_BE00h.
- The 32-bit value 0123_4567h in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as 2345_6700h.

Related Instructions

See the PSLLQ instruction.

See the PSLLW instruction.

See the PSRAD instruction.

See the PSRAW instruction.

See the PSRLD instruction.

See the PSRLQ instruction.

See the PSRLW instruction.

PSLLQ

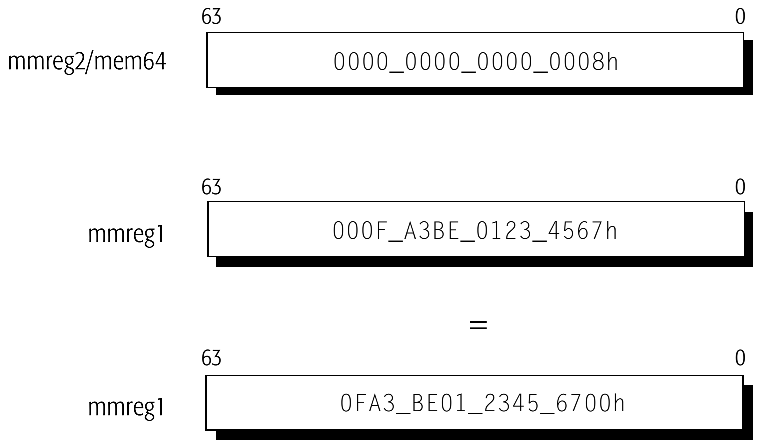
mnemonic	*opcode*	*description*
PSLLQ mmreg1, mmreg2/mem64	0F F3h	Shift left logical 64-bit values in mmreg1 the number of positions in mmreg2/mem64 with zero fill from the right
PSLLQ mmreg1, imm8	0F 73h /6	Shift left logical 64-bit values in mmreg1 the number of positions in imm8 with zero fill from the right

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.
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 PSLLQ instruction shifts the 64-bit operand in the destination operand (an MMX register) to the left by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted value is zero filled from the right. The 64-bit result is stored in the MMX register specified as the destination operand.

Functional Illustration of the PSLLQ Instruction



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

- The value 0000_0000_0000_0008h in mmreg2/mem64 indicates a shift of 8 bit positions to the left.
- The 64-bit value 000F_A3BE_0123_4567h in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as 0FA3_BE01_2345_6700h.

Related Instructions

See the PSLLD instruction.

See the PSLLW instruction.

See the PSRAD instruction.

See the PSRAW instruction.

See the PSRLD instruction.

See the PSRLQ instruction.

See the PSRLW instruction.

PSLLW

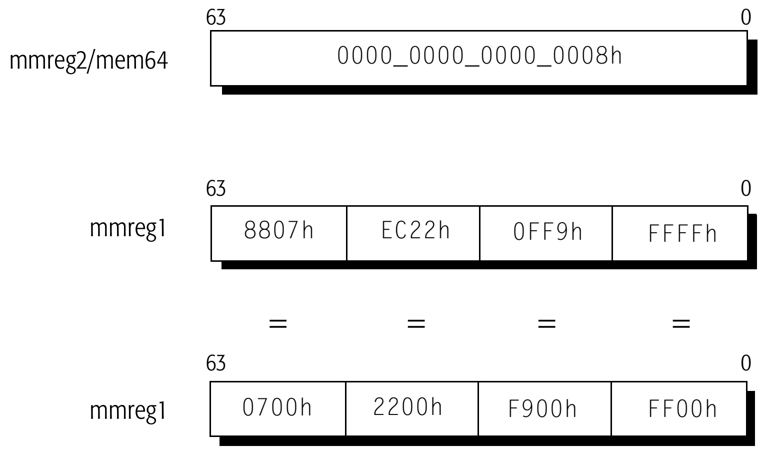
mnemonic	*opcode*	*description*
PSLLW mmreg1, mmreg2/mem64	0F F1h	Shift left logical packed 16-bit values in mmreg1 the number of positions in mmreg2/mem64 with zero fill from the right
PSLLW mmreg1, imm8	0F 71h /6	Shift left logical packed 16-bit values in mmreg1 the number of positions in imm8 with zero fill from the right

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.
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 PSLLW instruction shifts the four 16-bit operands in the destination operand (an MMX register) to the left by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted values are zero filled from the right. The four 16-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PSLLW Instruction



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

- The value 0000_0000_0000_0008h in mmreg2/mem64 indicates a shift of 8 bit positions to the left.
- The 16-bit value 8807h in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as 0700h.
- The 16-bit value EC22h in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as 2200h.
- The 16-bit value 0FF9h in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as F900h.
- The 16-bit value FFFFh in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as FF00h.

Related Instructions

See the PSLLD instruction.
See the PSLLQ instruction.
See the PSRAD instruction.
See the PSRAW instruction.
See the PSRLD instruction.
See the PSRLQ instruction.
See the PSRLW instruction.

PSRAD

mnemonic	*opcode*	*description*
PSRAD mmreg1, mmreg2/mem64	0F E2h	Shift right arithmetic packed signed 32-bit values in mmreg1 the number of positions in mmreg2/mem64 with sign fill from the left
PSRAD mmreg1, imm8	0F 72h /4	Shift right arithmetic packed signed 32-bit values in mmreg1 the number of positions in imm8 with sign fill from the left

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.
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 PSRAD instruction shifts the two signed 32-bit operands in the destination operand (an MMX register) to the right by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted values are sign filled from the left. The two signed 32-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PSRAD Instruction

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

- The value 0000_0000_0000_0010h in mmreg2/mem64 indicates a shift of 16 bit positions to the right.
- The 32-bit negative value FFF0_0000h in mmreg1 is shifted 16 bit positions to the right with sign fill from the left and stored in mmreg1 as FFFF_FFF0h.
- The 32-bit positive value 0123_0000h in mmreg1 is shifted 16 bit positions to the right with sign fill from the left and stored in mmreg1 as 0000_0123h.

Related Instructions

See the PSLLD instruction.
See the PSLLQ instruction.
See the PSLLW instruction.
See the PSRAW instruction.
See the PSRLD instruction.
See the PSRLQ instruction.
See the PSRLW instruction.
See the PUNPCKHWD instruction.
See the PUNPCKLWD instruction.

PSRAW

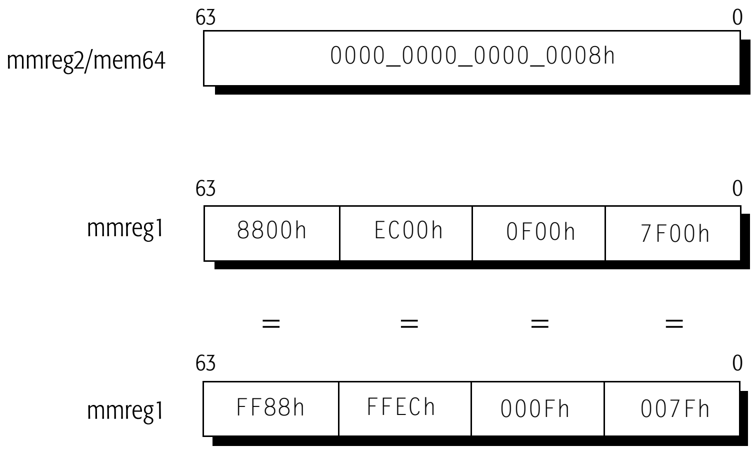
mnemonic	*opcode*	*description*
PSRAW mmreg1, mmreg2/mem64	0F E1h	Shift right arithmetic packed signed 16-bit values in mmreg1 the number of positions in mmreg2/mem64 with sign fill from the left
PSRAW mmreg1, imm8	0F 71h /4	Shift right arithmetic packed signed 16-bit values in mmreg1 the number of positions in imm8 with sign fill from the left

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.
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 PSRAW instruction shifts the four signed 16-bit operands in the destination operand (an MMX register) to the right by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted values are sign filled from the left. The four signed 16-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PSRAW Instruction



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

- The value 0000_0000_0000_0008h in mmreg2/mem64 indicates a shift of 8 bit positions to the right.
- The 16-bit negative value 8800h in mmreg1 is shifted 8 bit positions to the right with sign fill from the left and stored in mmreg1 as FF88h.
- The 16-bit negative value EC00h in mmreg1 is shifted 8 bit positions to the right with sign fill from the left and stored in mmreg1 as FFECh.
- The 16-bit positive value 0F00h in mmreg1 is shifted 8 bit positions to the right with sign fill from the left and stored in mmreg1 as 000Fh.
- The 16-bit positive value 7F00h in mmreg1 is shifted 8 bit positions to the right with sign fill from the left and stored in mmreg1 as 007Fh.

Related Instructions

See the PSLLD instruction.

See the PSLLQ instruction.

See the PSLLW instruction.

See the PSRAD instruction.

See the PSRLD instruction.

See the PSRLQ instruction.

See the PSRLW instruction.

See the PUNPCKHBW instruction.

See the PUNPCKLBW instruction.

PSRLD

mnemonic	*opcode*	*description*
PSRLD mmreg1, mmreg2/mem64	0F D2h	Shift right logical packed 32-bit values in mmreg1 the number of positions in mmreg2/mem64 with zero fill from the left
PSRLD mmreg1, imm8	0F 72h /2	Shift right logical packed 32-bit values in mmreg1 the number of positions in imm8 with zero fill from the left

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.
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 PSRLD instruction shifts the two 32-bit operands in the destination operand (an MMX register) to the right by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted values are zero filled from the left. The two 32-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PSRLD Instruction

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

- The value 0000_0000_0000_0010h in mmreg2/mem64 indicates a shift of 16 bit positions to the right.
- The 32-bit value FFF0_0000h in mmreg1 is shifted 16 bit positions to the right and stored in mmreg1 as 0000_FFF0h
- The 32-bit value 0123_4567h in mmreg1 is shifted 16 bit positions to the right and stored in mmreg1 as 0000_0123h.

Related Instructions

See the PSLLD instruction.

See the PSLLQ instruction.

See the PSLLW instruction.

See the PSRAD instruction.

See the PSRAW instruction.

See the PSRLQ instruction.

See the PSRLW instruction.

PSRLQ

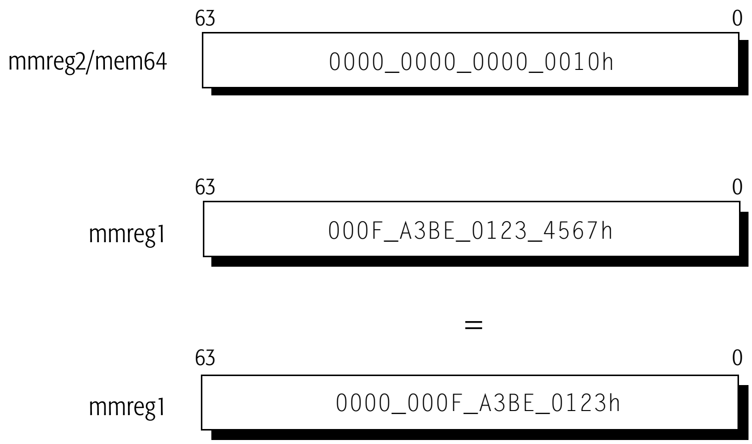
mnemonic	*opcode*	*description*
PSRLQ mmreg1, mmreg2/mem64	0F D3h	Shift right logical 64-bit values in mmreg1 the number of positions in mmreg2/mem64 with zero fill from the left
PSRLQ mmreg1, imm8	0F 73h /2	Shift right logical 64-bit values in mmreg1 the number of positions in imm8 with zero fill from the left

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.
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 PSRLQ instruction shifts the 64-bit operand in the destination operand (an MMX register) to the right by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted value is zero filled from the left. The result is stored in the MMX register specified as the destination operand.

Functional Illustration of the PSRLQ Instruction



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

- The value 0000_0000_0000_0010h in mmreg2/mem64 indicates a shift of 16 bit positions to the right.
- The 64-bit value 000F_A3BE_0123_4567h in mmreg1 is shifted 16 bit positions to the right and stored in mmreg1 as 0000_000F_A3BE_0123h.

Related Instructions

See the PSLLD instruction.

See the PSLLQ instruction.

See the PSLLW instruction.

See the PSRAD instruction.

See the PSRAW instruction.

See the PSRLD instruction.

See the PSRLW instruction.

PSRLW

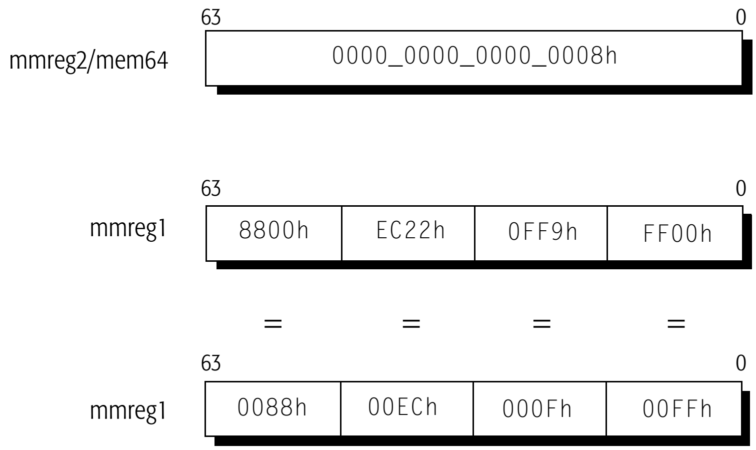
mnemonic	*opcode*	*description*
PSRLW mmreg1, mmreg2/mem64	0F D1h	Shift right logical packed 16-bit values in mmreg1 the number of positions in mmreg2/mem64 with zero fill from the left
PSRLW mmreg1, imm8	0F 71h /2	Shift right logical packed 16-bit values in mmreg1 the number of positions in imm8 with zero fill from the left

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.
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 PSRLW instruction shifts the four 16-bit operands in the destination operand (an MMX register) to the right by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted values are zero filled from the left. The four 16-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PSRLW Instruction



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

- The value 0000_0000_0000_0008h in mmreg2/mem64 indicates a shift of 8 bit positions to the right.
- The 16-bit value 8800h in mmreg1 is shifted 8 bit positions to the right and stored in mmreg1 as 0088h.
- The 16-bit value EC22h in mmreg1 is shifted 8 bit positions to the right and stored in mmreg1 as 00ECh.
- The 16-bit value 0FF9h in mmreg1 is shifted 8 bit positions to the right and stored in mmreg1 as 000Fh.
- The 16-bit value FF00h in mmreg1 is shifted 8 bit positions to the right and stored in mmreg1 as 00FFh.

Related Instructions

See the PSLLD instruction.

See the PSLLQ instruction.

See the PSLLW instruction.

See the PSRAD instruction.

See the PSRAW instruction.

See the PSRLD instruction.

See the PSRLQ instruction.

PSUBB

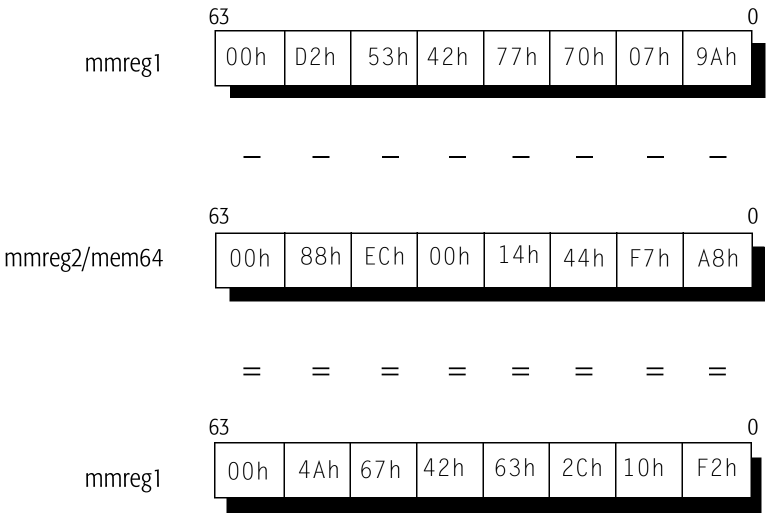
mnemonic	*opcode*	*description*
PSUBB mmreg1, mmreg2/mem64	0F F8h	Subtract unsigned packed 8-bit values with wraparound

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.
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 PSUBB instruction subtracts eight unsigned 8-bit values in the source operand (an MMX register or a 64-bit memory location) from the eight corresponding unsigned 8-bit values in the destination operand (an MMX register). If the source operand is larger than the destination operand, the result wraps around.

Functional Illustration of the PSUBB Instruction



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

- The unsigned 8-bit value ECh is subtracted from the unsigned 8-bit value 53h and wraps around to 67h.
- The unsigned 8-bit value F7h is subtracted from the unsigned 8-bit value 07h and wraps around to 10h.
- The unsigned 8-bit value A8h is subtracted from the unsigned 8-bit value 9Ah and wraps around to F2h.
- All the remaining operations are simple subtraction with no wraparound.

Related Instructions

See the PSUBD instruction.

See the PSUBW instruction.

See the PSUBSB instruction.

See the PSUBSW instruction.

See the PSUBUSB instruction.

See the PSUBUSW instruction.

PSUBD

mnemonic	*opcode*	*description*
PSUBD mmreg1, mmreg2/mem64	0F FAh	Subtract unsigned packed 32-bit values with wraparound

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.
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 PSUBD instruction subtracts two unsigned 32-bit values in the source operand (an MMX register or a 64-bit memory location) from the two corresponding unsigned 32-bit values in the destination operand (an MMX register). If the source operand is larger than the destination operand, the result wraps around.

Functional Illustration of the PSUBD Instruction

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

- The unsigned 32-bit value 8000_0000h is subtracted from the unsigned 32-bit value 0123_4567h and wraps around to 8123_4567h.
- The remaining operation is a simple subtraction with no wraparound.

Related Instructions See the PSUBB instruction.
See the PSUBW instruction.
See the PSUBSB instruction.
See the PSUBSW instruction.
See the PSUBUSB instruction.
See the PSUBUSW instruction.

PSUBSB

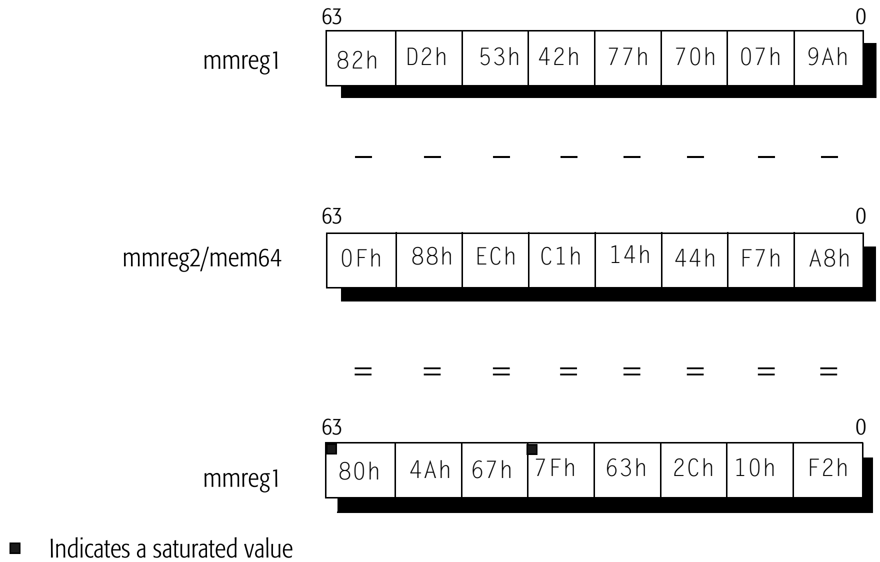
mnemonic	*opcode*	*description*
PSUBSB mmreg1, mmreg2/mem64	0F E8h	Subtract signed packed 8-bit values and saturate

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.
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 PSUBSB instruction subtracts eight signed 8-bit values in the source operand (an MMX register or a 64-bit memory location) from the eight corresponding signed 8-bit values in the destination operand (an MMX register). If a result is less than –128 (80h), it saturates to –128 (80h). If a result is greater than 127 (7Fh), it saturates to 127 (7Fh). The eight signed 8-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PSUBSB Instruction



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

- The signed 8-bit positive value 0Fh is subtracted from the signed 8-bit negative value 82h, and the result saturates to 80h because it is less than 80h, the smallest possible signed 8-bit value.
- The signed 8-bit negative value C1h is subtracted from the signed 8-bit positive value 42h, and the result saturates to 7Fh because it is greater than 7Fh, the largest possible signed 8-bit value.
- All the remaining operations are simple signed subtraction with no saturation.

Related Instructions See the PSUBB instruction.

See the PSUBD instruction.

See the PSUBW instruction.

See the PSUBSW instruction.

See the PSUBUSB instruction.

See the PSUBUSW instruction.

PSUBSW

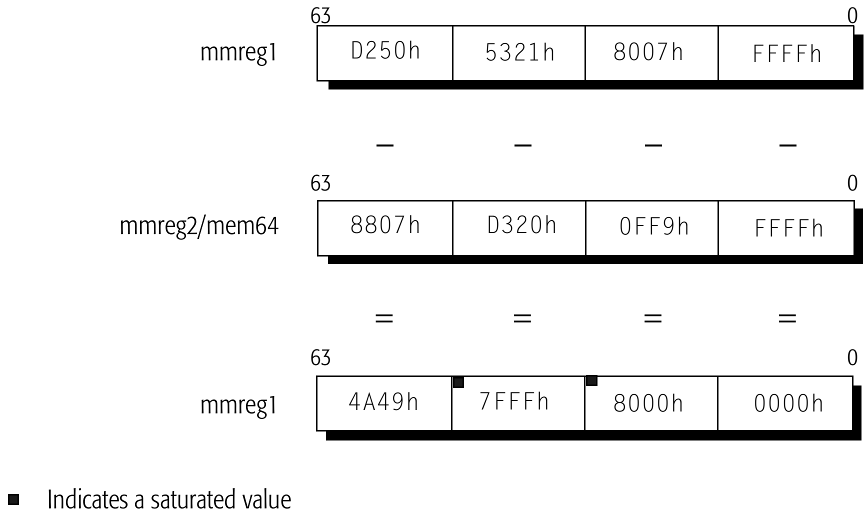
mnemonic	*opcode*	*description*
PSUBSW mmreg1, mmreg2/mem64	0F E9h	Subtract signed packed 16-bit values and saturate

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.
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 PSUBSW instruction subtracts four signed 16-bit values in the source operand (an MMX register or a 64-bit memory location) from the four corresponding signed 16-bit values in the destination operand (an MMX register). If a result is less than –32768 (8000h), it saturates to –32768 (8000h). If a result is greater than 32767 (7FFFh), it saturates to 32767 (7FFFh). The four signed 16-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PSUBSW Instruction



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

- The signed 16-bit negative value D320h is subtracted from the signed 16-bit positive value 5321h, and the result saturates to 7FFFh because it is greater than 7FFFh, the largest possible signed 16-bit value.
- The signed 16-bit positive value 0FF9h is subtracted from the signed 16-bit negative value 8007h, and the result saturates to 8000h because it is less than 8000h, the smallest possible signed 16-bit value.
- The remaining operations are simple signed subtraction with no saturation.

Related Instructions See the PSUBB instruction.

See the PSUBD instruction.

See the PSUBW instruction.

See the PSUBSB instruction.

See the PSUBUSB instruction.

See the PSUBUSW instruction.

PSUBUSB

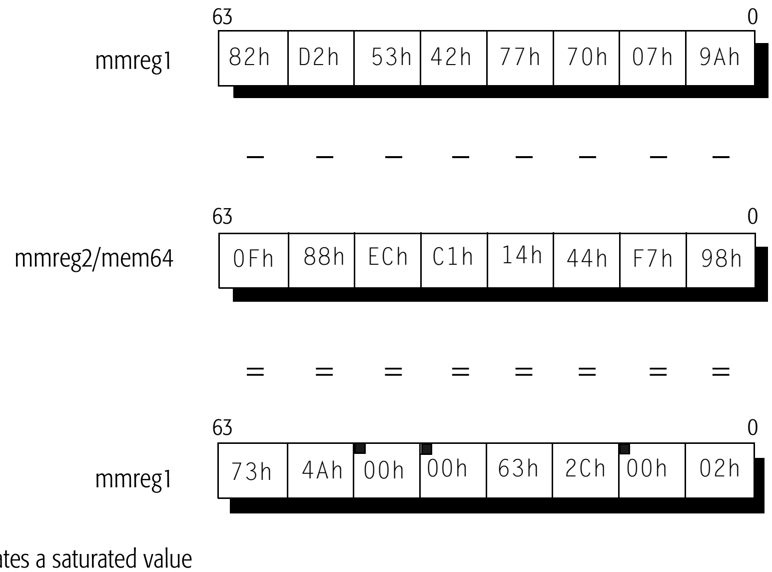
mnemonic	*opcode*	*description*
PSUBUSB mmreg1, mmreg2/mem64	0F D8h	Subtract unsigned packed 8-bit values and saturate

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.
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 PSUBUSB instruction subtracts eight unsigned 8-bit values in the source operand (an MMX register or a 64-bit memory location) from the eight corresponding unsigned 8-bit values in the destination operand (an MMX register). If any 8-bit source value is greater than its corresponding 8-bit destination value, the result saturates to 00h. The eight unsigned 8-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PSUBUSB Instruction



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

- The unsigned 8-bit value ECh is subtracted from the unsigned 8-bit value 53h, and the result saturates to 00h because the source operand is greater than the destination operand.
- The unsigned 8-bit value C1h is subtracted from the unsigned 8-bit value 42h, and the result saturates to 00h because the source operand is greater than the destination operand.
- The unsigned 8-bit value F7h is subtracted from the unsigned 8-bit value 07h, and the result saturates to 00h because the source operand is greater than the destination operand.
- All the remaining operations are simple unsigned subtraction with no saturation.

Related Instructions See the PSUBB instruction.

See the PSUBD instruction.

See the PSUBW instruction.

See the PSUBSB instruction.

See the PSUBSW instruction.

See the PSUBUSW instruction.

PSUBUSW

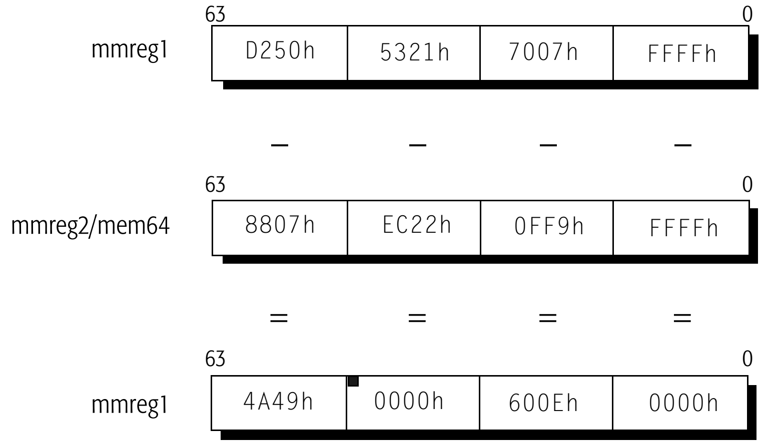
mnemonic	*opcode*	*description*
PSUBUSW mmreg1, mmreg2/mem64	0F D9h	Subtract unsigned packed 16-bit values and saturate

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.
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 PSUBUSW instruction subtracts four unsigned 16-bit values in the source operand (an MMX register or a 64-bit memory location) from the four corresponding unsigned 16-bit values in the destination operand (an MMX register). If any 16-bit source value is greater than its corresponding 16-bit destination value, the result saturates to 0000h. The four unsigned 16-bit results are stored in the MMX register specified as the destination operand.

Functional Illustration of the PSUBUSW Instruction



■ Indicates a saturated value

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

- ■ The unsigned 16-bit value EC22h is subtracted from the unsigned 16-bit value 5321h, and the result saturates to 0000h because the source operand is greater than the destination operand.
- ■ The remaining operations are simple unsigned subtraction with no saturation.

Related Instructions See the PSUBB instruction.

See the PSUBD instruction.

See the PSUBW instruction.

See the PSUBSB instruction.

See the PSUBSW instruction.

See the PSUBUSB instruction.

PSUBW

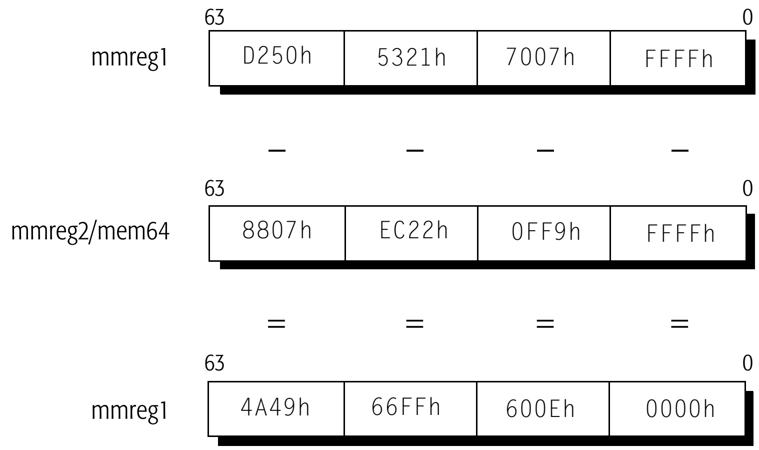
mnemonic	*opcode*	*description*
PSUBW mmreg1, mmreg2/mem64	0F F9h	Subtract unsigned packed 16-bit values with wraparound

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.
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 PSUBW instruction subtracts four unsigned 16-bit values in the source operand (an MMX register or a 64-bit memory location) from the four corresponding unsigned 16-bit values in the destination operand (an MMX register). If the source operand is larger than the destination operand, the result wraps around.

Functional Illustration of the PSUBW Instruction



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

- The unsigned 16-bit value EC22h is subtracted from the unsigned 16-bit value 5321h and the result wraps around to 66FFh.
- The remaining operations are simple unsigned subtraction with no saturation.

Related Instructions

See the PSUBB instruction.

See the PSUBD instruction.

See the PSUBSB instruction.

See the PSUBSW instruction.

See the PSUBUSB instruction.

See the PSUBUSW instruction.

PUNPCKHBW

mnemonic	*opcode*	*description*
PUNPCKHBW mmreg1, mmreg2/mem64	0F 68h	Unpack the high 32 bits of packed 8-bit values

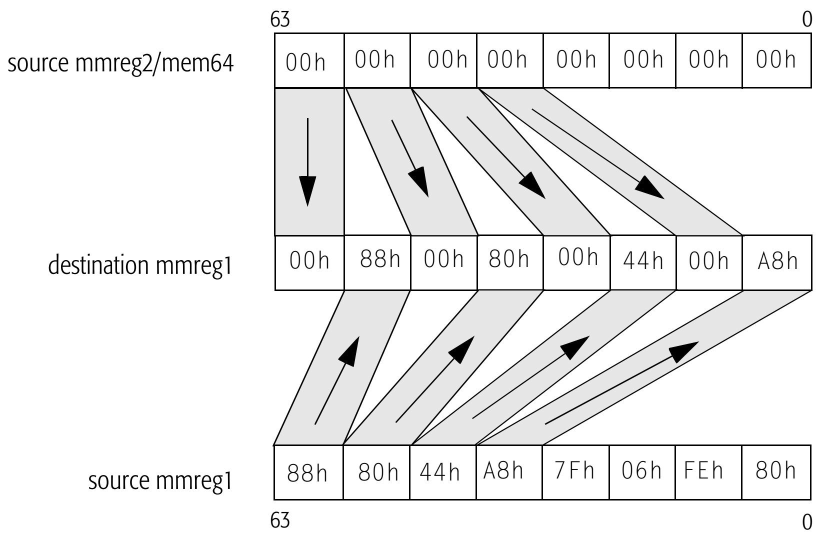
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate 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.
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 PUNPCKHBW instruction unpacks and interleaves four 8-bit values from the high 32 bits of the source operand (an MMX register or a 64-bit memory location) and four 8-bit values from the high 32 bits of the destination operand (an MMX register). The 8-bit values from the source operand become the high 8 bits of the 16-bit results, and the 8-bit values from the destination operand become the low 8 bits of the 16-bit results. The eight interleaved 8-bit values are stored in the MMX register specified as the destination operand.

Functional Illustration of the PUNPCKHBW Instruction

In the following figure, the destination register is shown at the center to illustrate the flow of data from the two source operands.



In the functional illustration of the PUNPCKHBW instruction, the 8-bit values from mmreg1 are stored in the low-order 8 bits of the 16-bit result. The mmreg2/mem64 source operand is set to all zero bits so it can provide zero fill in the high-order 8 bits of the 16-bit result. This is a method that can be used to expand unsigned 8-bit values into unsigned 16-bit operands for subsequent processing that requires higher precision.

Related Instructions

See the PACKSSWB instruction.

See the PACKUSWB instruction.

See the PSRAW instruction.

See the PUNPCKHDQ instruction.

See the PUNPCKHWD instruction.

See the PUNPCKLBW instruction.

See the PUNPCKLDQ instruction.

See the PUNPCKLWD instruction.

PUNPCKHDQ

mnemonic	*opcode*	*description*
PUNPCKHDQ mmreg1, mmreg2/mem64	0F 6Ah	Unpack the high 32 bits of packed 32-bit values

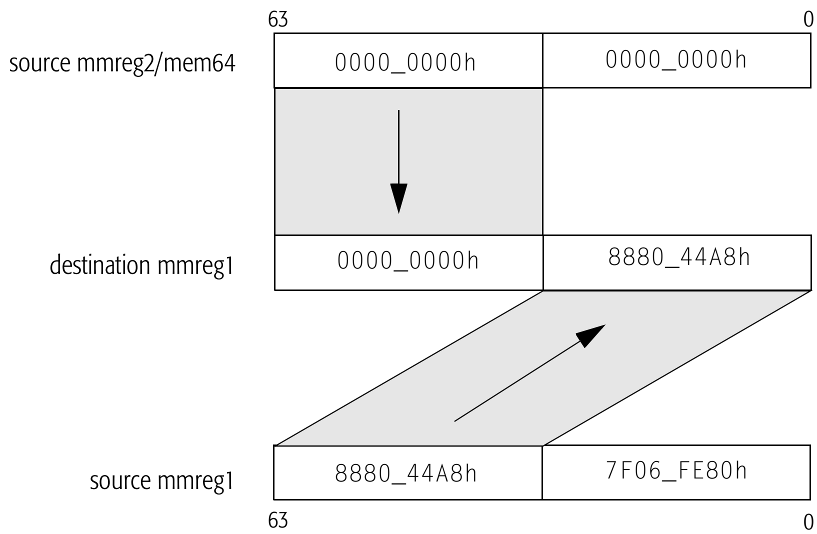
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate 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.
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 PUNPCKHDQ instruction unpacks and interleaves the high 32 bits of the source operand (an MMX register or a 64-bit memory location) and the high 32 bits of the destination operand (an MMX register). The 32-bit value from the source operand becomes the high 32 bits of the 64-bit result, and the 32-bit value from the destination operand becomes the low 32 bits of the 64-bit result. The interleaved 32-bit values are stored in the MMX register specified as the destination operand.

Functional Illustration of the PUNPCKHDQ Instruction

In the following figure, the destination register is shown at the center to illustrate the flow of data from the two source operands.



In the functional illustration of the PUNPCKHDQ instruction, the 32-bit value from mmreg1 is stored in the low-order 32 bits of the 64-bit result. The mmreg2/mem64 source operand is set to all zero bits so it can provide zero fill in the high-order 32 bits of the 64-bit result. This is a method that can be used to expand unsigned 32-bit values into unsigned 64-bit operands for subsequent processing that requires higher precision.

Related Instructions

See the PUNPCKHBW instruction.

See the PUNPCKHWD instruction.

See the PUNPCKLBW instruction.

See the PUNPCKLDQ instruction.

See the PUNPCKLWD instruction.

PUNPCKHWD

mnemonic	*opcode*	*description*
PUNPCKHWD mmreg1, mmreg2/mem64	0F 69h	Unpack the high 32 bits of packed 16-bit values

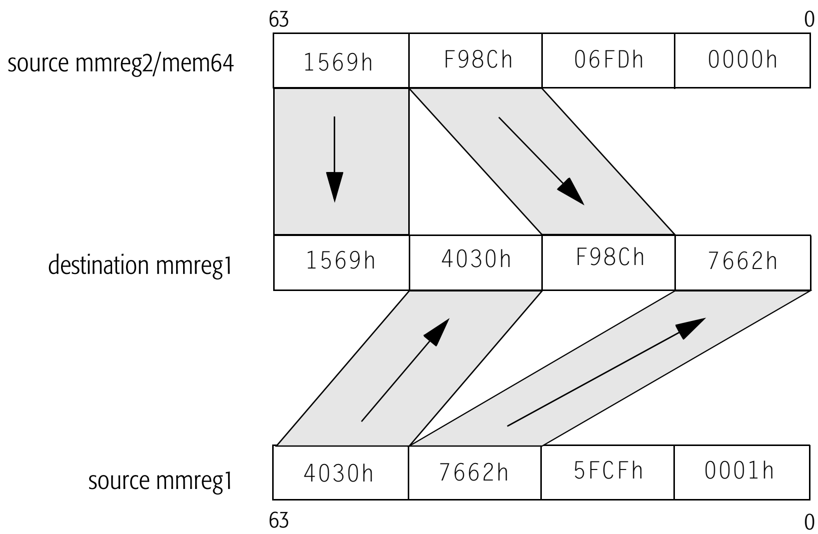
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate 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.
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 PUNPCKHWD instruction unpacks and interleaves two 16-bit values from the high 32 bits of the source operand (an MMX register or a 64-bit memory location) and two 16-bit values from the high 32 bits of the destination operand (an MMX register). The 16-bit values from the source operand become the high 16 bits of the 32-bit results, and the 16-bit values from the destination operand become the low 16 bits of the 32-bit results. The four interleaved 16-bit values are stored in the MMX register specified as the destination operand.

Functional Illustration of the PUNPCKHWD Instruction

In the following figure, the destination register is shown at the center to illustrate the flow of data from the two source operands.



In the functional illustration of the PUNPCKHWD instruction, the 16-bit values from mmreg1 are stored in the low-order 16 bits of the 32-bit result. The 16-bit values from the mmreg2/mem64 source operand are stored in the high-order 16 bits of the 32-bit result. This is an example of the use of the PUNPCKHWD instruction to assemble 32-bit operands from the high and low 16-bit results produced by the PMULHW and PMULLW instructions. In this example, the high and low 16-bit results are interleaved to produce the signed 32-bit results 1569_4030h and F98C_7662h.

Related Instructions

See the PACKSSDW instruction.

See the PSRAD instruction.

See the PMULHW instruction.

See the PMULLW instruction.

See the PUNPCKHBW instruction.

See the PUNPCKHDQ instruction.

See the PUNPCKLBW instruction.

See the PUNPCKLDQ instruction.

See the PUNPCKLWD instruction.

PUNPCKLBW

mnemonic	*opcode*	*description*
PUNPCKLBW mmreg1, mmreg2/mem32	0F 60h	Unpack the low 32-bits of packed 8-bit values

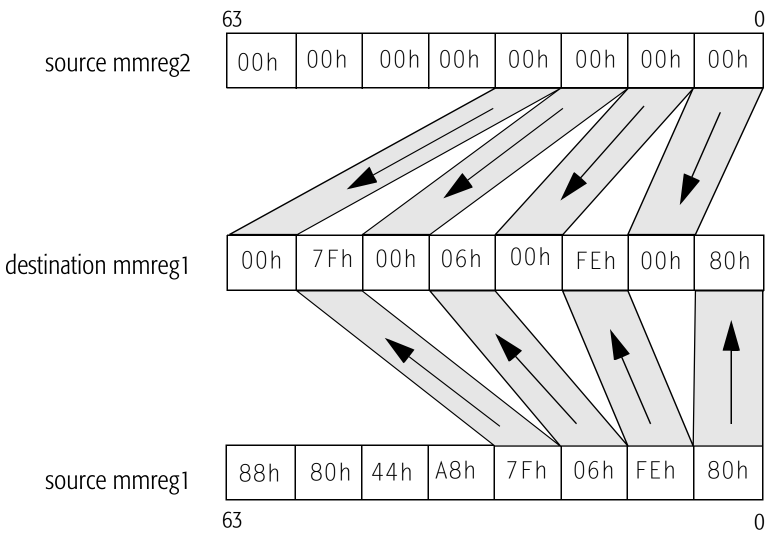
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

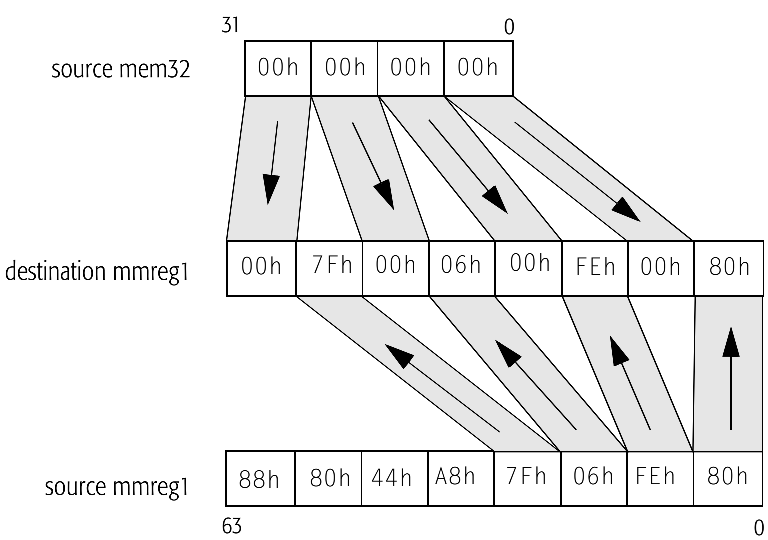
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (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 PUNPCKLBW instruction unpacks and interleaves four 8-bit values from the low 32 bits of the source operand (an MMX register or a 32-bit memory location) and four 8-bit values from the low 32 bits of the destination operand (an MMX register). The 8-bit values from the source operand become the high 8 bits of the 16-bit results, and the 8-bit values from the destination operand become the low 8 bits of the 16-bit results. The eight interleaved 8-bit values are stored in the MMX register specified as the destination operand.

Functional Illustration of the PUNPCKLBW Instruction

In the following figure, the destination register is shown at the center to illustrate the flow of data from the two source operands.





In the functional illustration of the PUNPCKLBW instruction, the 8-bit values from mmreg1 are stored in the low-order 8 bits of the 16-bit result. The mmreg2/mem32 source operand is set to all zero bits so it can provide zero fill in the high-order 8 bits of the 16-bit result. This is a method that can be used to expand unsigned 8-bit values into unsigned 16-bit operands for subsequent processing that requires higher precision.

Related Instructions

See the PACKSSWB instruction.

See the PACKUSWB instruction.

See the PSRAW instruction.

See the PUNPCKHBW instruction

See the PUNPCKHDQ instruction.

See the PUNPCKHWD instruction.

See the PUNPCKLDQ instruction.

See the PUNPCKLWD instruction.

PUNPCKLDQ

mnemonic	*opcode*	*description*
PUNPCKLDQ mmreg1, mmreg2/mem32	0F 62h	Unpack the low 32 bits of packed 32-bit values

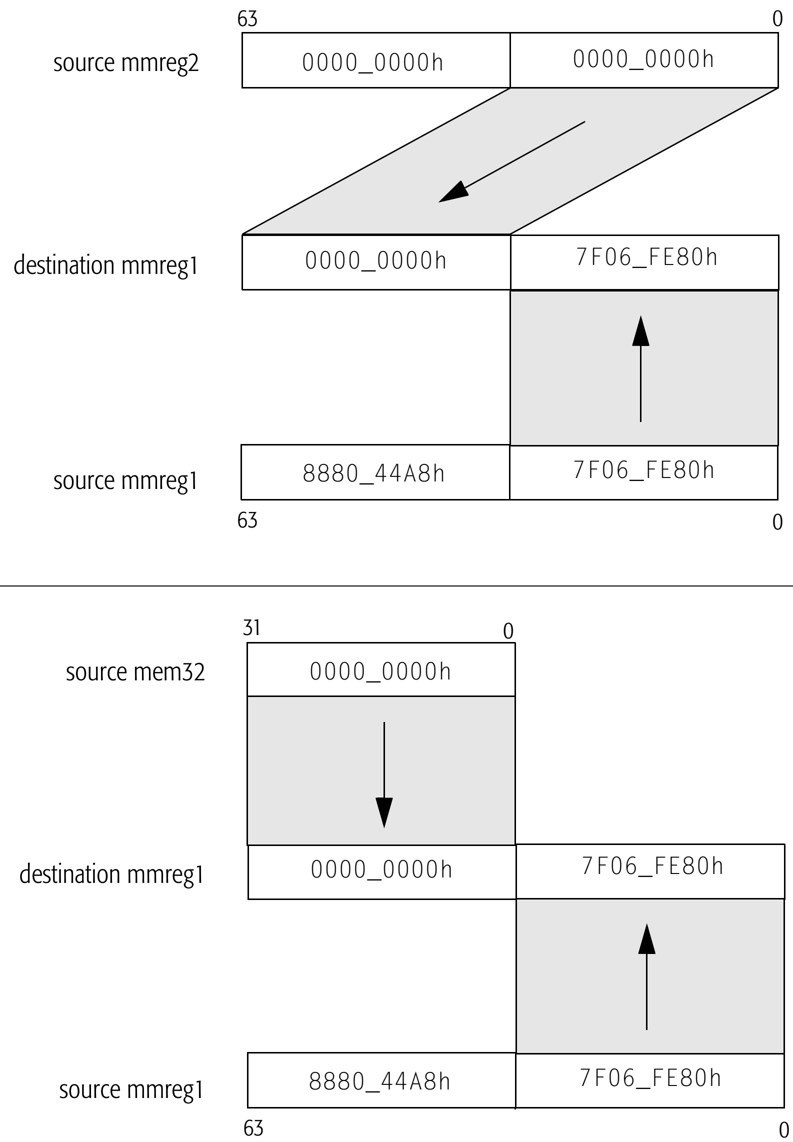
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	**Real**	**Virtual 8086**	**Protected**	**Description**
Invalid opcode (6)	X	X	X	The emulate 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.
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 PUNPCKLDQ instruction unpacks and interleaves the low 32 bits of the source operand (an MMX register or a 32-bit memory location) and the low 32 bits of the destination operand (an MMX register). The 32-bit value from the source operand becomes the high 32 bits of the 64-bit result, and the 32-bit value from the destination operand becomes the low 32 bits of the 64-bit result. The interleaved 32-bit values are stored in the MMX register specified as the destination operand.

Functional Illustration of the PUNPCKLDQ Instruction

In the following figure, the destination register is shown at the center to illustrate the flow of data from the two source operands.



In the functional illustration of the PUNPCKLDQ instruction, the 32-bit value from mmreg1 is stored in the low-order 32 bits of the 64-bit result. The mmreg2/mem32 source operand is set to all zero bits so it can provide zero fill in the high-order 32 bits of the 64-bit result. This is a method that can be used to expand unsigned 32-bit values into unsigned 64-bit operands for subsequent processing that requires higher precision.

Related Instructions

See the PUNPCKHBW instruction.
See the PUNPCKHDQ instruction.
See the PUNPCKHWD instruction.
See the PUNPCKLBW instruction.
See the PUNPCKLWD instruction.

PUNPCKLWD

mnemonic	*opcode*	*description*
PUNPCKLWD mmreg1, mmreg2/mem32	0F 61h	Unpack the low 32 bits of packed 16-bit values

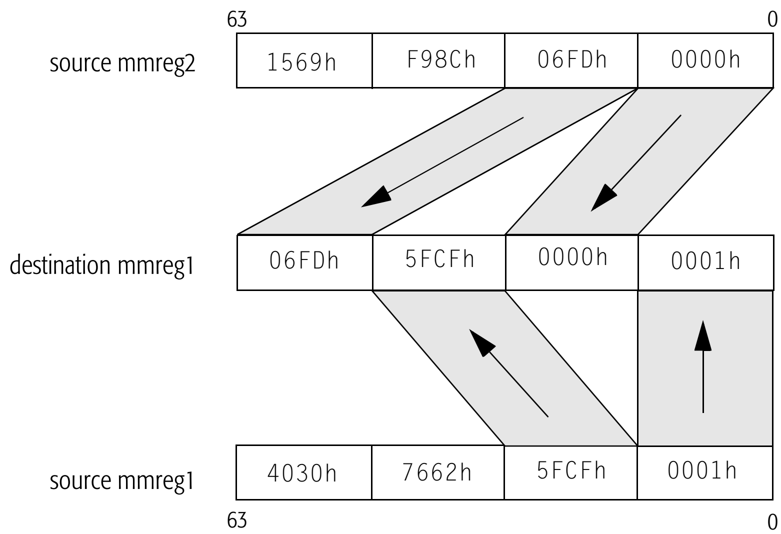
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

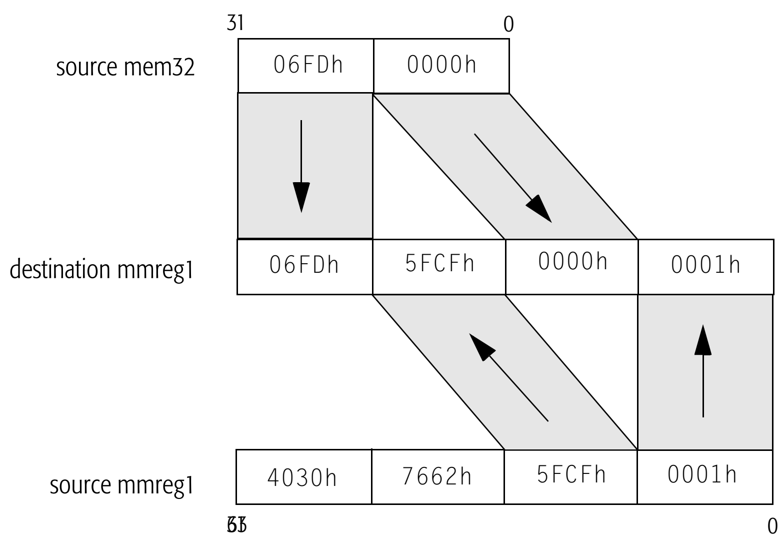
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (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 PUNPCKLWD instruction unpacks and interleaves two 16-bit values from the low 32 bits of the source operand (an MMX register or a 32-bit memory location) and two 16-bit values from the low 32 bits of the destination operand (an MMX register). The 16-bit values from the source operand become the high 16 bits of the 32-bit results, and the 16-bit values from the destination operand become the low 16 bits of the 32-bit results. The four interleaved 16-bit values are stored in the MMX register specified as the destination operand.

Functional Illustration of the PUNPCKLWD Instruction

In the following figure, the destination register is shown at the center to illustrate the flow of data from the two source operands.





In the functional illustration of the PUNPCKLWD instruction, the 16-bit values from mmreg1 are stored in the low-order 16 bits of the 32-bit result. The 16-bit values from the mmreg2/mem32 source operand are stored in the high-order 16 bits of the 32-bit result. This is an example of the use of the PUNPCKLWD instruction to assemble 32-bit operands from the high and low 16-bit results produced by the PMULHW and PMULLW instructions. In this example, the high and low 16-bit results are interleaved to produce the signed 32-bit results 06FD_5FCFh and 0000_0001h.

Related Instructions

See the PACKSSWD instruction.

See the PSRAD instruction.

See the PMULHW instruction.

See the PMULLW instruction.

See the PUNPCKHBW instruction.

See the PUNPCKHDQ instruction.

See the PUNPCKHWD instruction.

See the PUNPCKLBW instruction.

See the PUNPCKLDQ instruction.

PXOR

mnemonic	*opcode*	*description*
PXOR mmreg1, mmreg2/mem64	0F EFh	XOR 64-bit values

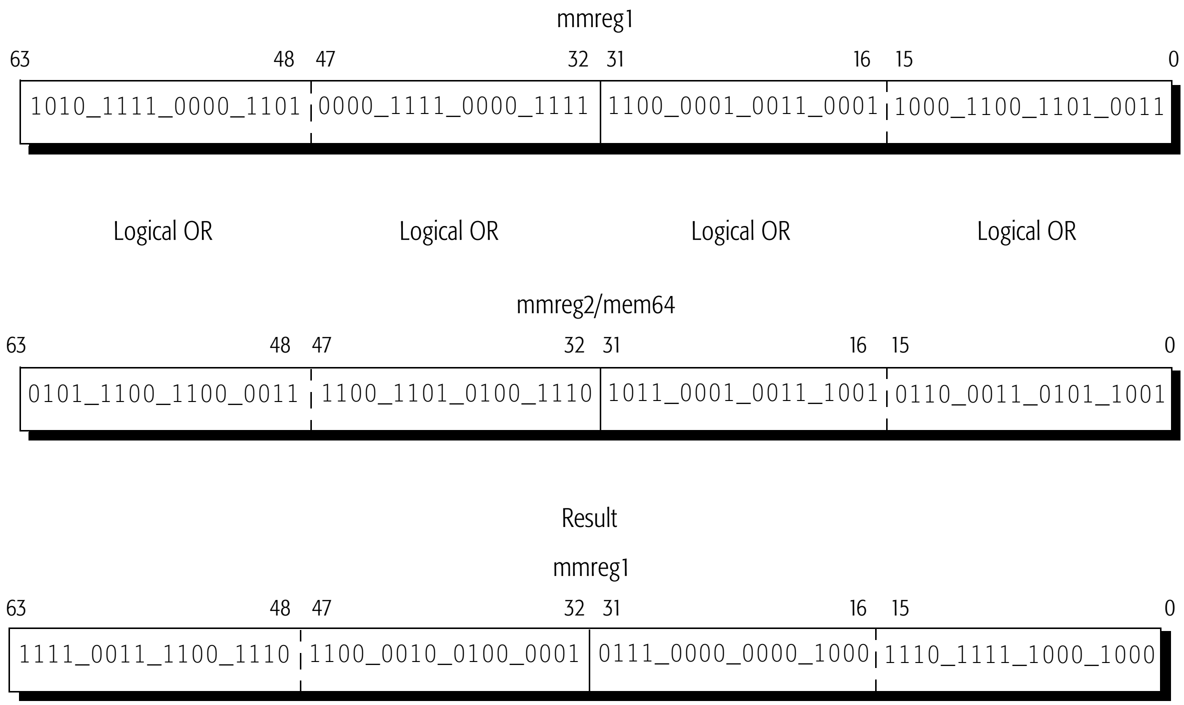
Privilege: none
Registers Affected: MMX
Flags Affected: none
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate 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.
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 PXOR instruction logically XORs the 64 bits of the source operand (an MMX register or a 64-bit memory location) with the 64 bits of the destination operand (an MMX register) and stores the result in the destination register.

A logical XOR produces a 1 bit if only one of the two input bits is a 1. If both input bits are 0 or both input bits are 1, a logical XOR produces a 0 bit.

Functional Illustration of the PXOR Instruction



In the functional illustration of the PXOR instruction, the 64-bit source value is logically XOR'd to the 64-bit destination value, and the result is stored in the destination register.

Related Instructions See the PAND instruction.

See the PANDN instruction.

See the POR instruction.