Intel Architecture MMX™ Technology

Programmer's Reference Manual

March 1996
Order No. 243007-002

Subject to the terms and conditions set forth below, Intel hereby grants you a nonexclusive, nontransferable license, under its patents and copyrights on the example code sequences contained in Chapters 3, 4 and 5 of the Programmer's Reference Manual , to use, reproduce and distribute such example code sequences solely as part of your computer program(s) and solely in order to allow your computer program(s) to implement the multimedia instruction extensions contained in such sequences solely with respect to the Intel instruction set architecture. No other license, express, implied, statutory, by estoppel or otherwise, to any other intellectual property rights is granted herein.

THIS DOCUMENT AND ALL INFORMATION, PROPOSALS, SAMPLES AND OTHER MATERIALS PROVIDED IN CONNECTION WITH OR IN RELATION TO THIS DOCUMENT (INCLUDING, WITHOUT LIMITATION, THE EXAMPLE CODE SEQUENCES) ARE PROVIDED "AS IS" WITH NO WARRANTIES, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, AND INTEL SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT OR FITNESS FOR ANY PARTICULAR PURPOSE.

Any use or distribution of this document or the materials contained herein must fully comply with all then current laws of the United States including, without limitation, rules and regulations of the United States Office of Export Administration and other applicable U.S. governmental agencies.

THIS DOCUMENT AND THE MATERIALS PROVIDED HEREIN ARE PROVIDED WITHOUT CHARGE. THEREFORE, IN NO EVENT WILL INTEL BE LIABLE FOR ANY DAMAGES OF ANY KIND, INCLUDING DIRECT OR INDIRECT DAMAGES, LOSS OF DATA, LOST PROFITS, COST OF COVER OR SPECIAL, INCIDENTAL, CONSEQUENTIAL, DAMAGES ARISING FROM THE USE OF THE MATERIALS PROVIDED HEREIN, INCLUDING WITHOUT LIMITATION THE EXAMPLE CODE SEQUENCES, HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY. THIS LIMITATION WILL APPLY EVEN IF INTEL OR ANY AUTHORIZED AGENT OF INTEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Information in this document is provided in connection with Intel products. No license under any patent or copyright is granted expressly or impliedly by this publication. Intel assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of Intel products except as provided in Intel's Terms and Conditions of Sale for such products.

Intel retains the right to make changes to these specifications at any time, without notice. Microcomputer Products may have minor variations to their specifications known as errata.

*Other brands and names are the property of their respective owners.

Contact your local Intel sales office or your distributor to obtain the latest specifications before placing product orders.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation

P.O. Box 7641

Mt. Prospect IL 60056-764

or call 1-800-879-4683

`TABLE OF CONTENTS

CHAPTER 3
APPLICATION PROGRAMMING MODEL

CHAPTER 4
SYSTEM PROGRAMMING MODEL

CHAPTER 5
INTEL ARCHITECTURE MMX™ INSTRUCTION SET

APPENDIX A
IA MMX™ INSTRUCTION SET SUMMARY

APPENDIX B
IA MMX™ INSTRUCTION FORMATS AND ENCODINGS

APPENDIX C
ALPHABETICAL LIST OF IA MMX™ INSTRUCTION SET MNEMONICS

APPENDIX D
IA MMX™ INSTRUCTION SET OPCODE MAP

Figures

Tables

Examples

1

Introduction to the Intel Architecture MMX™ Technology

CHAPTER 1
INTRODUCTION TO THE INTEL ARCHITECTURE MMX™ TECHNOLOGY

1.1. ABOUT THE INTEL ARCHITECTURE MMX™ TECHNOLOGY

The media extensions for the Intel Architecture (IA) were designed to enhance performance of advanced media and communication applications. The MMX™ technology provides a new level of performance to computer platforms by adding new instructions and defining new 64-bit data types, while preserving compatibility with software and operating systems developed for the Intel Architecture.

The MMX technology introduces new general-purpose instructions. These instructions operate in parallel on multiple data elements packed into 64-bit quantities. They perform arithmetic and logical operations on the different data types. These instructions accelerate the performance of applications with compute-intensive algorithms that perform localized, recurring operations on small native data. This includes applications such as motion video, combined graphics with video, image processing, audio synthesis, speech synthesis and compression, telephony, video conferencing, 2D graphics, and 3D graphics

The IA MMX instruction set has a simple and flexible software model with no new mode or operating-system visible state. The MMX instruction set is fully compatible with all Intel Architecture microprocessors. All existing software continues to run correctly, without modification, on microprocessors that incorporate the MMX technology, as well as in the presence of existing and new applications that incorporate this technology.

1.1.1. Single Instruction, Multiple Data (SIMD) Technique

The MMX technology uses the Single Instruction, Multiple Data (SIMD) technique. This technique speeds up software performance by processing multiple data elements in parallel, using a single instruction. The MMX technology supports parallel operations on byte, word, and doubleword data elements, and the new quadword (64-bit) integer data type.

1.1.2. Performance Improvement

Modern media, communications, and graphics applications now include sophisticated algorithms that perform recurring operations on small data types. The MMX technology directly addresses the need of these applications. For example, most audio data is represented in 16-bit (word) quantities. The MMX instructions can operate on four of these words simultaneously with one instruction. Video and graphics information is commonly represented as palletized 8-bit (byte) quantities; one MMX instruction can operate on eight of these bytes simultaneously.

1.2. ABOUT THIS MANUAL

It is assumed that the reader is familiar with the Intel Architecture software model and Assembly language programming.

This manual describes the IA MMX instruction set and introduces the architectural features, instruction set, data types, data formats, application programming model, and system programming model of the MMX technology. It also explains how to use the new instructions to significantly increase the performance of applications.

In this context, architecture refers to the conceptual structure and functional behavior of MMX technology as seen by a programmer, but not the logical organization or performance aspects of the actual implementation.

This manual is organized into five chapters, including this chapter (Chapter 1), and four appendices:

Chapter 1—Introduction to the Intel Architecture MMX™ Technology

Chapter 2—Intel Architecture MMX™ Technology Features: This chapter provides an overview of the IA MMX technology and its new features.

Chapter 3—Application Programming Model: This chapter describes the software conventions and architecture of the IA MMX technology. It defines the steps for writing MMX code.

Chapter 4—System Programming Model: This chapter discusses interfacing with the operating system and compatibility with Intel Architecture.

Chapter 5—Intel Architecture MMX™ Instruction Set: This chapter details the instructions, mnemonics, and instruction notations. A full description including graphical representations of the new instructions is presented.

Appendix A—IA MMX™ Instruction Set Summary: This appendix summarizes the instructions by functional groups.

Appendix B—IA MMX™ Instruction Formats and Encodings: This appendix lists the instruction formats and encodings. It also lists a detailed break-down of the instruction operations and the supported data types.

Appendix C—Alphabetical list of IA MMX™ Instruction Set Mnemonics: This appendix summarizes operand types, encodings in hexadecimal, and the formats used.

Appendix D—IA MMX™ Instruction Set Opcode Map: This appendix provides a detailed encoding table of opcode mappings.

1.3. RELATED DOCUMENTATION

Refer to the following documentation for more information related to Intel Architecture:

Pentium® Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual. Intel Corporation, Order Number 240897.

Pentium® Pro Processor Developer's Manual, Volumes 2 and 3. Intel Corporation, Order Numbers 242691 and 242692.

Intel Architecture MMX™ Technology Developers' Manual - Intel Corporation, Order Number 243010.

Refer to Intel's corporate website for the latest information on related documentation:

http://www.intel.com

2

Intel Architecture MMX™ Technology Features

CHAPTER 2
INTEL ARCHITECTURE MMX™ TECHNOLOGY FEATURES

This chapter provides a general overview of the architectural features of the Intel Architecture MMX™ technology.

2.1. NEW FEATURES

MMX technology provides the following new features, while maintaining backward compatibility with all existing Intel Architecture microprocessors, IA applications, and operating systems.

- New data types
- Eight MMX registers
- Enhanced instruction set

The performance of applications which use these new features of MMX technology can be enchanced.

2.2. NEW DATA TYPES

The principal data type of the IA MMX technology is the packed fixed-point integer. The decimal point of the fixed-point values is implicit and is left for the user to control for maximum flexibility.

The IA MMX technology defines the following four new 64-bit data types (See Figure 2-1):

Packed byte	Eight bytes packed into one 64-bit quantity
Packed word	Four words packed into one 64-bit quantity
Packed doubleword	Two doublewords packed into one 64-bit quantity
Quadword	One 64-bit quantity

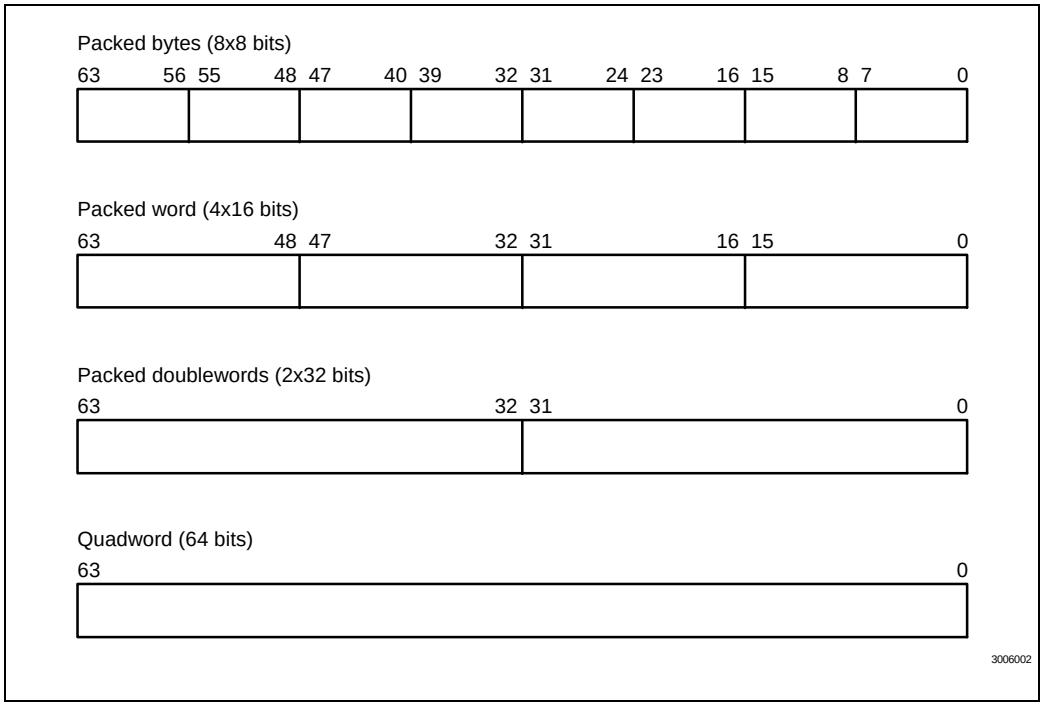


Figure 2-1. Packed Data Types

2.3. MMX™ REGISTERS

The IA MMX technology provides eight 64-bit, general-purpose registers. These registers are aliased on the floating-point registers. The operating system handles the MMX technology as it would handle floating-point. (See Section 4.3 for more details on register aliasing.)

The MMX registers can hold packed 64-bit data types. The MMX instructions access the MMX registers directly using the register names MM0 to MM7 (See Figure 2-2).

MMX registers can be used to perform calculations on data. They cannot be used to address memory; addressing is accomplished by using the integer registers and standard IA addressing modes.

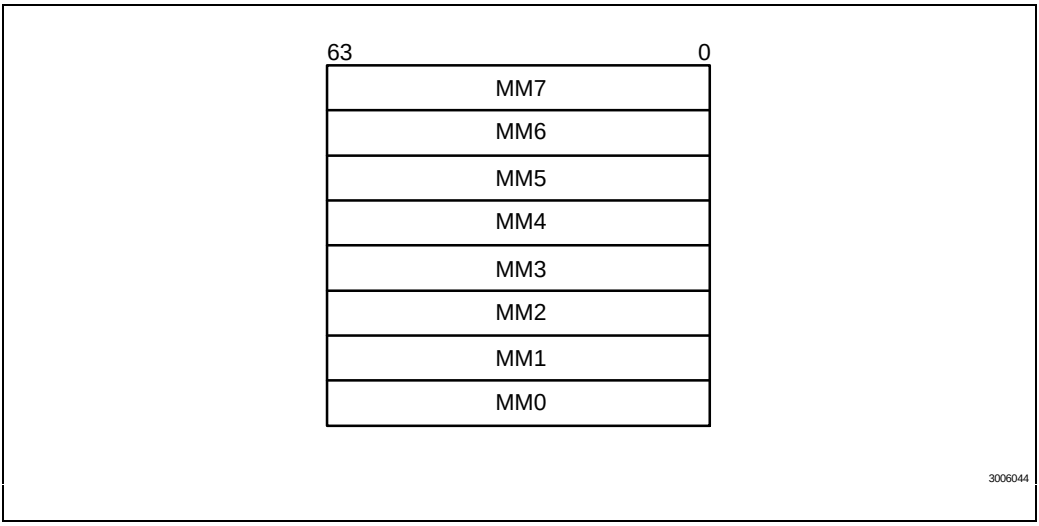


Figure 2-2. MMX™ Register Set

2.4. EXTENDED INSTRUCTION SET

The IA MMX instruction set supplies a rich set of instructions that operate on all data elements of a packed data type, in parallel. The MMX instructions can operate on either signed or unsigned data elements.

The MMX instructions implement two new principles (discussed in section 2.4.2.):

- Operations on packed data
- Saturation arithmetic

2.4.1. Packed Data

The MMX instructions can operate on groups of eight bytes, four words, and two doublewords. These groups of 64 bits are referred to as packed data. The same 64 bits of data can be treated as any one of the packed data types. Data is cast by the type specified by the instruction.

For example, the PADDB (Add Packed Bytes) instruction adds two groups of eight packed bytes. The PADDW (Add Packed Words) instruction, which adds packed words, could

operate on the same 64 bits as the PADDB instruction treating the 64 bits as four 16-bit words.

2.4.2. Saturation Arithmetic and Wrap Around

The MMX technology supports a new arithmetic capability known as saturating arithmetic. Saturation is best defined by contrasting it with wraparound mode.

In wraparound mode, results that overflow or underflow are truncated and only the lower (least significant) bits of the result are returned. That is, the carry is ignored.

In saturation mode, results of an operation that overflow or underflow are clipped (saturated) to a data-range limit for the data type (see Table 2-1). The result of an operation that exceeds the range of a data-type saturates to the maximum value of the range. A result that is less than the range of a data type saturates to the minimum value of the range. This is useful in many cases, such as color calculations.

For example, when the result exceeds the data range limit for signed bytes, it is saturated to 0x7F (0xFF for unsigned bytes). If a value is less than the data range limit, it is saturated to 0x80 for signed bytes (0x00 for unsigned bytes).

Saturation provides a useful feature of avoiding wraparound artifacts. In the example of color calculations, saturation causes a color to remain pure black or pure white without allowing for an inversion.

Table 2-1. Data Range Limits for Saturation

	Lower Limit		Upper Limit	
Signed	**Hexadecimal**	**Decimal**	**Hexadecimal**	**Decimal**
Byte	80H	-128	7FH	127
Word	8000H	-32,768	7FFFH	32,767
Unsigned				
Byte	00H	0	FFH	255
Word	0000H	0	FFFFH	65,535

MMX instructions do not indicate overflow or underflow occurrence by generating exceptions or setting flags.

2.4.3. Instruction Group Overview

This section provides an overview of the MMX instruction groups. See Chapter 5 for detailed information on the instructions, including information on encoding, operation, and exceptions.

The fifty-seven new MMX instructions are grouped into these categories:

- Arithmetic Instructions
- Comparison Instructions
- Conversion Instructions
- Logical Instructions
- Shift Instructions
- Data Transfer Instructions
- Empty MMX State (EMMS) Instruction

2.4.3.1. ARITHMETIC INSTRUCTIONS

Packed Addition and Subtraction

The PADD (Packed Add) and PSUB (Packed Subtract) instructions add or subtract the signed or unsigned data elements of the source operand to or from the destination operand in wrap-around mode. These instructions support packed byte, packed word, and packed doubleword data types.

The PADDS (Packed Add with Saturation) and PSUBS (Packed Subtract with Saturation) instructions add or subtract the signed data elements of the source operand to or from the signed data elements of the destination operand and saturate the result to the limits of the signed data-type range. These instructions support packed byte and packed word data types.

The PADDUS (Packed Add Unsigned with Saturation) and PSUBUS (Packed Subtract Unsigned with Saturation) instructions add or subtract the unsigned data elements of the source operand to or from the unsigned data elements of the destination operand and saturate the result to the limits of the unsigned data-type range. These instructions support packed byte and packed word data types.

Packed Multiplication

Packed multiplication instructions perform four multiplications on pairs of signed 16-bit operands, producing 32-bit intermediate results. Users may choose the low-order or high-order parts of each 32-bit result.

The PMULHW (Packed Multiply High) and PMULLW (Packed Multiply Low) instructions multiply the signed words of the source and destination operands and write the high-order or low-order 16 bits of each of the results to the destination operand.

Packed Multiply Add

The PMADDWD (Packed Multiply and Add) instruction calculates the products of the signed words of the source and destination operands. The four intermediate 32-bit doubleword products are summed in pairs to produce two 32-bit doubleword results.

2.4.3.2. COMPARISON INSTRUCTIONS

The PCMPEQ (Packed Compare for Equal) and PCMPGT (Packed Compare for Greater Than) instructions compare the corresponding data elements in the source and destination operands for equality or value greater than, respectively. These instructions generate a mask of ones or zeros which are written to the destination operand. Logical operations can use the mask to select elements. This can be used to implement a packed conditional move operation without a branch or a set of branch instructions. No flags are set.

These instructions support packed byte, packed word and packed doubleword data types.

2.4.3.3. CONVERSION INSTRUCTIONS

Pack and Unpack

The Pack and Unpack instructions perform conversions between the packed data types.

The PACKSS (Packed with Signed Saturation) instruction converts signed words into signed bytes or signed doublewords into signed words, in signed saturation mode.

The PACKUS (Packed with Unsigned Saturation) instruction converts signed words into unsigned bytes, in unsigned saturation mode.

The PUNPCKH (Unpack High Packed Data) and PUNPCKL (Unpack Low Packed Data) instructions convert bytes to words, words to doublewords, or doublewords to quadwords.

2.4.3.4. LOGICAL INSTRUCTIONS

The PAND (Bitwise Logical And), PANDN (Bitwise Logical And Not), POR (Bitwise Logical OR), and PXOR (Bitwise Logical Exclusive OR) instructions perform bitwise logical operations on 64-bit quantities.

2.4.3.5. SHIFT INSTRUCTIONS

The logical shift left, logical shift right and arithmetic shift right instructions shift each element by a specified number of bits. The logical left and right shifts also enable a 64-bit quantity (quadword) to be shifted as one block, assisting in data type conversions and alignment operations.

The PSLL (Packed Shift Left Logical) and PSRL (Packed Shift Right Logical) instructions perform a logical left or right shift, and fill the empty high or low order bit positions with zeros. These instructions support packed word, packed doubleword, and quadword data types.

The PSRA (Packed Shift Right Arithmetic) instruction performs an arithmetic right shift, copying the sign bit into empty bit positions on the upper end of the operand. This instruction supports packed word and packed doubleword data types.

2.4.3.6. DATA TRANSFER INSTRUCTIONS

The MOVD (Move 32 Bits) instruction transfers 32 bits of packed data from memory to MMX registers and visa versa, or from integer registers to MMX registers and visa versa.

The MOVQ (Move 64 Bits) instruction transfers 64-bits of packed data from memory to MMX registers and vise versa, or transfers data between MMX registers.

2.4.3.7. EMMS (EMPTY MMX™ STATE) INSTRUCTION

The EMMS instruction empties the MMX state. This instruction must be used to clear the IA MMX state (empty the floating-point tag word) at the end of an MMX routine before calling other routines that can execute floating-point instructions.

2.4.4. Instruction Operand

All MMX instructions, except the EMMS instruction, reference and operate on two operands: the source and destination operands. The right operand is the source and the left operand is the destination. The destination operand may also be a second source operand for the operation. The instruction overwrites the destination operand with the result.

For example, a two-operand instruction would be decoded as:

DEST(left operand) ← DEST (left operand) OP SRC (right operand)

The source operand for all the MMX instructions (except the data transfer instructions), can reside either in memory or in an MMX register. The destination operand resides in an MMX register.

For data transfer instructions, the source and destination operands can also be an integer register (for the MOVD instruction) or memory location (for both the MOVD and MOVQ instructions).

2.5. COMPATIBILITY

The IA MMX state is aliased upon the IA floating-point state. No new state or mode is added to support the MMX technology. The same floating-point instructions that save and restore the floating-point state also handle the IA MMX state (for example, during context switching).

MMX technology uses the same interface techniques between the floating-point architecture and the operating system (primarily for task switching purposes). For more detail, see Section 4.1.

3

Application Programming Model

CHAPTER 3
APPLICATION PROGRAMMING MODEL

This chapter describes the application programming environment as seen by compiler writers and assembly-language programmers. It also describes the architectural features which directly affect applications.

3.1. DATA FORMATS

3.1.1. Memory Data Formats

The Intel Architecture MMX™ technology introduces new packed data types, each 64 bits long. The data elements can be:

- eight packed, consecutive 8-bit bytes
- four packed, consecutive 16-bit words
- two packed, consecutive 32-bit doublewords

The 64 bits are numbered 0 through 63. Bit 0 is the least significant bit (LSB), and bit 63 is the most significant bit (MSB).

The low-order bits are the lower part of the data element and the high-order bits are the upper part of the data element. For example, a word contains 16 bits numbered 0 through 15, the byte containing bits 0-7 of the word is called the low byte, and the byte containing bits 8-15 is called the high byte.

Bytes in a multi-byte format have consecutive memory addresses. The ordering is always little endian. That is, the bytes with the lower addresses are less significant than the bytes with the higher addresses.

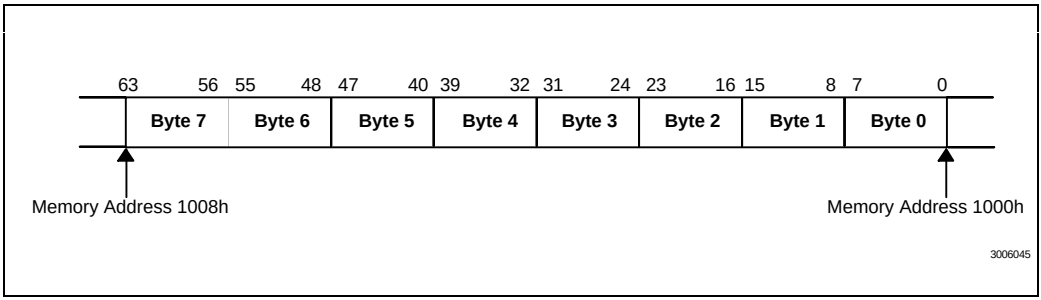


Figure 3-1. Eight Packed Bytes in Memory (at address 1000H)

3.1.2. IA MMX™ Register Data Formats

Values in IA MMX registers have the same format as a 64-bit quantity in memory. MMX registers have two data access modes: 64-bit access mode and 32-bit access mode.

The 64-bit access mode is used for 64-bit memory access, 64-bit transfer between MMX registers, all pack, logical and arithmetic instructions, and some unpack instructions.

The 32-bit access mode is used for 32-bit memory access, 32-bit transfer between integer registers and MMX registers, and some unpack instructions.

3.1.3. IA MMX™ Instructions and the Floating-Point Tag Word

After each MMX instruction, the entire floating-point tag word is set to Valid (00s). The Empty MMX State (EMMS) instruction sets the entire floating-point tag word to Empty (11s).

Section 4.3.2. describes the effects of floating-point and MMX instructions on the floating-point tag word. For details on floating-point tag word, refer to the *Pentium® Processor Family Developer's Manual*, Volume 3, Section 6.2.1.4.

3.2. PREFIXES

Table 3-1 details the effect of a prefix on IA MMX instructions.

Table 3-1. IA MMX™ Instruction Behavior with Prefixes Used by Application Programs

Prefix Type	The Effect of Prefix on IA MMX™ Instructions
Address size (67H)	Affects IA MMX instructions with a memory operand. Ignored by IA MMX instructions without a memory operand.
Operand size (66H)	Ignored.
Segment override	Affects IA MMX instructions with a memory operand. Ignored by IA MMX instructions without a memory operand.
Repeat	Ignored.
Lock (F0H)	Generates an invalid opcode exception.

See the *Pentium® Processor Family Developer's Manual*, Volume 3, Section 3.4. for information related to prefixes.

3.3. WRITING APPLICATIONS WITH IA MMX™ CODE

3.3.1. Detecting IA MMX™ Technology Existence Using the CPUID Instruction

Use the CPUID instruction to determine whether the processor supports the IA MMX instruction set (refer to the *Pentium® Processor Family Developer's Manual*, Volume 3, Chapter 25, for more detail on the CPUID instruction). When the IA MMX technology support is detected by the CPUID instruction, it is signaled by setting bit 23 (IA MMX technology bit) in the feature flags to 1. In general, two versions of the routine can be created: one with scalar instructions and one with MMX instructions. The application will call the appropriate routine depending on the results of the CPUID instruction. If MMX technology support is detected, then the MMX routine is called; if no support for the MMX technology exists, the application calls the scalar routine.

NOTE

The CPUID instruction will continue to report the existence of the IA MMX technology if the CR0.EM bit is set (which signifies that the CPU is configured to generate exception Int 7 that can be used to emulate floating

point instructions). In this case, executing an MMX instruction results in an invalid opcode exception.

Example 3-1 illustrates how to use the CPUID instruction. This example does not represent the entire CPUID sequence, but shows the portion used for IA MMX technology detection.

Example 3-1. Partial sequence of IA MMX™ technology detection by CPUID

```
...                             ; identify existence of CPUID instruction
...
...                             ; identify Intel processor
....
mov     EAX, 1                  ; request for feature flags
CPUID                           ; 0Fh, 0A2h CPUID instruction
test    EDX, 00800000h          ; Is IA MMX technology bit (Bit 23 of EDX) in feature flags set?
jnz     MMX_Technology_Found
```

3.3.2. The EMMS Instruction

When integrating the MMX routine into an application running under an existing operating system (OS), programmers need to take special precautions, similar to those when writing floating-point (FP) code.

When an MMX instruction executes, the floating-point tag word is marked valid (00s). Subsequent floating-point instructions that will be executed may produce unexpected results because the floating-point stack seems to contain valid data. The EMMS instruction marks the floating-point tag word as empty. Therefore, it is imperative to use the EMMS instruction at the end of every MMX routine.

The EMMS instruction must be used in each of the following cases:

- Application utilizing FP instructions calls an MMX technology library/DLL
- Application utilizing MMX instructions calls a FP library/DLL
- Switch between MMX code in a task/thread and other tasks/threads in cooperative operating systems.

If the EMMS instruction is not used when trying to execute a floating-point instruction, the following may occur:

- Depending on the exception mask bits of the floating-point control word, a floating-point exception event may be generated.

- A "soft exception" may occur. In this case floating-point code continues to execute, but generates incorrect results. This happens when the floating-point exceptions are masked and no visible exceptions occur. The internal exception handler (microcode, not user visible) loads a NaN (Not a Number) with an exponent of 11..11B onto the floating-point stack. The NaN is used for further calculations, yielding incorrect results.
- A potential error may occur only if the operating system does NOT manage floating-point context across task switches. These operating systems are usually cooperative operating systems. It is imperative that the EMMS instruction execute at the end of all the MMX routines that may enable a task switch immediately after they end execution (explicit yield API or implicit yield API).

3.3.3. Interfacing with IA MMX™ Technology Procedures and Functions

The MMX technology enables direct access to all the MMX registers. This means that all existing interface conventions that apply to the use of other general registers such as EAX, EBX will also apply to the MMX register usage.

An efficient interface might pass parameters and return values via the pre-defined MMX registers, or a combination of memory locations (via the stack) and MMX registers. This interface would have to be written in assembly language since passing parameters through MMX registers is not currently supported by any existing C compilers. Do not use the EMMS instruction when the interface to the MMX code has been defined to retain values in the MMX register.

If a high-level language, such as C, is used, the data types could be defined as a 64-bit structure with packed data types.

When implementing usage of IA MMX instructions in high level languages other approaches can be taken, such as:

- Passing MMX type parameters to a procedure by passing a pointer to a structure via the integer stack.
- Returning a value from a function by returning the pointer to a structure.

3.3.4. Writing Code with IA MMX™ and Floating-Point Instructions

The MMX technology aliases the MMX registers on the floating-point registers. The main reason for this is to enable MMX technology to be fully compatible and transparent to

existing software environments (operating systems and applications). This way operating systems will be able to include new applications and drivers that use the IA MMX technology.

An application can contain both floating-point and MMX code. However, the user is discouraged from causing frequent transitions between MMX and floating-point instructions by mixing MMX code and floating-point code.

3.3.4.1. RECOMMENDATIONS AND GUIDELINES

Do not mix MMX code and floating-point code at the instruction level for the following reasons:

- The TOS (top of stack) value of the floating-point status word is set to 0 after each MMX instruction. This means that the floating-point code loses its pointer to its floating-point registers if the code mixes MMX instructions within a floating-point routine.
- An MMX instruction write to an MMX 64-bit register writes ones (11s) to the exponent part of the corresponding floating-point register.
- Floating-point code that uses register contents that were generated by the MMX instructions may cause floating-point exceptions or incorrect results. These floating-point exceptions are related to undefined floating-point values and floating-point stack usage.
- All MMX instructions (except EMMS) set the entire tag word to the valid state (00s in all tag fields) without preserving the previous floating-point state.
- Frequent transitions between the MMX and floating-point instructions may result in significant performance degradation in some implementations.

If the application contains floating-point and MMX instructions, follow these guidelines:

- Partition the MMX technology module and the floating-point module into separate instruction streams (separate loops or subroutines) so that they contain only instructions of one type.
- Do not rely on register contents across transitions.
- When the MMX state is not required, empty the MMX state using the EMMS instruction.
- Exit the floating-point code section with an empty stack.

Example 3-2. Floating-point and MMX™ Code

```
FP_code:
                ..
                ..          (*leave the FP stack empty*)
MMX_code:
                ..
                EMMS        (*mark the FP tag word as empty*)
FP_code 1:
                ..
                ..          (*leave the FP stack empty*)
```

3.3.5. Multitasking Operating System Environment

An application needs to identify the nature of the multitasking operating system on which it runs. Each task retains its own state which must be saved when a task switch occurs. The processor state (context) consists of the integer registers and floating-point and MMX registers.

Operating systems can be classified into two types:

- Cooperative multitasking operating system
- Preemptive multitasking operating system

The behavior of the two operating system types in context switching is described in Section 4.1.1.

3.3.5.1. COOPERATIVE MULTITASKING OPERATING SYSTEM

Cooperative multitasking operating systems do not save the FP or MMX state when performing a context switch. Therefore, the application needs to save the relevant state before relinquishing direct or indirect control to the operating system.

3.3.5.2. PREEMPTIVE MULTITASKING OPERATING SYSTEM

Preemptive multitasking operating systems are responsible for saving and restoring the FP and MMX state when performing a context switch. Therefore, the application does not have to save or restore the FP and MMX state.

3.3.6. Exception Handling in IA MMX™ Application Code

MMX instructions generate the same type of memory-access exceptions as other Intel Architecture instructions. Some examples are: page fault, segment not present, and limit violations. Existing exception handlers can handle these types of exceptions. They do not have to be modified.

Unless there is a pending floating-point exception, MMX instructions do not generate numeric exceptions. Therefore, there is no need to modify existing exception handlers or add new ones.

If a floating-point exception is pending, the subsequent MMX instruction generates a numeric error exception (Int 16 and/or FERR#). The MMX instruction resumes execution upon return from the exception handler.

3.3.7. Register Mapping

The IA MMX registers and their tags are mapped to physical locations of the floating-point registers and their tags. Register aliasing and mapping is described in more detail in Section 4.3.1.

4

System Programming Model

CHAPTER 4
SYSTEM PROGRAMMING MODEL

This chapter presents the interface of the Intel Architecture MMX™ technology to the operating system.

4.1. CONTEXT SWITCHING

This section describes the behavior of operating systems during context switching.

Different operating systems take different approaches for state-saving:

- Some operating systems save the entire floating-point state.
- Some save the floating-point state only when it is required.
- Some may save a partial floating-point state.

The existing task switch code for IA implementations (including floating-point code) does not change for systems that include MMX code.

4.1.1. Cooperative Multitasking Operating System

In a cooperative operating system, application tasks can predetermine when it is about to be switched out. Tasks can prepare in advance for the switch.

Application programmers must know whether the operating system performs a state save or whether it is their responsibility to perform a state save.

4.1.2. Preemptive Multitasking Operating System

In a preemptive multitasking operating system, the application cannot know when it is preempted. Applications cannot prepare in advance for task switching. The operating system is responsible for saving and restoring the state when necessary.

The IA MMX technology was defined to support the same state-saving and restoring techniques as the floating-point state-saving and restoring techniques. Existing operating systems can continue to run without modifications.

Figure 4-1 illustrates an example of an operating system implementing floating-point or MMX state saving.

Detecting when to save the FP or MMX state needs to be saved is the same process used for detecting when the floating-point state needs to be saved. If CR0.TS=1 (task switch bit in control register 0), then the next FP or MMX instruction generates exception Int 7.

1. The operating system maintains a *save area* for each task (Save Areas A and B in Figure 4-1).
2. It defines a variable that indicates which task "owns" the FP or MMX state.
3. On a task switch, the OS sets the CR0.TS to 1 if the incoming task does not own the FP or MMX state. Otherwise, it sets it to 0.
4. If a new task attempts to use an MMX instruction, (while CR0.TS=1), exception Int 7 is generated. The Int 7 handler ("owned" by the operating system) saves the FP or MMX state to the save area of the FP or MMX state owner and restores the FP or MMX state from the save area of the current task.
5. The ownership of the FP or MMX state then changes to the current task and CR0.TS=0.

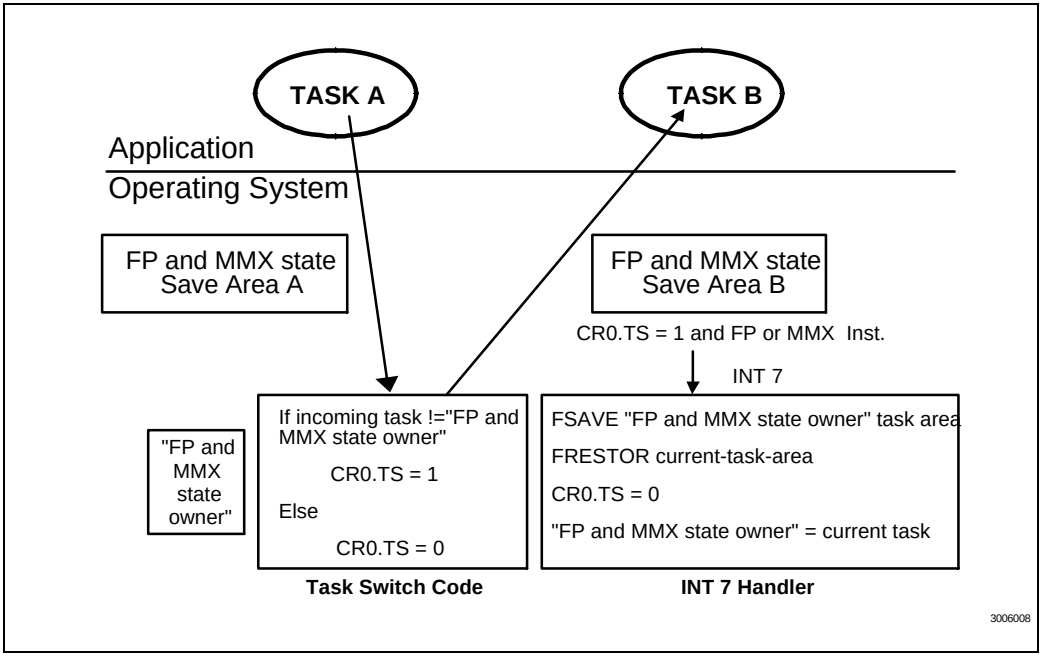


Figure 4-1. Example of FP and MMX State Saving in Operating System

4.2. EXCEPTIONS

MMX instructions do not generate numeric exceptions or affect the processor architecture status flags. Previously pending floating-point numeric errors are reported.

The MMX instructions can generate the following exceptions:

- Memory access exceptions:

  #SS Interrupt 12 - Stack exception

  #GP Interrupt 13 - General Protection

  #PF Interrupt 14 - Page Fault

  #AC Interrupt 17 - Alignment Check, if enabled by CPU configuration.

- System exceptions:

  #UD Interrupt 6 - Invalid Opcode

  Executing an MMX instruction when CR0.EM=1 generates an Invalid Opcode exception.

  #NM Interrupt 7 - Device not available. The TS bit in CR0 is set.

- Pending floating-point error:

  #MF Interrupt 16 - Pending floating-point error

- Other exceptions that occur indirectly due to faulty execution of the above exceptions. For example: Interrupt 12 occurs due to MMX instructions, and the interrupt gate directs the processor to invalid TSS (task state segment).

The MMX instructions are accessible from all operation modes of IA: Protected mode, Real address mode, and Virtual 8086 mode.

4.3. COMPATIBILITY WITH EXISTING SOFTWARE ENVIRONMENTS

4.3.1. Register Aliasing

The MMX state is aliased on the floating-point state:

- MMX registers MM0-MM7 are aliased on the 64-bit mantissas of the floating-point register (See Figure 4-2).
- A value written to an MMX register using MMX instructions also appears in one of the eight floating-point registers (bits 63-0). The exponent field of the corresponding floating-point register (bits 78-64) and its sign bit (bit 79) are set to ones (11s).
- The mantissa of a floating-point value written to a floating-point register by floating-point instructions also appears in an MMX register.

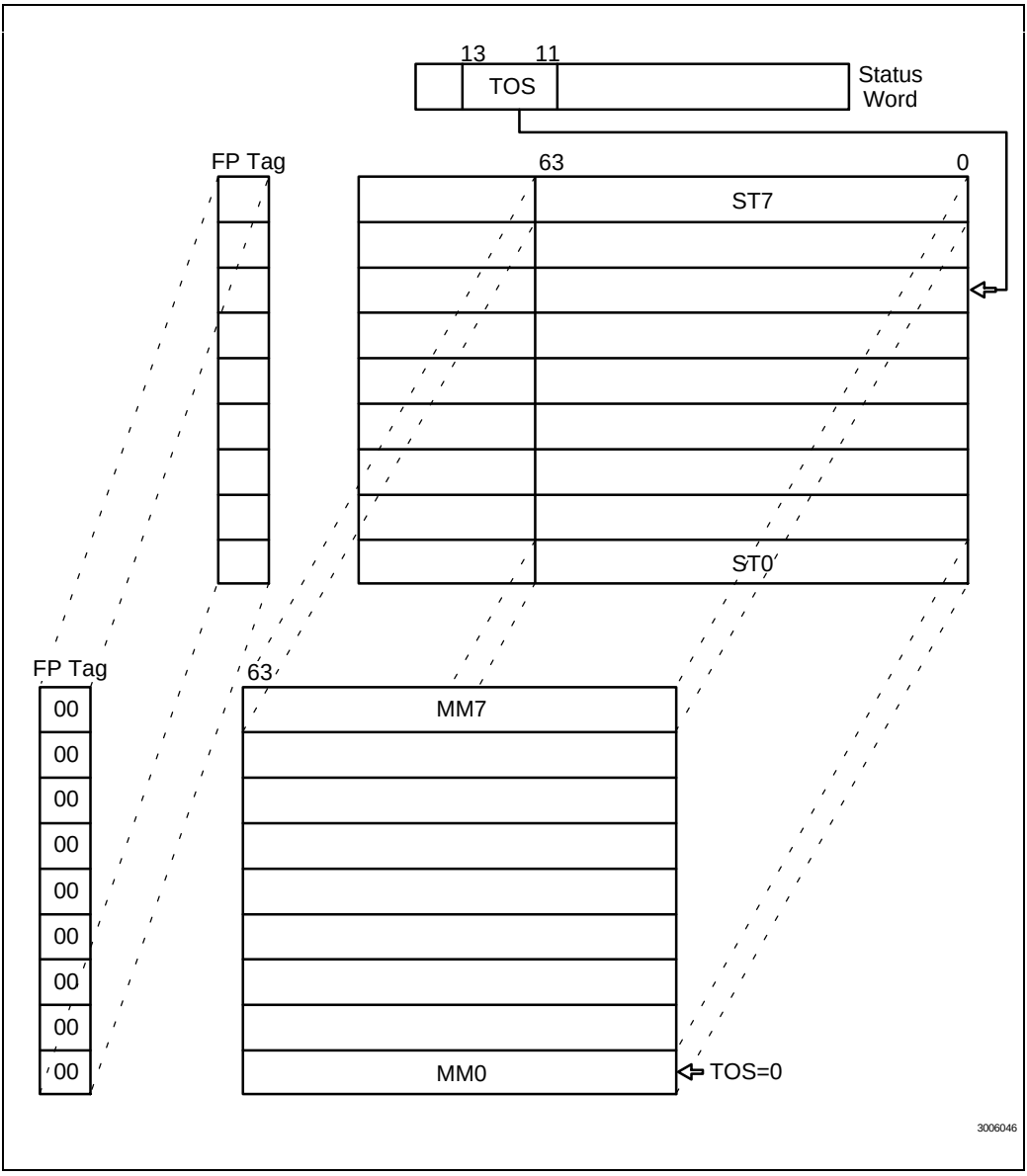


Figure 4-2. Aliasing of MMX™ to Floating-Point Registers

MMX registers map to the physical locations of floating-point registers. MMX register mapping is fixed and does not change when the TOS (Top Of Stack field in the floating-point status word, bits 11-13) changes.

The value of the TOS is set to 0 after each MMX instruction.

In the floating-point context, STn refers to the relative location of a FP register, n, to the TOS. However, the FP tag bits refer to the physical locations of the FP register. The MMX registers always refer to the physical location.

In Figure 4-3, the inner circle refers to the physical location of the FP and MMX registers. The outer circle refers to FP register's relative location to the current TOS.

When the TOS=0 (case a in Figure 4-3), ST0 points to the physical location 0 on the floating-point stack. MM0 maps to ST0, MM1 maps to ST1, and so on.

When the TOS=2 (case b in Figure 4-3), ST0 points to the physical location 2. MM0 maps to ST6, MM1 maps to ST7, MM2 maps to ST0, and so on.

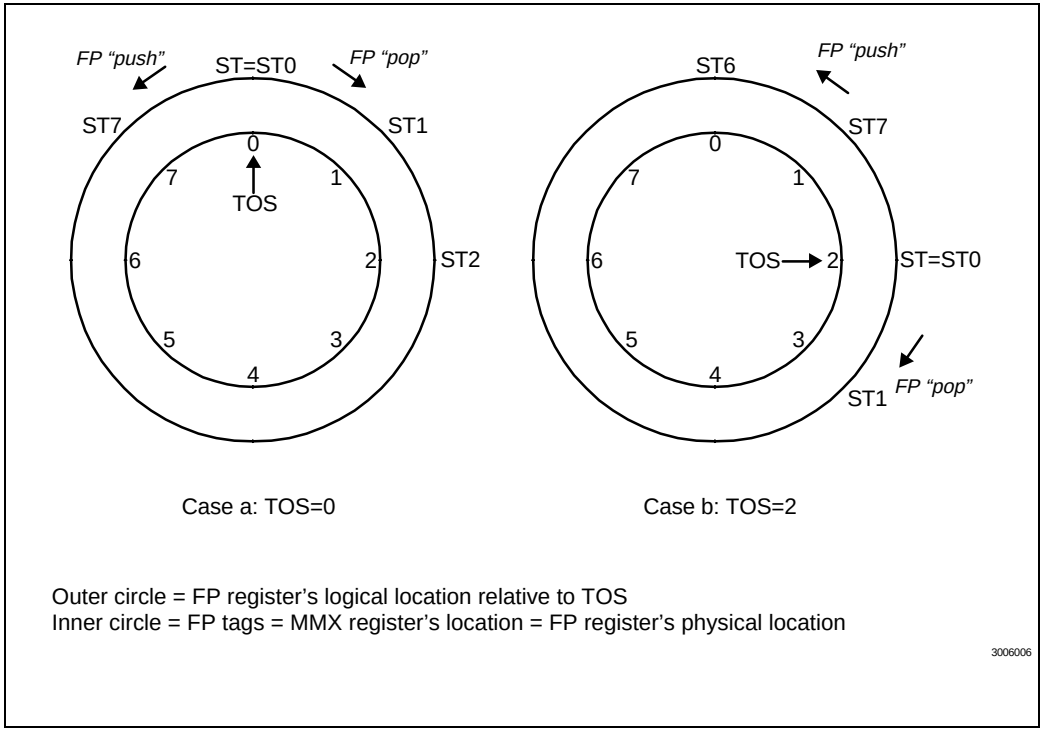


Figure 4-3. Mapping of MMX™ Registers to Floating Point Registers

4.3.2. The Effect of Floating-Point and MMX™ Instructions on the Floating-Point Tag Word

Using an MMX instruction (except EMMS) validates (sets to 00s) the entire floating-point tag word.

The EMMS instruction sets the entire FP tag bits register to empty (11s in each tag field).

FSAVE and FSTENV instructions read the FP tag word and store the contents of the FP tag word in memory. Executing these instructions calculates the precise values of the FP tag word fields based on the current contents of the registers. After executing these instructions, all tag bit values are valid for MMX instructions: Valid, Zero, Special, Empty. The value of the FP tag word does not affect the MMX registers or execution of MMX instructions.

Table 4-1 summarizes the effect of FP or MMX instructions and FSAVE/ FSTENV instructions on the tag bit fields in an FP or MMX register and defines their value in memory.

Table 4-1. Effect of the FP and MMX Instructions on the FP Tag Word

Instruction Type	Instruction	Tag Bits	Calculated FP Tag Word in Memory After FSAVE/FSTENV
MMX™	All (except EMMS)	All registers' tags are set to zeros (00).	00, 01, 10
MMX	EMMS	All registers' tags are set to ones (11).	11
FP	All (except FRSTOR, FLDENV)	Individual register tag is set to 00 or 11.	Each register's tags are set to 00, 11, 01 or 10.
FP	FRSTOR, FLDENV	All registers' tags are set to 00 or 11 or 01 or 10.	Each register's tags are set to 00, 11, 01 or 10.

4.3.2.1. ALIASING SUMMARY

Table 4-2 summarizes the effects of the MMX instructions on the floating-point state.

Table 4-2. Effects of MMX™ Instruction on FP State

Instruction Type	FP Tag Word	TOS ($SW_{13..11}$)	Other FP Environment (CW, Data Ptr, Code Ptr, Other Fields)	Exponent Bits + Signed Bit of MMn ($_{79..64}$)	Mantissa Part of MMn ($_{63..00}$)
MMX register read from MMX register (MMn)	All fields set to 00 (Valid)	000	Unchanged	Unchanged	Unchanged
MMX register write to MMX register (MMn)	All fields set to 00 (Valid)	000	Unchanged	Set to ones (11)	Overwritten
EMMS	All fields set to 11 (Empty)	000	Unchanged	Unchanged	Unchanged

Note: MMn refers to one MMX register.

4.3.3. Context Switch Support

If the task switch bit (TS) in control register 0 (CR0) is set (CR0.TS=1), the first FP or MMX instruction that executes will trigger Int 7, Device not available (DNA). Causing a DNA fault enables an operating system to save the context of the FP or MMX registers with the same code currently used to save the FP state. Both the FSAVE (Store FP state) and FRSTOR (Restore FP state) instructions are used to save and restore either the FP or MMX state.

See Section 4.1. for more details on context switching.

4.3.4. Floating-Point Exceptions

When floating-point exceptions are enabled and a FP exception is pending, subsequent MMX instruction execution reports an FP error (Int 16 and/or FERR# signal). The pending exception is handled by the FP exception handler. Execution resumes at the interrupted MMX instruction.

Before the MMX instruction is executed, the FP state is maintained and is visible to the FP exception handler.

See Section 3.3.6 for more detail.

4.3.5. Debugging

The debug features for Intel Architecture implementations operate in the same manner on the MMX instruction set. This enables debuggers to debug code that uses the MMX technology.

4.3.6. Emulation of the Instruction Set

There is no emulation support for microprocessors that support the MMX technology.

The CR0.EM bit used to emulate floating-point instructions cannot be used in the same way for MMX instruction emulation. If an MMX instruction executes when the CR0.EM bit is set, an invalid opcode exception (Int 6) is generated.

4.3.7. Exception handling in Operating Systems

This section specifies system exceptions. Exception handling in MMX code is discussed in Section 3.3.6.

An invalid opcode exception (Int 6) can occur due to MMX instruction execution two cases:

- On implementations that do not support IA MMX technology.
- When CR0.EM=1 and an MMX instruction is executed.

The CR0.EM bit is used to emulate the FP instructions in software. In this case, the operating system does not save the FP hardware state on task switches and does not save the MMX state. An invalid opcode exception is generated to flag this event to the operating system, and prevent application errors from occurring.

5

Intel Architecture MMX™ Instruction Set

CHAPTER 5
INTEL ARCHITECTURE MMX™ INSTRUCTION SET

This chapter presents the Intel Architecture MMX™ instructions in alphabetical order, with a full description of each instruction.

The IA MMX technology defines fifty-seven new instructions. The instructions are grouped into the following functional categories:

- Arithmetic Instructions
- Comparison Instructions
- Conversion Instructions
- Logical Instructions
- Shift Instructions
- Data Transfer Instructions
- Empty MMX State (EMMS) Instruction

Appendix A summarizes the MMX instructions grouped by categories of related functions. Appendix B provides instruction formats and encodings, and Appendix C provides an alphabetical list of instruction mnemonics, their source data types, encodings in hexadecimal, and format. Appendix D provides an Opcode Map of the MMX instructions.

Many of the instructions have multiple variations depending on the data types they support. Each variation has a different suffix. For example the PADD instruction has three variations: PADDB, PADDW, and PADDD, where the letters B, W, and D represent byte, word, and doubleword.

5.1. INSTRUCTION SYNTAX

Instructions vary by:

- Data type: packed bytes, packed words, packed doublewords or quadwords
- Signed - Unsigned numbers
- Wraparound - Saturate arithmetic

A typical MMX instruction has this syntax:

- Prefix: **P** for Packed
- Instruction operation: for example - ADD, CMP, or XOR
- Suffix:
  - **US** for Unsigned Saturation
  - **S** for Signed saturation
  - **B, W, D, Q** for the data type: packed byte, packed word, packed doubleword, or quadword.

Instructions that have different input and output data elements have two data-type suffixes. For example, the conversion instruction converts from one data type to another. It has two suffixes: one for the original data type and the second for the converted data type.

This is an example of an instruction mnemonic syntax :

PADDUSW (Packed Add Unsigned with Saturation for Word)

P	=	Packed
ADD	=	the instruction operation
US	=	Unsigned Saturation
W	=	Word

5.2. INSTRUCTION FORMAT

The IA MMX instructions use the existing IA instruction format. All instructions, except the EMMS instruction, use the ModR/M format. All are preceded by the 0F prefix byte. For more details about the ModR/M format refer to *Pentium® Processor Family Developer's Manual Volume 3*, Section 25.2.1.

For data-transfer instructions, the destination and source operands can reside in memory, integer registers, or MMX registers. For all other IA MMX instructions, the destination operands reside in MMX registers, and the source operands reside in memory, MMX registers, or immediate operands.

All existing address modes are supported using the SIB (Scale Index Base) format.

5.3. NOTATIONAL CONVENTIONS

The following conventions apply to all MMX instructions (except the EMMS instruction):

- The instructions reference and operate on two operands: the source and destination operands. The right operand is the source and the left operand is the destination. The destination operand may also supply one of the inputs for the operation. The instruction overwrites the destination operand with the result.
- When one of the operands is a memory location, the linear address corresponds to the address of the least significant byte of the referenced memory data.
- The MMX instructions do not affect the condition flags.

5.4. HOW TO READ THE INSTRUCTION SET PAGES

The following is an example of the format used for each MMX instruction description in this chapter:

PSLL—Packed Shift Left Logical

Opcode	Instruction	Description
0F F1 /r	PSLLW *mm, mm/m64*	Shift all words in MMX register to left by an amount specified in MMX register/memory, while shifting in zeros.

The above table gives the instruction mnemonic and a brief description of the mnemonic. The columns content are explained below.

Opcode Column

The "Opcode" column provides the complete opcode produced for each form of the instruction.

The codes are defined as hexadecimal bytes, in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- **/digit:** (digit is between 0 and 7) indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The **reg** field contains the digit that provides a technology to the instruction's opcode.
- **/r**: indicates that the ModR/M byte of the instruction contains both a register operand and an r/m operand.
- **ib**: a 1-byte, immediate operand to the instruction that follows the opcode, ModR/M bytes, and scale-indexing bytes. The opcode determines if the operand is a signed value.

Instruction Column

The "Instruction" column contains the instruction syntax. The following is a list of the symbols used to represent operands in the instruction statements:

- **imm8**: an immediate byte value, imm8 is a signed number between -128 and +127 inclusive.
- **r/m32**: a doubleword register or memory operand used for instructions whose operand-size attribute is 32 bits.

- **mm/m32**: indicates the lowest 32 bits of an MMX register or a 32-bit memory location.
- **mm/m64**: indicates a 64-bit MMX register or a 64-bit memory location.

Description Column

The "Description" column briefly explains the instruction activity.

Operation

The "Operation" section contains an algorithmic description of the operation performed by the instruction.

The register name or memory location implies the contents of the register or memory.

The bit values are written from high-order to low-order and indicate the address within the register or memory. The bit addresses are specified along with the register name or memory location in brackets. For example mm(7..0) represents the low-order 8 bits in an MMX register.

The algorithms are composed of the following elements:

- Comments are enclosed with the symbol pairs “(*” and “*)”.
- Compound statements are enclosed between the keywords of the “if” statement (IF, THEN, ELSE).
- A ← B; indicates that the value of B is assigned to A.
- The symbols =, <>, >, <, ≥, and ≤ are relational operators used to compare two values, meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A=B is TRUE if the value for A is equal to B; otherwise it is FALSE.

The following functions are used in the algorithmic descriptions:

- **ZeroExtend (value)** returns a value zero-extended to the operand-size attribute of the instruction. For example, if OperandSize = 32, ZeroExtend of a byte value of -10 converts the byte from F6H to doubleword with hexadecimal value 000000F6H. If the value passed to ZeroExtend and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.
- **SignExtend (value)** returns a value sign-extended to the operand-size attribute of the instruction. For example, if OperandSize = 32, SignExtend of a byte containing the value -10 converts the byte from F6H to doubleword with hexadecimal value FFFFFFF6H. If the value passed to SignExtend and the operand-size attribute are the same size, SignExtend returns the value unaltered.

- **SaturateSignedWordToSignedByte** converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than -128, it is represented by the saturated value -128 (0x80). If it is greater than 127, it is represented by the saturated value 127 (0x7F).
- **SaturateSignedDwordToSignedWord** converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than -32768, it is represented by the saturated value -32768 (0x8000). If it is greater than 32767, it is represented by the saturated value 32767 (0x7FFF).
- **SaturateSignedWordToUnsignedByte** converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero it is represented by the saturated value zero (0x00).If it is greater than 255 it is represented by the saturated value 255 (0xFF).
- **SaturateToSignedByte** represents the result of an operation as a signed 8-bit value. If the result is less than -128, it is represented by the saturated value -128 (0x80). If it is greater than 127, it is represented by the saturated value 127 (0x7F).
- **SaturateToSignedWord** represents the result of an operation as a signed 16-bit value. If the result is less than -32768, it is represented by the saturated value -32768 (0x8000).If it is greater than 32767, it is represented by the saturated value 32767 (0x7FFF).
- **SaturateToUnsignedByte** represents the result of an operation as a signed 8-bit value. If the result is less than zero it is represented by the saturated value zero (0x00). If it is greater than 255, it is represented by the saturated value 255 (0xFF).
- **SaturateToUnsignedWord** represents the result of an operation as a signed 16-bit value. If the result is less than zero it is represented by the saturated value zero (0x00).I If it is greater than 65535, it is represented by the saturated value 65535 (0xFFFF).

Description

The "Description" section describes the operation for all variations of the instruction.

Example

The “Example” section contains a graphical representation of the instruction's functional behavior.

Exceptions

The "Exceptions" section lists the exceptions in the three different modes: Protected mode, Real Address mode, and Virtual-8086 mode.

Refer to Section 4.2 of this document for more detail on these exceptions. See also the *Pentium® Processor Family Manual, Volume 3*, Section 9.4 and Chapter 14.

EMMS—Empty MMX™ State

Opcode	Instruction	Description
0F 77	EMMS	Set the FP tag word to empty.

Operation

TW ← 0xFFFF;

Description

The EMMS instruction sets the values of the floating-point (FP) tag word to empty (all ones). EMMS marks the registers as available, so they can subsequently be used by floating-point instructions.

If a floating-point instruction loads into one of the registers before it has been reset by the EMMS instruction, a floating-point stack overflow can occur, which results in an FP exception or incorrect result.

All other MMX instructions validate the entire FP tag word (all zeros).

NOTE

This instruction must be used to clear the MMX state at the end of all MMX routines, and before calling other routines that may execute floating-point instructions.

Figure 5-1 shows the format of the FP Tag Word.

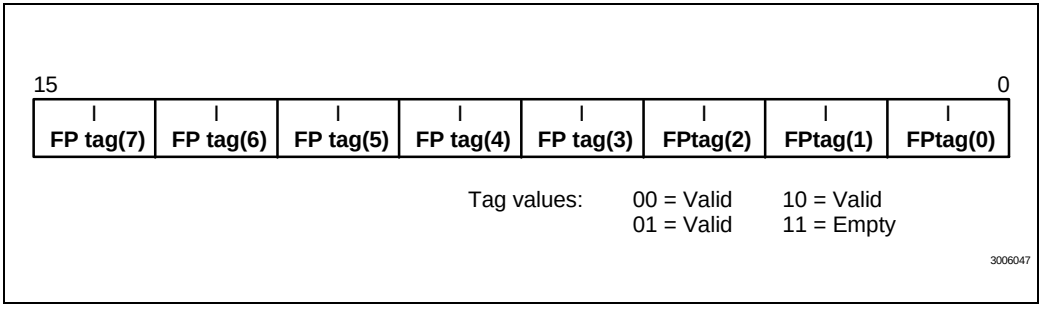


Figure 5-1. Floating Point Tag Word Format

Flags Affected

None.

Protected Mode Exceptions

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

MOVD—Move 32 Bits

Opcode	Instruction	Description
0F 6E /r	MOVD *mm, r/m32*	Move 32 bits from integer register/memory to MMX register.
0F 7E /r	MOVD *r/m32, mm*	Move 32 bits from MMX register to integer register/memory.

Operation

```
IF destination = mm
THEN
   mm(63..0) ← ZeroExtend(r/m32);
ELSE
   r/m32 ← mm(31..0);
```

Description

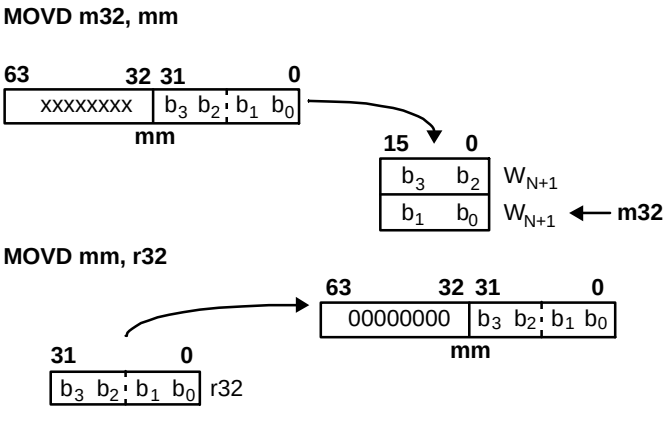
The MOVD instruction copies 32 bits from the source operand to the destination operand.

The destination and source operands can be either MMX registers, 32-bit memory operands, or 32-bit integer registers. The MOVD cannot transfer data from an MMX register to an MMX register, from memory to memory, or from an integer register to an integer register.

When the destination operand is an MMX register, the 32-bit source operand is written to the low-order 32 bits of the 64-bit destination register. The destination register is zero-extended to 64 bits.

When the source operand is an MMX register, the low-order 32 bits of the MMX register are written to the 32-bit integer register or 32-bit memory location.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

MOVQ—Move 64 Bits

Opcode	Instruction	Description
0F 6F /r	MOVQ *mm, mm/m64*	Move 64 bits from MMX register/memory to MMX register.
0F 7F /r	MOVQ *mm/m64, mm*	Move 64 bits from MMX register to MMX register/memory.

Operation

```
IF destination = mm
THEN
   mm ← mm/m64;
ELSE
   mm/m64 ← mm;
```

Description

The MOVQ instruction copies 64 bits from the source operand to the destination operand.

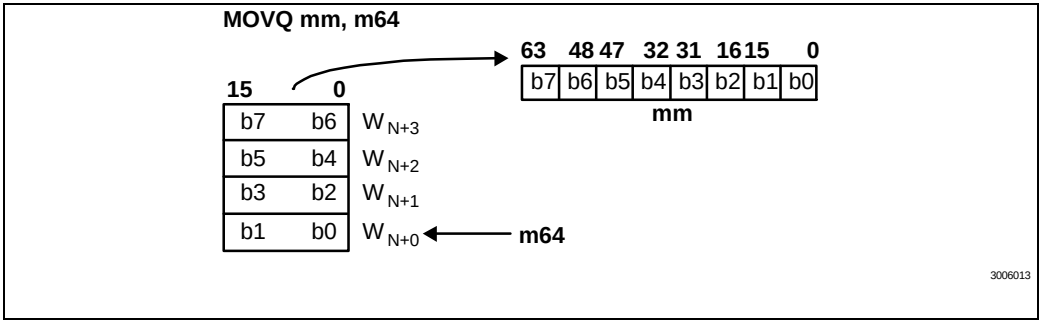
The destination and source operands can be either MMX registers or 64-bit memory operands. The MOVQ instruction cannot transfer data from memory to memory.

When the destination is an MMX register and the source is a 64-bit memory operand, the 64 bits of data at the memory location are copied into the MMX register.

When the destination is a 64-bit memory operand and the source is an MMX register, the 64 bits of data are copied from the MMX register into the memory location.

When the destination and source are both MMX registers, the contents of the MMX register (source) are copied into an MMX register (destination).

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PACKSSWB /PACKSSDW—Pack with Signed Saturation

Opcode	Instruction	Description
0F 63 /r	PACKSSWB *mm, mm/m64*	Pack and saturate signed words from MMX register and MMX register/memory into signed bytes in MMX register.
0F 6B /r	PACKSSDW *mm, mm/m64*	Pack and saturate signed dwords from MMX register and MMX register/memory into signed words in MMX register.

Operation

```
IF instruction is PACKSSWB
THEN {
   mm(7..0) ← SaturateSignedWordToSignedByte mm(15..0);
   mm(15..8) ← SaturateSignedWordToSignedByte mm(31..16);
   mm(23..16) ← SaturateSignedWordToSignedByte mm(47..32);
   mm(31..24) ← SaturateSignedWordToSignedByte mm(63..48);
   mm(39..32) ← SaturateSignedWordToSignedByte mm/m64(15..0);
   mm(47..40) ← SaturateSignedWordToSignedByte mm/m64(31..16);
   mm(55..48) ← SaturateSignedWordToSignedByte mm/m64(47..32);
   mm(63..56) ← SaturateSignedWordToSignedByte mm/m64(63..48);
   }
ELSE { (* instruction is PACKSSDW *)
   mm(15..0) ← SaturateSignedDwordToSignedWord mm(31..0);
   mm(31..16) ← SaturateSignedDwordToSignedWord mm(63..32);
   mm(47..32) ← SaturateSignedDwordToSignedWord mm/m64(31..0);
   mm(63..48) ← SaturateSignedDwordToSignedWord mm/m64(63..32);
   }
```

Description

The PACKSS instruction packs and saturates the signed data elements from the source and the destination operands and writes the signed results to the destination operand.

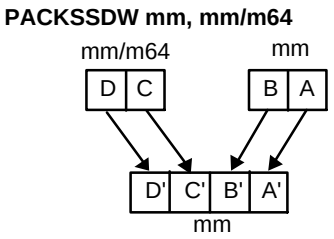
The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

The PACKSSWB instruction packs four signed words from the source operand and four signed words from the destination operand into eight signed bytes in the destination register. If the signed value of a word is larger or smaller than the range of a signed byte, the value is saturated (in the case of an overflow - to 0x7F, and in the case of an underflow - to 0x80).

The PACKSSDW instruction packs two signed doublewords from the source operand and two signed doublewords from the destination operand into four signed words in the destination register. If the signed value of a doubleword is larger or smaller than the range of

a signed word, the value is saturated (in the case of an overflow - to 0x7FFF, and in the case of an underflow - to 0x8000).

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PACKUSWB—Pack with Unsigned Saturation

Opcode	Instruction	Description
0F 67 /r	PACKUSWB *mm, mm/m64*	Pack and saturate signed words from MMX register and MMX register/memory into unsigned bytes in MMX register.

Operation

```
mm(7..0) ← SaturateSignedWordToUnsignedByte mm(15..0);
mm(15..8)← SaturateSignedWordToUnsignedByte mm(31..15);
mm(23..16) ← SaturateSignedWordToUnsignedByte mm(47..32);
mm(31..24) ← SaturateSignedWordToUnsignedByte mm(63..48);
mm(39..32) ← SaturateSignedWordToUnsignedByte mm/m64(15..0);
mm(47..40) ← SaturateSignedWordToUnsignedByte mm/m64(31..16);
mm(55..48) ← SaturateSignedWordToUnsignedByte mm/m64(47..32);
mm(63..56) ← SaturateSignedWordToUnsignedByte mm/m64(63..48);
```

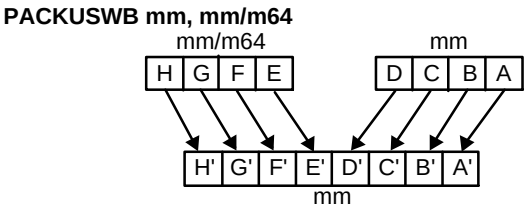
Description:

The PACKUSWB packs and saturates four signed words of the source operand and four signed words of the destination operand into eight unsigned bytes. The result is written to the destination operand

The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

If the signed value of the word is larger or smaller than the range of an unsigned byte, the value is saturated (in the case of an overflow - to 0xFF and in the case of an underflow - to 0x00).

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PADDB/PADDW/PADDD—Packed Add

Opcode	Instruction	Description
0F FC /r	PADDB *mm, mm/m64*	Add packed byte from MMX register/memory to packed byte in MMX register.
0F FD /r	PADDW *mm, mm/m64*	Add packed word from MMX register/memory to packed word in MMX register.
0F FE /r	PADDD *mm, mm/m64*	Add packed dword from MMX register/memory to packed dword in MMX register.

Operation

```
IF instruction is PADDB
THEN  {
   mm(7..0) ← mm(7..0) + mm/m64(7..0);
   mm(15..8) ← mm(15..8) + mm/m64(15..8);
   mm(23..16) ← mm(23..16)+ mm/m64(23..16);
   mm(31..24) ← mm(31..24) + mm/m64(31..24);
   mm(39..32) ← mm(39..32) + mm/m64(39..32);
   mm(47..40) ← mm(47..40)+ mm/m64(47..40);
   mm(55..48) ← mm(55..48) + mm/m64(55..48);
   mm(63..56) ← mm(63..56) + mm/m64(63..56);
   }
IF instruction is PADDW
THEN {
   mm(15..0) ← mm(15..0) + mm/m64(15..0);
   mm(31..16) ← mm(31..16) + mm/m64(31..16);
   mm(47..32) ← mm(47..32) + mm/m64(47..32);
   mm(63..48) ← mm(63..48) + mm/m64(63..48);
   }
ELSE { (* instruction is PADDD *)
   mm(31..0) ← mm(31..0) + mm/m64(31..0);
   mm(63..32) ← mm(63..32) + mm/m64(63..32);
   }
```

Description

The PADD instructions add the data elements of the source operand to the data elements of the destination register. The result is written to the destination register. If the result exceeds the data-range limit for the data type, it wraps around.

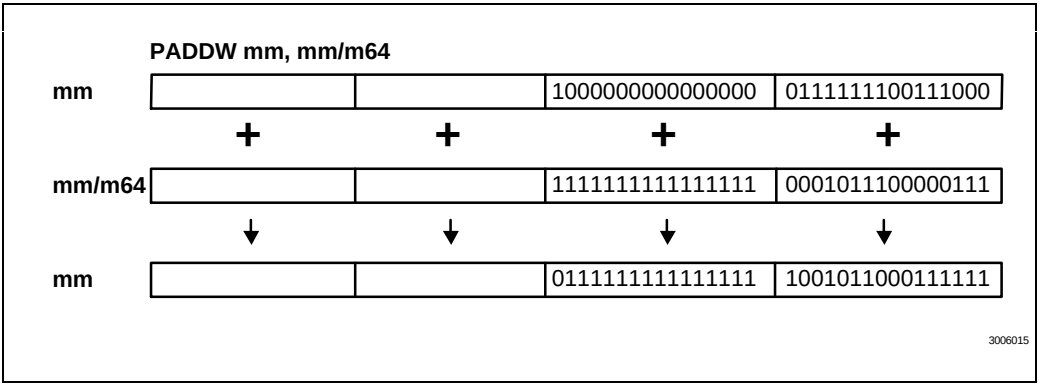
The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

The PADDB instruction adds the bytes of the source operand to the bytes of the destination operand and writes the results to the MMX register. When the result is too large to be represented in a packed byte (overflow), the result wraps around and the lower 8 bits are written to the destination register.

The PADDW instruction adds the words of the source operand to the words of the destination operand and writes the results to the MMX register. When the result is too large to be represented in a packed word (overflow), the result wraps around and the lower 16 bits are written to the destination register.

The PADDD instruction adds the doublewords of the source operand to the doublewords of the destination operand and writes the results to the MMX register. When the result is too large to be represented in a packed doubleword (overflow), the result wraps around and the lower 32 bits are written to the destination register.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PADDSB/PADDSW —Packed Add with Saturation

Opcode	Instruction	Description
0F EC /r	PADDSB mm, mm/m64	Add signed packed byte from MMX register/memory to signed packed byte in MMX register and saturate.
0F ED /r	PADDSW mm, mm/m64	Add signed packed word from MMX register/memory to signed packed word in MMX register and saturate.

Operation

```
IF instruction is PADDSB
THEN{
   mm(7..0) ← SaturateToSignedByte (mm(7..0) + mm/m64 (7..0)) ;
   mm(15..8) ← SaturateToSignedByte (mm(15..8) + mm/m64(15..8) );
   mm(23..16) ← SaturateToSignedByte (mm(23..16)+ mm/m64(23..16) );
   mm(31..24) ← SaturateToSignedByte (mm(31..24) + mm/m64(31..24) );
   mm(39..32) ← SaturateToSignedByte (mm(39..32) + mm/m64(39..32) );
   mm(47..40) ← SaturateToSignedByte (mm(47..40)+ mm/m64(47..40) );
   mm(55..48) ← SaturateToSignedByte (mm(55..48) + mm/m64(55..48) );
   mm(63..56) ← SaturateToSignedByte (mm(63..56) + mm/m64(63..56) );
   }
ELSE { (* instruction is PADDSW *)
   mm(15..0) ← SaturateToSignedWord (mm(15..0) + mm/m64(15..0) );
   mm(31..16) ← SaturateToSignedWord (mm(31..16) + mm/m64(31..16) );
   mm(47..32) ← SaturateToSignedWord (mm(47..32) + mm/m64(47..32) );
   mm(63..48) ← SaturateToSignedWord (mm(63..48) + mm/m64(63..48) );
   }
```

Description

The PADDS instructions add the packed signed data elements of the source operand to the packed signed data elements of the destination operand and saturate the result. The result is written to the destination operand.

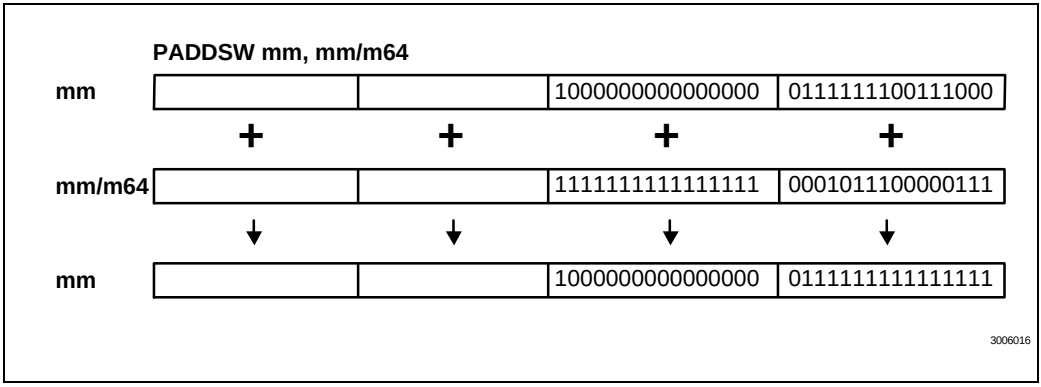
The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

The PADDSB instruction adds the signed bytes of the source operand to the signed bytes of the destination operand and writes the results to the MMX register. If the result is larger or smaller than the range of a signed byte, the value is saturated (in the case of an overflow - to 0x7F, and in the case of an underflow - to 0x80).

The PADDSW instruction adds the signed words of the source operand to the signed words of the destination operand and writes the results to the MMX register. If the result is larger or

smaller than the range of a signed word, the value is saturated (in the case of an overflow - to 0x7FFF, and in the case of an underflow - to 0x8000) .

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PADDUSB/PADDUSW—Packed Add Unsigned with Saturation

Opcode	Instruction	Description
0F DC /r	PADDUSB *mm, mm/m64*	Add unsigned packed byte from MMX register/memory to unsigned packed byte in MMX register and saturate.
0F DD /r	PADDUSW *mm, mm/m64*	Add unsigned packed word from MMX register/memory to unsigned packed word in MMX register and saturate.

Operation

```
IF instruction is PADDUSB
THEN{
   mm(7..0) ← SaturateToUnsignedByte (mm(7..0) + mm/m64 (7..0) );
   mm(15..8) ← SaturateToUnsignedByte (mm(15..8) + mm/m64(15..8) );
   mm(23..16) ← SaturateToUnsignedByte (mm(23..16)+ mm/m64(23..16) );
   mm(31..24) ← SaturateToUnsignedByte (mm(31..24) + mm/m64(31..24) );
   mm(39..32) ← SaturateToUnsignedByte (mm(39..32) + mm/m64(39..32) );
   mm(47..40) ← SaturateToUnsignedByte (mm(47..40)+ mm/m64(47..40) );
   mm(55..48) ← SaturateToUnsignedByte (mm(55..48) + mm/m64(55..48) );
   mm(63..56) ← SaturateToUnsignedByte (mm(63..56) + mm/m64(63..56) );
   }
ELSE { (* instruction is PADDUSW *)
   mm(15..0) ← SaturateToUnsignedWord (mm(15..0) + mm/m64(15..0) );
   mm(31..16) ← SaturateToUnsignedWord (mm(31..16) + mm/m64(31..16) );
   mm(47..32) ← SaturateToUnsignedWord (mm(47..32) + mm/m64(47..32) );
   mm(63..48) ← SaturateToUnsignedWord (mm(63..48) + mm/m64(63..48) );
   }
```

Description

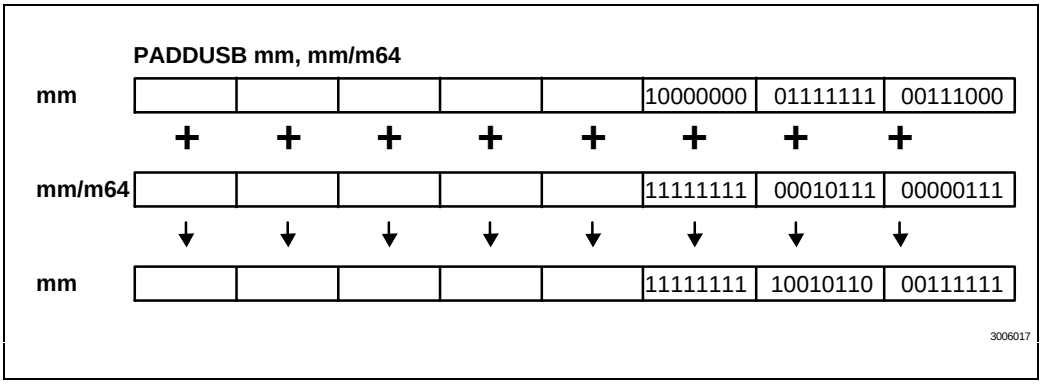
The PADDUS instructions add the packed unsigned data elements of the source operand to the packed unsigned data elements of the destination operand and saturate the results. The results are written to the destination operand.

The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

The PADDUSB instruction adds the unsigned bytes of the source operand to the unsigned bytes of the destination operand and writes the results to the MMX register. When the result is larger than the range of an unsigned byte (overflow), the value is saturated to 0xFF. When the result is smaller than the range of an unsigned byte (underflow), the value is saturated to 0x00.

The PADDUSW instruction adds the unsigned words of the source operand to the unsigned words of the destination operand and writes the results to the MMX register. When the result is larger than the range of an unsigned word (overflow), the value is saturated to 0xFFFF. When the result is smaller than the range of an unsigned word (underflow), the value is saturated to 0x0000.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PAND—Bitwise Logical And

Opcode	Instruction	Description
0F DB /r	PAND *mm, mm/m64*	AND 64 bits from MMX register/memory to MMX register.

Operation

mm ←mm AND mm/m64;

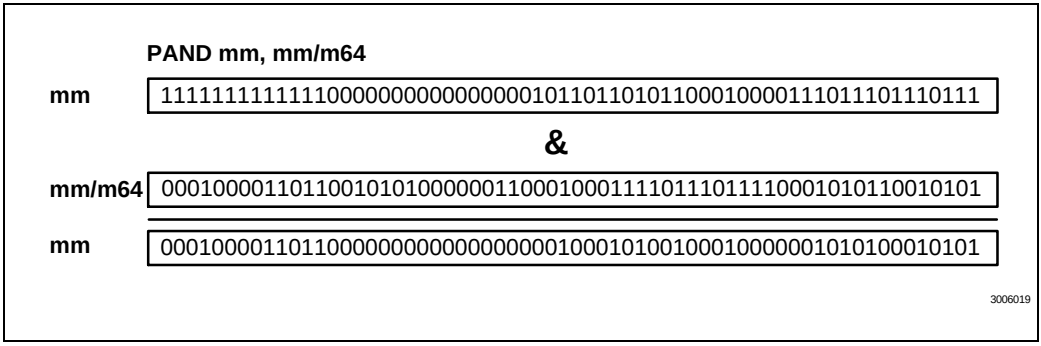
Description

The PAND instruction performs a bitwise logical AND on 64 bits of the source and destination operands, and writes the result to the destination operand.

Each bit of the result of the PAND instruction is set to 1 if the corresponding bits of the operands are 1. Otherwise, it is set to 0.

The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault;

#AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PAND—Bitwise Logical And

Opcode	Instruction	Description
0F DB /r	PAND mm, mm/m64	AND 64 bits from MMX register/memory to MMX register.

Operation

mm ←mm AND mm/m64;

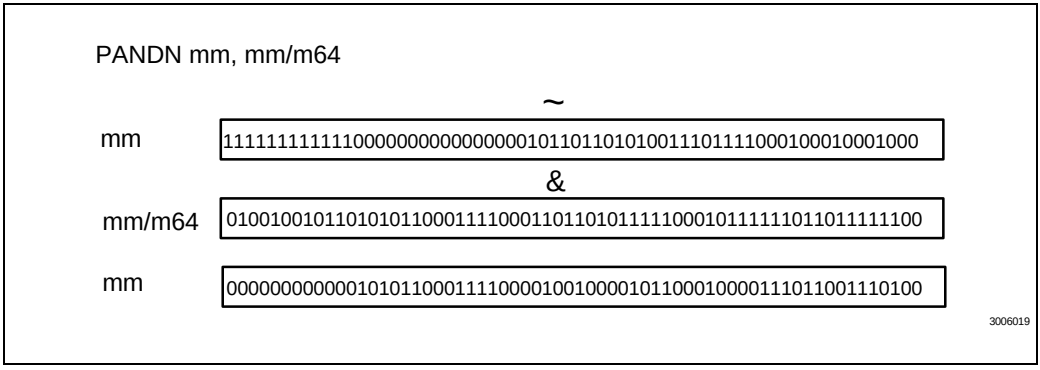
Description

The PAND instruction performs a bitwise logical AND on 64 bits of the source and destination operands, and writes the result to the destination operand.

Each bit of the result of the PAND instruction is set to 1 if the corresponding bits of the operands are 1. Otherwise, it is set to 0.

The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault;

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PCMPEQB/PCMPEQW/PCMPEQD—Packed Compare for Equal

Opcode	Instruction	Description
0F 74 /r	PCMPEQB *mm, mm/m64*	Compare packed byte in MMX register/memory with packed byte in MMX register for equality.
07, 75, /r	PCMPEQW *mm, mm/m64*	Compare packed word in MMX register/memory with packed word in MMX register for equality.
07, 76, /r	PCMPEQD *mm, mm/m64*	Compare packed dword in MMX register/memory with packed dword in MMX register for equality.

Operation

```
IF instruction is PCMPEQB
THEN {
   IF mm(7..0) = mm/m64(7..0)
   THEN mm(7  0) ← 0xFF;
   ELSE mm(7..0) ← 0;
   IF mm(15..8) = mm/m64(15.. 8)
   THEN mm(15..8) ← 0xFF;
   ELSE mm(15..8) ← 0;
   ...
   IF mm(63..56) = mm/m64(63..56)
   THEN mm(63..56) ← 0xFF;
   ELSE mm(63..56) ← 0;
   }
ELSE IF instruction is PCMPEQW
THEN {
   IF mm(15..0) = mm/m64(15..0)
   THEN mm(15..0) ← 0xFFFF;
   ELSE mm(15..0) ← 0;
   IF mm(31..16) = mm/m64(31..16)
   THEN mm(31..16) ← 0xFFFF;
   ELSE mm(31..16) ← 0;
   ...
   IF mm(63..48) = mm/m64(63..48)
   THEN mm(63..48) ← 0xFFFF;
   ELSE mm(63..48) ← 0;
   }
```

```
ELSE {     (* instruction is PCMPEQD *)
   IF mm(31..0) = mm/m64(31..0)
   THEN mm(31..0) ← 0xFFFFFFFF;
   ELSE mm(31..0) ← 0;
   IF mm(63..32) = mm/m64(63..32)
   THEN mm(63..32) ← 0xFFFFFFFF;
   ELSE mm(63..32) ← 0;
   }
```

Description

The PCMPEQ instructions compare the data elements in the destination operand to the corresponding data elements in the source operand. If the data elements are equal, the corresponding data element in the destination register is set to all ones. If they are not equal, the corresponding data element is set to all zeros.

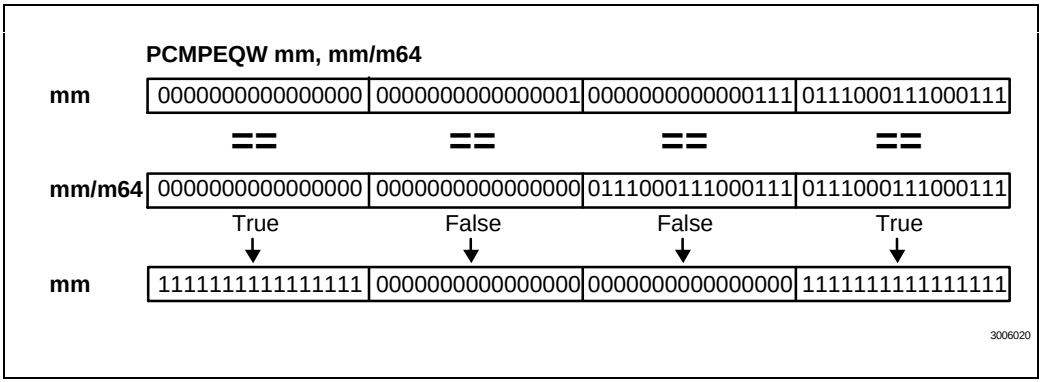
The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

The PCMPEQB instruction compares the bytes in the destination operand to the bytes in the source operand. The bytes in the destination operand are set accordingly.

The PCMPEQW instruction compares the words in the destination operand to the words in the source operand. The words in the destination operand are set accordingly.

The PCMPEQD instruction compares the doublewords in the destination operand to the doublewords in the source operand. The doublewords in the destination operand are set accordingly.

Example



Flags Affected

None:

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PCMPGTB/PCMPGTW/PCMPGTD—Packed Compare for Greater Than

Opcode	Instruction	Description
0F 64 /r	PCMPGTB *mm, mm/m64*	Compare packed byte in MMX register with packed byte in MMX register/memory for greater value.
0F 65 /r	PCMPGTW *mm, mm/m64*	Compare packed word in MMX register with packed word in MMX register/memory for greater value.
0F 66 /r	PCMPGTD *mm, mm/m64*	Compare packed dword in MMX register with packed dword in MMX register/memory for greater value.

Operation

```
IF instruction is PCMPGTB
THEN {
   IF mm(7..0) > mm/m64(7..0)
   THEN mm(7  0) ← 0xFF;
   ELSE mm(7..0) ← 0;
   IF mm(15..8) > mm/m64(15.. 8)
   THEN mm(15..8) ← 0xFF;
   ELSE mm(15..8) ← 0;
   ...
   IF mm(63..56) > mm/m64(63..56)
   THEN mm(63..56) ← 0xFF;
   ELSE mm(63..56) ← 0;
   }
 ELSE IF instruction is PCMPGTW
THEN {
   IF mm(15..0) > mm/m64(15..0)
   THEN mm(15..0) ← 0xFFFF;
   ELSE mm(15..0) ←0;
   IF mm(31..16) > mm/m64(31..16)
   THEN mm(31..16) ← 0xFFFF;
   ELSE mm(31..16) ← 0;
   ...
   IF mm(63..48) > mm/m64(63..48)
   THEN mm(63..48) ← 0xFFFF;
   ELSE mm(63..48) ← 0;
   }
ELSE { (* instruction is PCMPGTD *)
   IF mm(31..0) > mm/m64(31..0)
   THEN mm(31..0) ← 0xFFFFFFFF;
   ELSE mm(31..0) ← 0;
   IF mm(63..32) > mm/m64(63..32)
```

```
THEN mm(63..32) ← 0xFFFFFFFF;
ELSE mm(63..32) ← 0;
}
```

Description

The PCMPGT instructions compare the signed data elements in the destination operand to the signed data elements in the source operand. If the signed data elements in the destination register are greater than those in the source operand, the corresponding data element in the destination operand is set to all ones. Otherwise, it is set to all zeros.

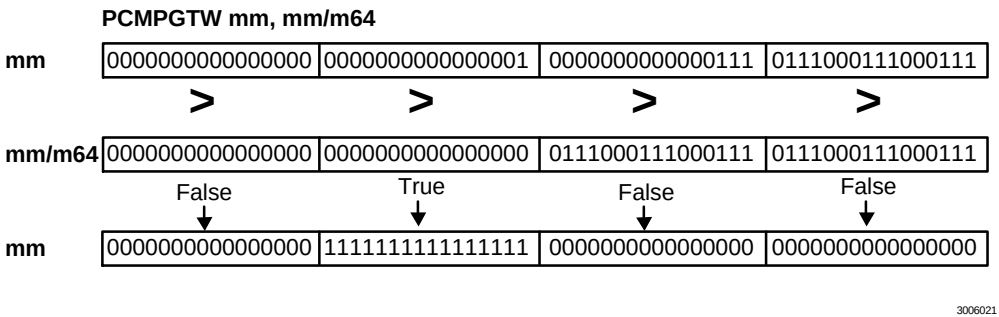
The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

The PCMPGTB instruction compares the signed bytes in the destination operand to the corresponding signed bytes in the source operand. The bytes in the destination register are set accordingly.

The PCMPGTW instruction compares the signed words in the destination operand to the corresponding signed words in the source operand. The words in the destination register are set accordingly.

The PCMPGTD instruction compares the signed doublewords in the destination operand to the corresponding signed doublewords in the source operand. The doublewords in the destination register are set accordingly.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PMADDWD—Packed Multiply and Add

Opcode	Instruction	Description
0F F5 /r	PMADDWD *mm, mm/m64*	Multiply the packed word in MMX register by the packed word in MMX reg/memory. Add the 32-bit results pairwise and store in MMX register as dword

Operation

mm(31..0) ← mm(15..0) * mm/m64(15..0) + mm(31..16) * mm/m64(31..16);
mm(63..32) ← mm(47..32) * mm/m64(47..32) + mm(63..48) * mm/m64(63..48);

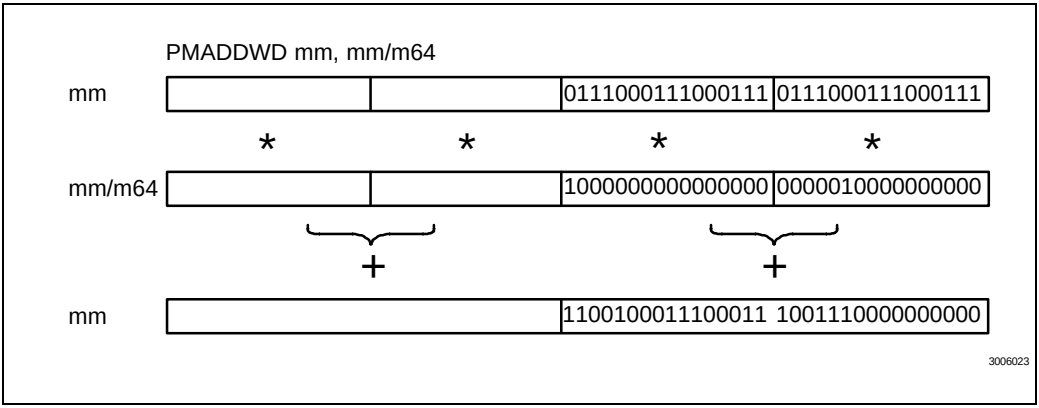
Description

The PMADDWD instruction multiplies the four signed words of the destination operand by the four signed words of the source operand. The result is two 32-bit doublewords. The two high-order words are summed and stored in the upper doubleword of the destination operand. The two low-order words are summed and stored in the lower doubleword of the destination operand. This result is written to the destination operand.

The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

The PMADDWD instruction wraps around to 0x80000000 only when all four words of both the source and destination operands are 0x8000.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PMULHW—Packed Multiply High

Opcode	Instruction	Description
0F E5 /r	PMULHW *mm, mm/m64*	Multiply the signed packed word in MMX register with the signed packed word in MMX reg/memory, then store the high-order 16 bits of the results in MMX register.

Operation

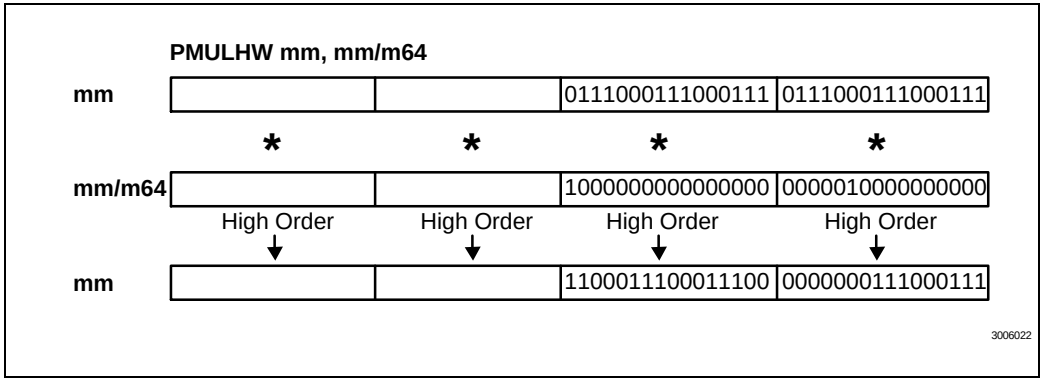
mm(15..0) ← (mm(15..0) * mm/m64(15..0)) (31..16);
mm(31..16)← (mm(31..16) * mm/m64(31..16)) (31..16);
mm(47..32) ← (mm(47..32) * mm/m64(47..32)) (31..16);
mm(63..48) ← (mm(63..48) * mm/m64(63..48)) (31..16);

Description

The PMULHW instruction multiplies the four signed words of the destination operand with the four signed words of the source operand. The high-order 16 bits of the 32-bit intermediate results are written to the destination operand.

The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PMULLW—Packed Multiply Low

Opcode	Instruction	Description
0F D5 /r	PMULLW *mm, mm/m64*	Multiply the packed word in MMX register with the packed word in MMX reg/memory, then store the low-order 16 bits of the results in MMX register.

Operation

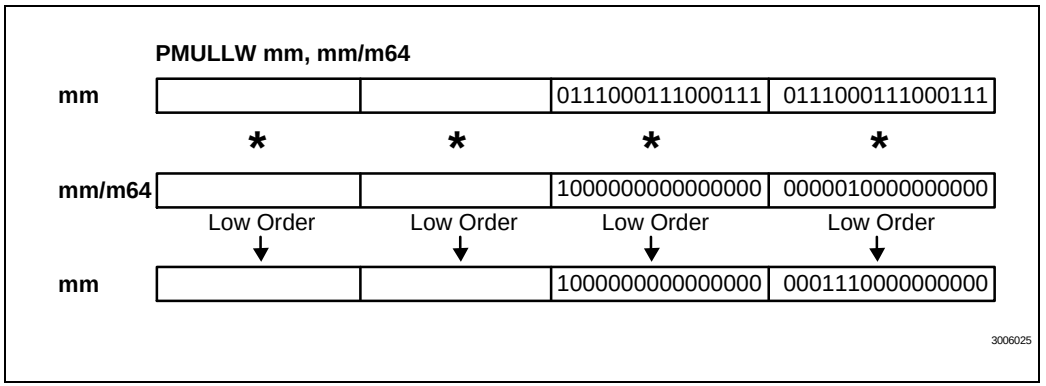
mm(15..0) ← (mm(15..0) * mm/m64(15..0)) (15..0);
mm(31..16) ← (mm(31..16) * mm/m64(31..16)) (15..0);
mm(47..32) ← (mm(47..32) * mm/m64(47..32)) (15..0);
mm(63..48) ← (mm(63..48) * mm/m64(63..48)) (15..0);

Description

The PMULLW instruction multiplies the four signed or unsigned words of the destination operand with the four signed or unsigned words of the source operand. The low-order 16 bits of the 32-bit intermediate results are written to the destination operand.

The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

POR—Bitwise Logical Or

Opcode	Instruction	Description
0F EB /r	POR *mm, mm/m64*	OR 64 bits from MMX reg/memory with MMX register.

Operation

mm ←mm OR mm/m64;

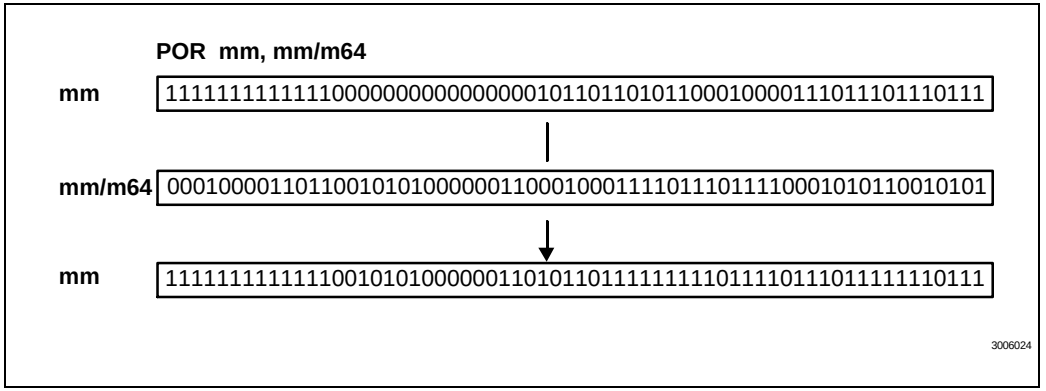
Description

The POR instruction performs a bitwise logical OR on 64 bits of the destination and source operands, and writes the result to the destination register.

Each bit of the result is set to 0 if the corresponding bits of the two operands are 0. Otherwise, the bit is 1.

The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PSLLW/PSLLD/PSLLQ—Packed Shift Left Logical

Opcode	Instruction	Description
0F F1 /r	PSLLW *mm, mm/m64*	Shift words in MMX register left by amount specified in MMX reg/memory, while shifting in zeros.
0F 71 /6, ib	PSLLW *mm, imm8*	Shift words in MMX register left by Imm8, while shifting in zeros.
0F F2 /r	PSLLD *mm, mm/m64*	Shift dwords in MMX register left by amount specified in MMX reg/memory, while shifting in zeros.
0F 72 /6 ib	PSLLD *mm, imm8*	Shift dwords in MMX register by Imm8, while shifting in zeros..
0F F3 /r	PSLLQ *mm, mm/m64*	Shift MMX register left by amount specified in MMX reg/memory, while shifting in zeros.
0F 73 /6 ib	PSLLQ *mm, imm8*	Shift MMX register left by Imm8, while shifting in zeros.

Operation

```
IF the second operand is imm8
THEN
   temp ← imm8;
ELSE (* second operand is mm/m64 *)
   temp ← mm/m64;
IF instruction is PSLLW
THEN {
   mm(15..0) ← mm(15..0) << temp;
   mm(31..16) ← mm(31..16) << temp;
   mm(47..32) ← mm(47..32) << temp;
   mm(63..48) ← mm(63..48) << temp;
   }
ELSE IF instruction is PSLLD
THEN {
   mm(31..0) ← mm(31..0) << temp;
   mm(63..32) ← mm(63..32) << temp;
   }
ELSE  (* instruction is PSLLQ *)
   mm ← mm << temp;
```

Description

The PSLL instructions shift the bits of the first operand to the left by the amount of bits specified in the count operand. The result of the shift operation is written to the destination register. The empty low-order bits are cleared (set to zero). If the value specified by the second operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords), then the destination is set to all zeros.

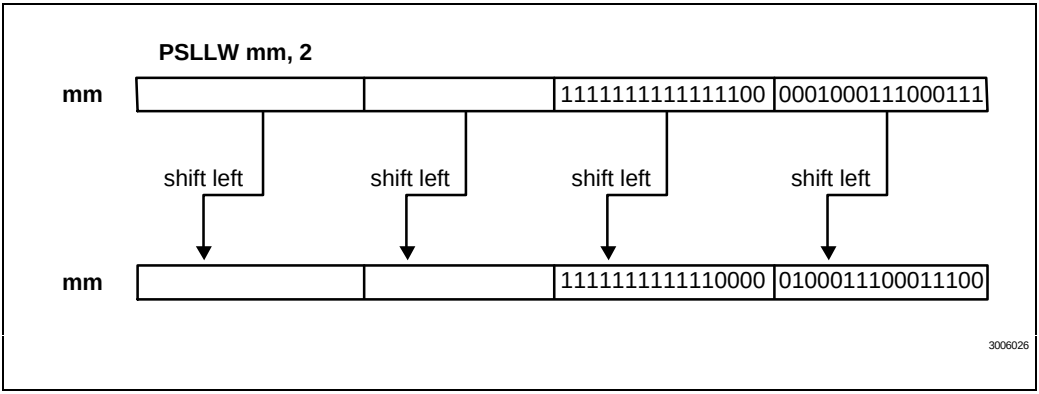
The destination operand is an MMX register. The count operand (source operand) can be either an MMX register, a 64-bit memory operand, or an immediate 8-bit operand.

The PSLLW instruction shifts each of the four words of the destination register to the left by the number of bits specified in the count operand. The low-order bit positions (of each word) are filled with zeros.

The PSLLD instruction shifts each of the two doublewords of the destination register to the left by the number of bits specified in the count operand. The low-order bit positions (of each doubleword) are filled with zeros.

The PSLLQ instruction shifts the 64-bit quadword in the destination register to the left by the number of bits specified in the count operand. The low-order bit positions are filled with zeros.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PSRAW/PSRAD—Packed Shift Right Arithmetic

Opcode	Instruction	Description
0F E1 /r	PSRAW *mm, mm/m64*	Shift words in MMX register right by amount specified in MMX reg/memory while shifting in sign bits.
0F 71 /4 ib	PSRAW *mm, imm8*	Shift words in MMX register right by Imm8 while shifting in sign bits
0F E2 /r	PSRAD *mm, mm/m64*	Shift dwords in MMX register right by amount specified in MMX reg/memory while shifting in sign bits.
0F 72 /4 ib	PSRAD *mm, imm8*	Shift dwords in MMX register right by Imm8 while shifting in sign bits.

Operation

```
IF the second operand is imm8
THEN
   temp ← imm8;
ELSE (* second operand is mm/m64 *)
   temp ← mm/m64;
IF instruction is PSRAW
THEN {
   mm(15..0) ← SignExtend (mm(15..0) >>temp);
   mm(31..16) ← SignExtend (mm(31..16) >> temp);
   mm(47..32) ← SignExtend (mm(47..32) >> temp);
   mm(63..48) ← SignExtend (mm(63..48) >> temp);
   }
ELSE { (*instruction is PSRAD *)
   mm(31..0) ← SignExtend (mm(31..0) >> temp);
   mm(63..32) ← SignExtend (mm(63..32) >> temp);
   }
```

Description

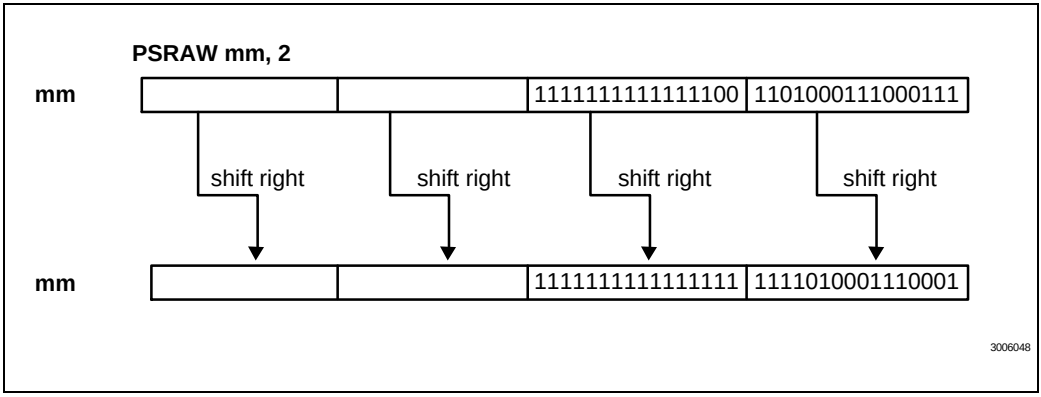
The PSRA instructions shift the bits of the first operand to the right by the amount of bits specified in the count operand. The result of the shift operation is written to the destination register. The empty high-order bits of each element are filled with the initial value of the sign bit of the data element. If the value specified by the second operand is greater than 15 (for words), or 31 (for doublewords), each destination element is filled with the initial value of the sign bit of the element.

The destination operand is an MMX register. The count operand (source operand) can be either an MMX register, a 64-bit memory operand, or an immediate 8-bit operand.

The PSRAW instruction shifts each of the four words in the destination register to the right by the number of bits specified in the count operand. The initial value of the sign bit of the data elements in the destination operand is copied into the most significant bits of the data element.

The PSRAD instruction shifts each of the two doublewords in the destination register to the right by the number of bits specified in the count operand. The initial value of the sign bit of the data elements in the destination operand is copied into the most significant bits of the data element.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PSRLW/PSRLD/PSRLQ—Packed Shift Right Logical

Opcode	Instruction	Description
0F D1 /r	PSRLW *mm, mm/m64*	Shift words in MMX register right by amount specified in MMX reg/memory while shifting in zeros.
0F 71 /2 ib	PSRLW *mm, imm8*	Shift words in MMX register right by Imm8.
0F D2 /r	PSRLD *mm, mm/m64*	Shift dwords in MMX register right by amount specified in MMX reg/memory while shifting in zeros.
0F 72 /2 ib	PSRLD *mm, imm8*	Shift dwords in MMX register right by Imm8 .
0F D3 /r	PSRLQ *mm, mm/m64*	Shift MMX register right by amount specified in MMX reg/memory while shifting in zeros.
0F 73 /2 ib	PSRLQ *mm, imm8*	Shift MMX register right by Imm8 while shifting in zeros.

Operation

```
IF the second operand is imm8
THEN
   temp ← imm8;
ELSE (* second operand is mm/m64 *)
   temp ← mm/m64;
IF instruction is PSRLW
THEN {
   mm(15..0) ← mm(15..0) >> temp;
   mm(31..16) ← mm(31..16) >> temp;
   mm(47..32) ← mm(47..32) >> temp;
   mm(63..48) ← mm(63..48) >> temp;
   }
ELSE IF instruction is PSRLD
THEN {
   mm(31..0) ← mm(31..0) >> temp;
   mm(63..32) ← mm(63..32) >> temp;
   }
ELSE  (* instruction is PSRLQ *)
   mm ← mm >> temp;
```

Description

The PSRL instructions shift the bits of the first operand to the right by the amount of bits specified in the count operand. The result of the shift operation is written to the destination register. The empty high-order bits are cleared (set to zero). If the value specified by the second operand is greater than 15 (for words), or 31 (for doublewords), or 63 (for quadwords), then the destination is set to all zeros.

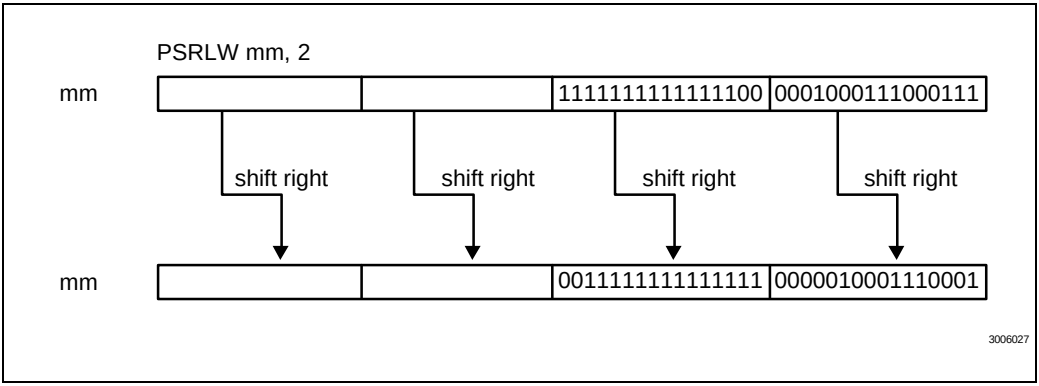
The destination operand is an MMX register. The count operand (source operand) can be either an MMX register, a 64-bit memory operand, or an immediate 8-bit operand.

The PSRLW instruction shifts each of the four words in the destination register to the right by the number of bits specified in the count operand. The empty high-order bits (of each word) are filled with zeros.

The PSRLD instruction shifts each of the two doublewords in the destination register to the right by the number of bits specified in the count operand. The empty high-order bits (of each doubleword) are filled with zeros.

The PSRLQ instruction shifts the 64-bit quadword in the destination register to the right by the number of bits specified in the count operand. The empty high-order bits are filled with zeros.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PSUBB/PSUBW/PSUBD—Packed Subtract

Opcode	Instruction	Description
0F F8 /r	PSUBB *mm, mm/m64*	Subtract packed byte in MMX reg/memory from packed byte in MMX register.
0F F9 /r	PSUBW *mm, mm/m64*	Subtract packed word in MMX reg/memory from packed word in MMX register.
0F FA /r	PSUBD *mm, mm/m64*	Subtract packed dword in MMX reg/memory from packed dword in MMX register.

Operation

```
IF instruction is PSUBB
 THEN  {
   mm(7..0) ← mm(7..0) - mm/m64(7..0);
   mm(15..8) ← mm(15..8) - mm/m64(15..8);
   mm(23..16) ← mm(23..16) - mm/m64(23..16);
   mm(31..24) ← mm(31..24) - mm/m64(31..24);
   mm(39..32) ← mm(39..32) - mm/m64(39..32);
   mm(47..40) ← mm(47..40) - mm/m64(47..40);
   mm(55..48) ← mm(55..48) - mm/m64(55..48);
   mm(63..56) ← mm(63..56) - mm/m64(63..56);
   }
IF instruction is PSUBW
THEN {
   mm(15..0) ← mm(15..0) - mm/m64(15..0);
   mm(31..16) ← mm(31..16) - mm/m64(31..16);
   mm(47..32) ← mm(47..32) - mm/m64(47..32);
   mm(63..48) ← mm(63..48) - mm/m64(63..48);
   }
ELSE { (* instruction is PSUBD *)
   mm(31..0) ← mm(31..0) - mm/m64(31..0);
   mm(63..32) ← mm(63..32) - mm/m64(63..32);
   }
```

Description

The PSUB instructions subtract the data elements of the source operand from the data elements of the destination operand. The result is written to the destination register. If the result is larger or smaller than the data-range limit for the data type, it wraps around.

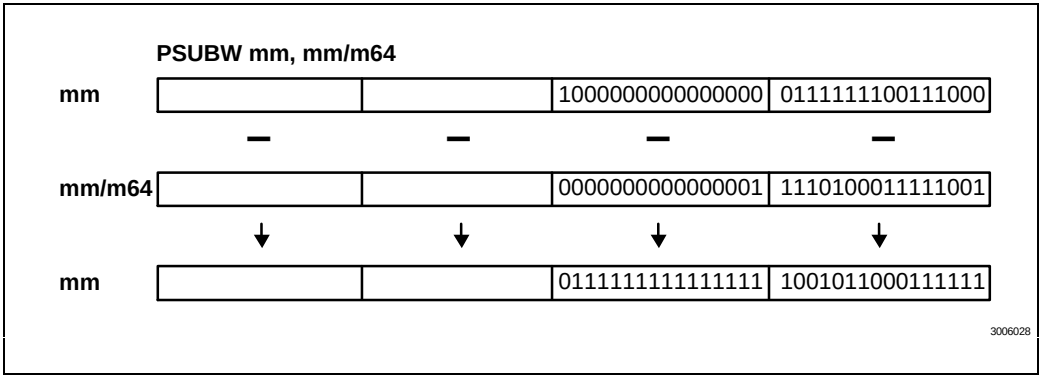
The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

The PSUBB instruction subtracts the bytes of the source operand from the bytes of the destination operand. The result is written to the MMX register. When the result is too large or too small to be represented in a byte, the result wraps around and the lower 8 bits are written to the destination register.

The PSUBW instruction subtracts the words of the source operand from the words of the destination operand. The result is written to the MMX register. When the result is too large or too small to be represented in a word, the result wraps around and the lower 16 bits are written to the destination register.

The PSUBD instruction subtracts the doublewords of the source operand from the doublewords of the destination operand. The result is written to the MMX register. When the result is too large or too small to be represented in a doubleword, the result wraps around and the lower 32 bits are written to the destination register.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory.

PSUBSB/PSUBSW —Packed Subtract with Saturation

Opcode	Instruction	Description
0F E8 /r	PSUBSB mm, mm/m64	Subtract signed packed byte in MMX reg/memory from signed packed byte in MMX register and saturate.
0F E9 /r	PSUBSW mm, mm/m64	Subtract signed packed word in MMX reg/memory from signed packed word in MMX register and saturate.

Operation

```
IF instruction is PSUBSB
 THEN{
   mm(7..0) ← SaturateToSignedByte (mm(7..0) - mm/m64 (7..0));
   mm(15..8) ← SaturateToSignedByte (mm(15..8) - mm/m64(15..8));
   mm(23..16) ← SaturateToSignedByte (mm(23..16) - mm/m64(23..16));
   mm(31..24) ← SaturateToSignedByte (mm(31..24) - mm/m64(31..24));
   mm(39..32) ← SaturateToSignedByte (mm(39..32) - mm/m64(39..32));
   mm(47..40) ← SaturateToSignedByte (mm(47..40) - mm/m64(47..40));
   mm(55..48) ← SaturateToSignedByte (mm(55..48) - mm/m64(55..48));
   mm(63..56) ← SaturateToSignedByte (mm(63..56) - mm/m64(63..56))
   }
ELSE { (* instruction is PSUBSW *)
   mm(15..0) ← SaturateToSignedWord (mm(15..0) - mm/m64(15..0));
   mm(31..16) ← SaturateToSignedWord (mm(31..16) - mm/m64(31..16));
   mm(47..32) ← SaturateToSignedWord (mm(47..32) - mm/m64(47..32));
   mm(63..48) ← SaturateToSignedWord (mm(63..48) - mm/m64(63..48));
   }
```

Description

The PSUBS instructions subtract the data elements of the source operand from the data elements of the destination operand. The results are saturated to the limits of a signed data element and written to the destination operand.

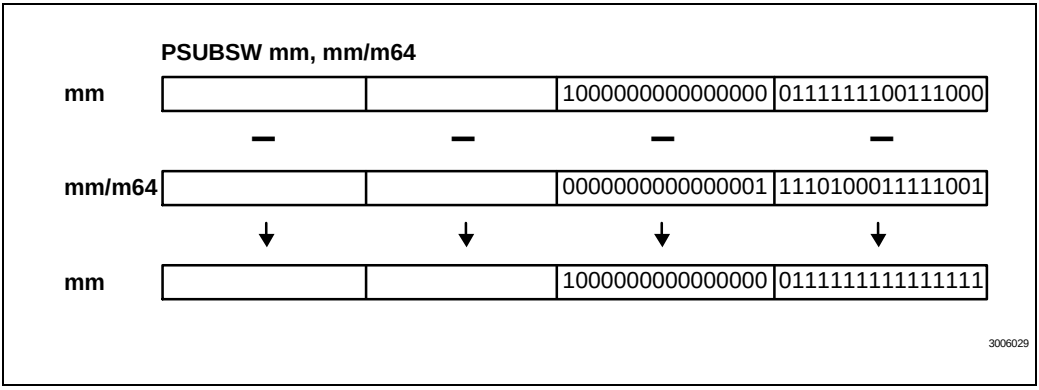
The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

The PSUBSB instruction subtracts the signed bytes of the source operand from the signed bytes of the destination operand, and writes the results to the destination register. If the result is larger or smaller than the range of a signed byte, the value is saturated; in the case of an overflow - to 0x7F, and in the case of an underflow - to 0x80.

The PSUBSW instruction subtracts the signed words of the source operand from the signed words of the destination operand and writes the results to the destination register. If the result

is larger or smaller than the range of a signed word, the value is saturated; in the case of an overflow to 0x7FFF, and in the case of an underflow to 0x8000.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PSUBUSB/PSUBUSW —Packed Subtract Unsigned with Saturation

Opcode	Instruction	Description
0F D8 /r	PSUBUSB mm, mm/m64	Subtract unsigned packed byte in MMX reg/memory from unsigned packed byte in MMX register and saturate.
0F D9 /r	PSUBUSW mm, mm/m64	Subtract unsigned packed word in MMX reg/memory from unsigned packed word in MMX register and saturate.

Operation

```
IF instruction is PSUBUSB
 THEN{
   mm(7..0) ← SaturateToUnsignedByte (mm(7..0) - mm/m64 (7..0) );
   mm(15..8) ← SaturateToUnsignedByte ( mm(15..8) - mm/m64(15..8) );
   mm(23..16) ← SaturateToUnsignedByte (mm(23..16) - mm/m64(23..16) );
   mm(31..24) ← SaturateToUnsignedByte (mm(31..24) - mm/m64(31..24) );
   mm(39..32) ← SaturateToUnsignedByte (mm(39..32) - mm/m64(39..32) );
   mm(47..40) ← SaturateToUnsignedByte (mm(47..40) - mm/m64(47..40) );
   mm(55..48) ← SaturateToUnsignedByte (mm(55..48) - mm/m64(55..48) );
   mm(63..56) ← SaturateToUnsignedByte (mm(63..56) - mm/m64(63..56) );
   }
ELSE { (* instruction is PSUBUSW *)
   mm(15..0) ← SaturateToUnsignedWord (mm(15..0) - mm/m64(15..0) );
   mm(31..16) ← SaturateToUnsignedWord (mm(31..16) - mm/m64(31..16) );
   mm(47..32) ← SaturateToUnsignedWord (mm(47..32) - mm/m64(47..32) );
   mm(63..48) ← SaturateToUnsignedWord (mm(63..48) - mm/m64(63..48) );
   }
```

Description

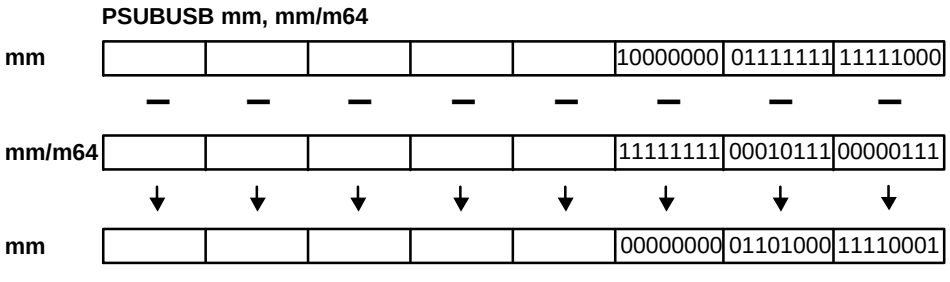
The PSUBUS instructions subtract the data elements of the source operand from the data elements of the destination register. The results are saturated to the limits of an unsigned data element and written to the destination operand.

The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

The PSUBUSB instruction subtracts the bytes of the source operand from the bytes of the destination operand and writes the results to the destination register. If the result element is less than zero (a negative value), it is saturated to 0x00.

The PSUBUSW instruction subtracts the words of the source operand from the words of the destination operand and writes the results to the destination register. If the result element is less than zero (a negative value), it is saturated to 0x0000.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ—Unpack High Packed Data

Opcode	Instruction	Description
0F 68 /r	PUNPCKHBW *mm, mm/m64*	Interleave bytes from the high halves of MMX register and MMX reg/memory into MMX register.
0F 69 /r	PUNPCKHWD *mm, mm/m64*	Interleave words from the high halves of MMX register and MMX reg/memory into MMX register.
0F 6A /r	PUNPCKHDQ *mm, mm/m64*	Interleave dwords from the high halves of MMX register and MMX reg/memory into MMX register.

Operation

```
IF instruction is PUNPCKHBW
THEN {
   mm(63..56) ← mm/m64(63..56);
   mm(55..48) ← mm(63..56);
   mm(47..40) ← mm/m64(55..48);
   mm(39..32) ← mm(55..48);
   mm(31..24) ← mm/m64(47..40);
   mm(23..16) ← mm(47..40);
   mm(15..8) ← mm/m64(39..32);
   mm(7..0) ← mm(39..32);
ELSE IF instruction is PUNPCKHW
THEN {
   mm(63..48) ← mm/m64(63..48);
   mm(47..32) ← mm(63..48);
   mm(31..16) ← mm/m64(47..32);
   mm(15..0) ← mm(47..32);
   }
ELSE { (* instruction is PUNPCKHDQ *)
   mm(63..32) ← mm/m64(63..32);
   mm(31..0) ← mm(63..32)
   }
```

Description

The PUNPCKH instructions unpack and interleave the high-order data elements of the destination and source operands into the destination operand. The low-order data elements are ignored.

The destination operand is an MMX register. The source operand can either be an MMX register or a 64-bit memory operand.

When unpacking from a memory operand, the full 64-bit operand is accessed from memory. The instruction uses only the high-order 32 bits.

The PUNPCKHBW instruction interleaves the four high-order bytes of the source operand and the four high-order bytes of the destination operand and writes them to the MMX register.

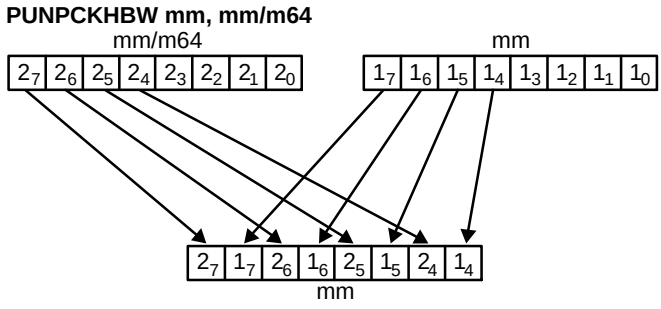
The PUNPCKHWD instruction interleaves the two high-order words of the source operand and the two high-order words of the destination operand and writes them to the MMX register.

The PUNPCKHDQ instruction interleaves the high-order 32 bits of the doubleword of the source operand and the high-order 32-bits of the doubleword of the destination operand and writes them to the MMX register.

Note

If the source operand is all zeros, the result is a zero extension of the high order elements of the destination operand. When using the PUNPCKHBW instruction the bytes are zero extended, or unpacked into unsigned words. When using the PUNPCKHWD instruction, the words are zero exended, or unpacked into unsigned doublewords.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ—Unpack Low Packed Data

Opcode	Instruction	Description
0F 60 /r	PUNPCKLBW *mm, mm/m32*	Interleave bytes from the low halves of MMX register and MMX reg/memory into MMX register.
0F 61 /r	PUNPCKLWD *mm, mm/m32*	Interleave words from the low halves of MMX register and MMX reg/memory into MMX register.
0F 62 /r	PUNPCKLDQ *mm, mm/m32*	Interleave dwords from the low halves of MMX register and MMX reg/memory into MMX register.

Operation

```
IF instruction is PUNPCKLBW
THEN {
   mm(63..56) ←mm/m32(31..24);
   mm(55..48) ← mm(31..24);
   mm(47..40) ← mm/m32(23..16);
   mm(39..32) ← mm(23..16);
   mm(31..24) ← mm/m32(15..8);
   mm(23..16) ←mm(15..8);
   mm(15..8) ← mm/m32(7..0);
   mm(7..0) ← mm(7..0);
   }
ELSE IF instruction is PUNPCKLWD
THEN {
   mm(63..48) ← mm/m32(31..16);
   mm(47..32) ← mm(31..16);
   mm(31..16) ← mm/m32(15..0);
   mm(15..0) ← mm(15..0);
   }
ELSE{  (* instruction is PUNPCKLDQ *)
   mm(63..32) ← mm/m32(31..0);
   mm(31..0) ← mm(31..0);
   }
```

Description

The PUNPCKL instructions unpack and interleave the low-order data elements of the destination and source operands into the destination operand.

The destination operand is an MMX register. The source operand can either be an MMX register or a 32-bit memory operand. When the source data comes from 64-bit registers, the upper 32 bits are ignored.

When unpacking from a memory operand, only 32 bits are accessed. The instruction uses all 32 bits.

The PUNPCKLBW instruction interleaves the four low-order bytes of the source operand and the four low-order bytes of the destination operand and writes them to the MMX register.

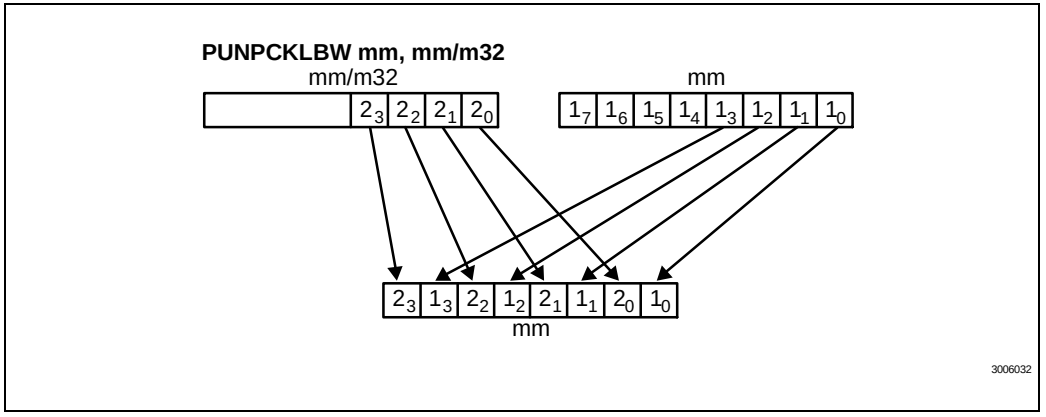
The PUNPCKLWD instruction interleaves the two low-order words of the source operand and the two low-order words of the destination operand and writes them to the MMX register.

The PUNPCKLDQ instruction interleaves the low-order doubleword of the source operand and the low-order doubleword of the destination operand and writes them to the MMX register.

Note

If the source operand has a value of all zeros, the result is a zero extension of the low order elements of the destination operand. When using the PUNPCKLBW instruction the bytes are zero extended, or unpacked into unsigned words. When using the PUNPCKLWD instruction, the words are zero extended, or unpacked into unsigned doublewords.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

PXOR—Bitwise Logical Exclusive OR

Opcode	Instruction	Description
0F EF /r	PXOR *mm, mm/m64*	XOR 64 bits from MMX reg/memory to MMX register.

Operation

mm ←mm XOR mm/m64;

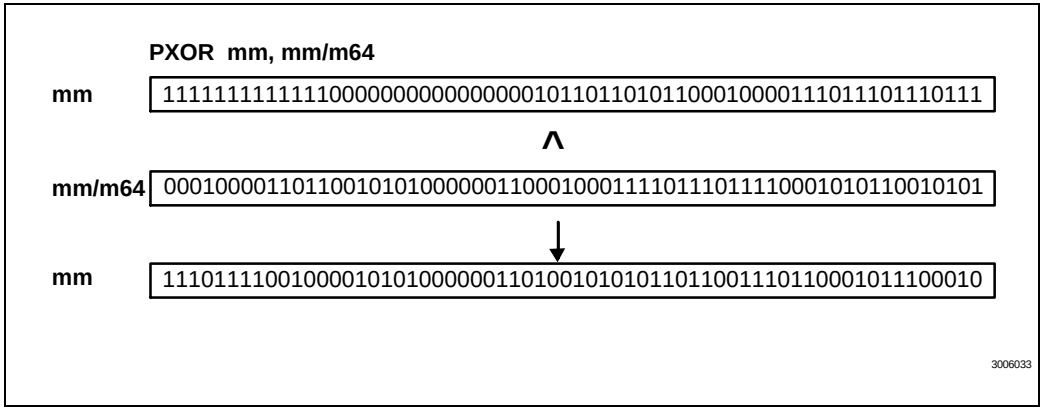
Description

The PXOR instruction performs a bitwise logical XOR on the 64 bits of the destination with the source operands and writes the result to destination register.

Each bit of the result is 1 if the corresponding bits of the two operands are different. Each bit is 0 if the corresponding bits of the operands are the same.

The source operand can either be an MMX register or a 64 bit memory operand.

Example



Flags Affected

None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand lies outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference.

A

IA MMX™ Instruction Set Summary

APPENDIX A
IA MMX™ INSTRUCTION SET SUMMARY

Table A-1 summarizes the IA MMX™ instruction set base mnemonics. The instructions are grouped by categories of related functions.

Most of the instructions have multiple variations that are not listed in Table A-1. For example, PADD has the following variations: PADDB, PADDW, and PADDD. The instruction variations and mnemonics are detailed in the Instruction description section of Chapter 5.

Table A-1. IA MMX Instruction Set Summary, Grouped into Functional Categories

CATEGORY				
Arithmetic		**Wraparound**	**Signed Saturation**	**Unsigned Saturation**
	addition	PADD	PADDS	PADDUS
	subtraction	PSUB	PSUBS	PSUBUS
	multiplication	PMULL/H		
	multiply and add	PMADD		
Comparison	compare	PCMPEQ PCMPGT		
Conversion	pack		PACKSS	PACKUS
	unpack	PUNPCKL/H		
Logical		**Packed**	**Full 64-bit**	
	and		PAND	
	and not		PANDN	
	or		POR	
	exclusive or		PXOR	
Shift	shift left logical	PSLL	PSLL	
	shift right logical	PSRL	PSRL	
	shift right arithmetic	PSRA		
Data Transfer Operations		**32-bit transfers**	**64-bit transfers**	
	register←register	MOVD	MOVQ	
	load from memory	MOVD	MOVQ	
	store to memory	MOVD	MOVQ	
FP and MMX™ State Management		EMMS		

B

IA MMX™ Instruction Formats and Encodings

APPENDIX B
IA MMX™ INSTRUCTION FORMATS AND ENCODINGS

B.1. INSTRUCTION FORMATS

All MMX instructions, except the EMMS instruction, use the same format similar as the two-byte Intel Architecture integer operations. Details of subfield encodings within these formats are presented below.

Table B-1. Encoding of Granularity of Data (gg) Field

gg	Granularity of Data
00	packed bytes
01	packed words
10	packed doublewords
11	quadword

Table B-2. Encoding of 32-bit General Purpose (reg) Field for Register-to-Register Operations

reg Field	Register Selected
000	EAX
001	ECX
010	EDX
011	EBX
100	ESP
101	EBP
110	ESI
111	EDI

NOTE: For register-to-register operations, the decoding of integer registers is independent of processor mode. For register-to-memory operations, the effective address is calculated based on the processor mode in effect.

Table B-3. Encoding of 64-bit MMX™ Register (mmxreg) Field

mmxreg Field	MMX Register Selected
000	mm0
001	mm1
010	mm2
011	mm3
100	mm4
101	mm5
110	mm6
111	mm7

For more details, see Table 25-2, Table 25-3, and Appendix F of the *Pentium® Processor Family Developer's Manual*.

B.2. INSTRUCTION ENCODINGS AND DATATYPE CROSS-REFERENCE

For each MMX instruction, Table B-4 lists instruction encodings and the datatypes supported—byte (B), word (W), doubleword (DW), and quadword (QW).

O	= output
I	= input
S	= signed saturation
U	= unsigned saturation
n/a	= not applicable

Figure B-1. Key to Codes for Datatype Cross-Reference

Table B-4. IA MMX™ Instruction Formats and Encodings

Instruction	Format	B	W	DW	QW
EMMS - Empty MMX state	0000 1111:01110111	n/a	n/a	n/a	n/a
MOVD - Move doubleword		N	N	Y	N
reg to mmxreg	0000 1111:01101110: 11 mmxreg reg				
reg from mmxreg	0000 1111:01111110: 11 mmxreg reg				
mem to mmxreg	0000 1111:01101110: mod mmxreg r/m				
mem from mmxreg	0000 1111:01111110: mod mmxreg r/m				
MOVQ - Move quadword		N	N	N	Y
mmxreg2 to mmxreg1	0000 1111:01101111: 11 mmxreg1 mmxreg2				
mmxreg2 from mmxreg1	0000 1111:01111111: 11 mmxreg1 mmxreg2				
mem to mmxreg	0000 1111:01101111: mod mmxreg r/m				
mem from mmxreg	0000 1111:01111111: mod mmxreg r/m				

Table B-4. IA MMX™ Instruction Formats and Encodings (Contd.)

Instruction	Format	B	W	DW	QW
PACKSSDW[1] - Pack dword to word data (signed with saturation)		n/a	O	I	n/a
mmxreg2 to mmxreg1	0000 1111:01101011: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:01101011: mod mmxreg r/m				
PACKSSWB[1] - Pack word to byte data (signed with saturation)		O	I	n/a	n/a
mmxreg2 to mmxreg1	0000 1111:01100011: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:01100011: mod mmxreg r/m				
PACKUSWB[1] - Pack word to byte data (unsigned with saturation)		O	I	n/a	n/a
mmxreg2 to mmxreg1	0000 1111:01100111: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:01100111: mod mmxreg r/m				
PADD - Add with wrap-around		Y	Y	Y	N
mmxreg2 to mmxreg1	0000 1111: 111111gg: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111: 111111gg: mod mmxreg r/m				

Table B-4. IA MMX™ Instruction Formats and Encodings (Contd.)

Instruction	Format	B	W	DW	QW
PADDS - Add signed with saturation		Y	Y	N	N
mmxreg2 to mmxreg1	0000 1111: 111011gg: 11 mmxreg1 mmxreg2				
memory to reg	0000 1111: 111011gg: mod reg r/m				
PADDUS - Add unsigned with saturation		Y	Y	N	N
mmxreg2 to mmxreg1	0000 1111: 110111gg: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111: 110111gg: mod mmxreg r/m				
PAND - Bitwise And		N	N	N	Y
mmxreg2 to mmxreg1	0000 1111:11011011: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:11011011: mod mmxreg r/m				
PANDN - Bitwise AndNot		N	N	N	Y
mmxreg2 to mmxreg1	0000 1111:11011111: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:11011111: mod mmxreg r/m				
PCMPEQ - Packed compare for equality		Y	Y	Y	N
mmxreg2 with mmxreg1	0000 1111:011101gg: 11 mmxreg1 mmxreg2				
memory with mmxreg	0000 1111:011101gg: mod mmxreg r/m				

Table B-4. IA MMX™ Instruction Formats and Encodings (Contd.)

Instruction	Format	B	W	DW	QW
PCMPGT - Packed compare greater (signed)		Y	Y	Y	N
mmxreg2 with mmxreg1	0000 1111:011001gg: 11 mmxreg1 mmxreg2				
memory with mmxreg	0000 1111:011001gg: mod mmxreg r/m				
PMADD - Packed multiply add		n/a	I	O	n/a
mmxreg2 to mmxreg1	0000 1111:11110101: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:11110101: mod mmxreg r/m				
PMULH - Packed multiplication		N	Y	N	N
mmxreg2 to mmxreg1	0000 1111:11100101: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:11100101: mod mmxreg r/m				
PMULL - Packed multiplication		N	Y	N	N
mmxreg2 to mmxreg1	0000 1111:11010101: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:11010101: mod mmxreg r/m				
POR - Bitwise Or		N	N	N	Y
mmxreg2 to mmxreg1	0000 1111:11101011: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:11101011: mod mmxreg r/m				

Table B-4. IA MMX™ Instruction Formats and Encodings (Contd.)

Instruction	Format	B	W	DW	QW
PSLL[2] - Packed shift left logical		N	Y	Y	Y
mmxreg2 by mmxreg1	0000 1111:111100gg: 11 mmxreg1 mmxreg2				
mmxreg by memory	0000 1111:111100gg: 11 mmxreg r/m				
mmxreg by immediate	0000 1111:011100gg: 11 110 mmxreg: imm8 data				
PSRA[2] - Packed shift right arithmetic		N	Y	Y	N
mmxreg2 by mmxreg1	0000 1111:111000gg: 11 mmxreg1 mmxreg2				
mmxreg by memory	0000 1111:111000gg: 11 mmxreg r/m				
mmxreg by immediate	0000 1111:011100gg: 11 100 mmxreg: imm8 data				
PSRL[2] - Packed shift right logical		N	Y	Y	Y
mmxreg2 by mmxreg1	0000 1111:110100gg: 11 mmxreg1 mmxreg2				
mmxreg by memory	0000 1111:110100gg: 11 mmxreg r/m				
mmxreg by immediate	0000 1111:011100gg: 11 010 mmxreg: imm8 data				
PSUB - Subtract with wrap-around		Y	Y	Y	N
mmxreg2 to mmxreg1	0000 1111:111110gg: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:111110gg: mod mmxreg r/m				

Table B-4. IA MMX™ Instruction Formats and Encodings (Contd.)

Instruction	Format	B	W	DW	QW
PSUBS - Subtract signed with saturation		Y	Y	N	N
mmxreg2 to mmxreg1	0000 1111:111010gg: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:111010gg: mod mmxreg r/m				
PSUBUS - Subtract unsigned with saturation		Y	Y	N	N
mmxreg2 to mmxreg1	0000 1111:110110gg: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:110110gg: mod mmxreg r/m				
PUNPCKH - Unpack high data to next larger type		Y	Y	Y	N
mmxreg2 to mmxreg1	0000 1111:011010gg: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:011010gg: mod mmxreg r/m				
PUNPCKL - Unpack low data to next larger type		Y	Y	Y	N
mmxreg2 to mmxreg1	0000 1111:011000gg: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:011000gg: mod mmxreg r/m				

Table B-4. IA MMX™ Instruction Formats and Encodings (Contd.)

Instruction	Format	B	W	DW	QW
PXOR - Bitwise Xor		N	N	N	Y
mmxreg2 to mmxreg1	0000 1111:11101111: 11 mmxreg1 mmxreg2				
memory to mmxreg	0000 1111:11101111: mod mmxreg r/m				

NOTE:

1. The PACK instructions perform saturation from signed packed data of one type to signed or unsigned data of the next smaller type.
2. The format of shift instructions has one additional format to support shifting by immediate shift-counts. The shift operations are not supported equally for all data types.

C

Alphabetical list of IA MMX™ Instruction Set Mnemonics

APPENDIX C
ALPHABETICAL LIST OF IA MMX™ INSTRUCTION SET MNEMONICS

The following table lists the mnemonics of the IA MMX™ instructions in alphabetical order. For each mnemonic, it summarizes the type of source data, the encoding of the first and second bytes in hexadecimal, and the format used.

Table C-1. IA MMX™ Instruction Set Mnemonics

MNEMONIC	OPERAND TYPES	Byte 1	Byte 2	Byte 3, [4]
EMMS	None	0F	77	mod-rm, [sib]
MOVD	register, memory/iregister	0F	6E	mod-rm, [sib]
MOVD	memory/iregister, register	0F	7E	mod-rm, [sib]
MOVQ	register, memory/register	0F	6F	mod-rm, [sib]
MOVQ	memory/register, register	0F	7F	mod-rm, [sib]
PACKSSDW	register, memory/register	0F	6B	mod-rm, [sib]
PACKSSWB	register, memory/register	0F	63	mod-rm, [sib]
PACKUSWB	register, memory/register	0F	67	mod-rm, [sib]
PADDB	register, memory/register	0F	FC	mod-rm, [sib]
PADDD	register, memory/register	0F	FE	mod-rm, [sib]
PADDSB	register, memory/register	0F	EC	mod-rm, [sib]
PADDSW	register, memory/register	0F	ED	mod-rm, [sib]
PADDUSB	register, memory/register	0F	DC	mod-rm, [sib]
PADDUSW	register, memory/register	0F	DD	mod-rm, [sib]
PADDW	register, memory/register	0F	FD	mod-rm, [sib]
PAND	register, memory/register	0F	DB	mod-rm, [sib]
PANDN	register, memory/register	0F	DF	mod-rm, [sib]
PCMPEQB	register, memory/register	0F	74	mod-rm, [sib]
PCMPEQD	register, memory/register	0F	76	mod-rm, [sib]

Table C-1. IA MMX™ Instruction Set Mnemonics (Contd.)

MNEMONIC	OPERAND TYPES	Byte 1	Byte 2	Byte 3, [4]
PCMPEQW	register, memory/register	0F	75	mod-rm, [sib]
PCMPGTB	register, memory/register	0F	64	mod-rm, [sib]
PCMPGTD	register, memory/register	0F	66	mod-rm, [sib]
PCMPGTW	register, memory/register	0F	65	mod-rm, [sib]
PMADDWD	register, memory/register	0F	F5	mod-rm, [sib]
PMULHW	register, memory/register	0F	E5	mod-rm, [sib]
PMULLW	register, memory/register	0F	D5	mod-rm, [sib]
POR	register, memory/register	0F	EB	mod-rm, [sib]
PSHIMD*	register, immediate	0F	72	mod-rm, imm
PSHIMQ*	register, immediate	0F	73	mod-rm, imm
PSHIMW*	register, immediate	0F	71	mod-rm, imm
PSLLD	register, memory/register	0F	F2	mod-rm, [sib]
PSLLQ	register, memory/register	0F	F3	mod-rm, [sib]
PSLLW	register, memory/register	0F	F1	mod-rm, [sib]
PSRAD	register, memory/register	0F	E2	mod-rm, [sib]
PSRAW	register, memory/register	0F	E1	mod-rm, [sib]
PSRLD	register, memory/register	0F	D2	mod-rm, [sib]
PSRLQ	register, memory/register	0F	D3	mod-rm, [sib]
PSRLW	register, memory/register	0F	D1	mod-rm, [sib]
PSUBB	register, memory/register	0F	F8	mod-rm, [sib]

Table C-1. IA MMX™ Instruction Set Mnemonics (Contd.)

MNEMONIC	OPERAND TYPES	Byte 1	Byte 2	Byte 3, [4]
PSUBD	register, memory/register	0F	FA	mod-rm, [sib]
PSUBSB	register, memory/register	0F	E8	mod-rm, [sib]
PSUBSW	register, memory/register	0F	E9	mod-rm, [sib]
PSUBUSB	register, memory/register	0F	D8	mod-rm, [sib]
PSUBUSW	register, memory/register	0F	D9	mod-rm, [sib]
PSUBW	register, memory/register	0F	F9	mod-rm, [sib]
PUNPCKHBW	register, memory/register	0F	68	mod-rm, [sib]
PUNPCKHDQ	register, memory/register	0F	6A	mod-rm, [sib]
PUNPCKHWD	register, memory/register	0F	69	mod-rm, [sib]
PUNPCKLBW	register, memory/register	0F	60	mod-rm, [sib]
PUNPCKLDQ	register, memory/register	0F	62	mod-rm, [sib]
PUNPCKLWD	register, memory/register	0F	61	mod-rm, [sib]
PXOR	register, memory/register	0F	EF	mod-rm, [sib]

Notes:

* These are not the actual mnemonics:

PSHIMD represents the PSLLD, PSRAD and PSRLD instructions when shifting by immediate shift counts.

PSHIMW represents the PSLLW, PSRAW and PSRLW instructions when shifting by immediate shift counts.

PSHIMQ represents the PSLLQ and PSRLQ instructions when shifting by immediate shift counts.

The instructions that shift by immediate counts are differentiated by the ModR/M bytes (See Appendix B).

D

IA MMX™ Instruction Set Opcode Map

APPENDIX D
IA MMX™ INSTRUCTION SET OPCODE MAP

The detailed encodings of the Intel Architecture MMX™ instructions are listed in the shaded boxes of the Opcode Map tables below. All MMX instructions, except the EMMS instruction, use the same format as the two-byte Intel Architecture integer operations.

All blanks in the Opcode Map are reserved and should not be used. Do not depend on the operation of unspecified opcodes. 0F0Bh or 0FB9h should be used when deliberately generating an illegal opcode exception.

Key to Abbreviations

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lowercase letter, specifies the type of operand. For opcodes with two operands, the left code refers to the destination operand and the right code refers to the source operand. All MMX instructions, except the EMMS instruction, reference and operate on two operands.

Codes for Addressing Method

C The reg field of the ModR/M byte selects a control register; e.g., MOV (0F20, 0F22).

D The reg field of the ModR/M byte selects a debug register; e.g., MOV (0F21, 0F23).

E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

G The reg field of the ModR/M byte selects a general register; e.g., AX(000).

I Immediate data. The value of the operand is encoded in subsequent bytes of the instruction.

M The ModR/M byte may refer only to memory; e.g., LSS, LFS, LGS, CMPXCHG8B.

P The reg field of the ModR/M byte selects a packed quadword MMX register.

Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX register or a memory address. If it is a memory address, the address is computed

from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

R The mod field of the ModR/M byte may refer only to a general register; e.g., MOV (0F20-0F24, 0F26).

Codes for Operand Type

b Byte (regardless of operand size attribute).

d Doubleword (regardless of operand size attribute).

p 32-bit or 48-bit pointer, depending on operand size attribute.

q Quadword (regardless of operand size attribute).

s Six-byte pseudo-descriptor.

v Word or doubleword, depending on operand size attribute.

w Word (regardless of operand size attribute).

Register Codes

When an operand is a specific register encoded in the opcode, the register is identified by its name, for example: AX, CL, or ESI. The name of the register indicates whether the register is 32-bits, 16-bits, or 8-bits wide. A register identifier of the form eXX is used when the width of the register depends on the oeprand size attribute; for example, eAX indicates that the AX register is used when the operand size attribute is 16 and the EAX register is used when the operand size attribute is 32.

Table D-1. Opcode Map (First Byte is 0FH)

	0	1	2	3	4	5	6	7
0	GRP 6		LAR Gv, Ew	LSL Gv, Ew			CLTS	
1								
2	MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3	WRMSR	RDTSC	RDMSR					
4								
5								
6	PUNPCKLBW Pq, Qd	PUNPCKLWD Pq, Qd	PUNPCKLDQ Pq, Qd	PACKSSWB Pq, Qq	PCMPGTB Pq, Qq	PCMPGTW Pq, Qq	PCMPGTD Pq, Qq	PACKUSWB Pq, Qq
7		Grp A			PCMPEQB Pq, Qq	PCMPEQW Pq, Qq	PCMPEQD Pq, Qq	EMMS
		PSHIMW	PSHIMD	PSHIMQ				
8	Long-displacement jump on condition (Jv)							
	JO	JNO	JB	JNB	JZ	JNZ	JBE	JNBE
9	Byte Set on condition (Eb)							
	SETO	SETNO	SETB	SETNB	SETZ	SETNZ	SETBE	SETNBE
A	PUSH FS	POP FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B	CMPXCH Eb, Gb	CMPXCH Ev, Gv	LSS Mp	BTR Ev, Gv	LFS Mp	LGS Mp	MOVZX Gv, Eb	MOVZX Gv, Ew
C	XADD Eb, Gb	XADD Ev, Gv						GRP 9
D		PSRLW Pq, Qq	PSRLD Pq, Qq	PSRLQ Pq, Qq		PMULLW Pq, Qq		
E		PSRAW Pq, Qq	PSRAD Pq, Qq			PMULHW Pq, Qq		
F		PSLLW Pq, Qq	PSLLD Pq, Qq	PSLLQ Pq, Qq		PMADDWD Pq, Qq		

Table D-1. Opcode Map (First Byte is 0FH) (Contd)

	8	9	A	B	C	D	E	F
0	INVD	WB INVD		Illegal opcode				
1								
2								
3								
4								
5								
6	PUNPCKHBW Pq, Qq	PUNPCKHWD Pq, Qq	PUNPCKHDQ Pq, Qq	PACKSSDW Pq, Qq			MOVD Pq, Ed	MOVQ Pq, Qq
7							MOVD Ed, Pd	MOVQ Qq, Pq
8	Long-displacement jump on condition (Jv)							
	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
	Byte Set on condition (jv)							
9	SETS Eb	SETNS Eb	SETP Eb	SETNP Eb	SETL Eb	SETNL Eb	SETLE Eb	SETNLE Eb
A	PUSH GS	POP GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL		IMUL Gv, Ev
B		Illegal opcode	GRP 8 Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSX Gv, Eb	MOVSX Gv, Ew
C	BSWAP EAX	BSWAP ECX	BSWAP EDX	BSWAP EBX	BSWAP ESP	BSWAP EBP	BSWAP ESI	BSWAP EDI
D	PSUBUSB Pq, Qq	PSUBUSW Pq, Qq		PAND Pq, Qq	PADDUSB Pq, Qq	PADDUSW Pq, Qq		PANDN Pq, Qq
E	PSUBSB Pq, Qq	PSUBSW Pq, Qq		POR Pq, Qq	PADDSB Pq, Qq	PADDSW Pq, Qq		PXOR Pq, Qq
F	PSUBB Pq, Qq	PSUBW Pq, Qq	PSUBD Pq, Qq		PADDB Pq, Qq	PADDW Pq, Qq	PADDD Pq, Qq	

Table D-2. Opcodes Determined by Bits 5, 4, 3 of Mod R/M Byte

mod	nnn	R/M

Group	000	001	010	011	100	101	110	111
1	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
2	ROL	ROR	RCL	RCR	SHL SAL	SHR		SAR
3	TEST Ib/Iv		NOT	NEG	MUL AL/eAX	IMUL AL/eAX	DIV AL/eAX	IDIV AL/eAX
4	INC Eb	DEC Eb						
5	INC Ev	DEC Ev	CALL Ev	CALL Ep	JMP Ev	JMP Ep	PUSH Pv	
6	SLDT Ew	STR Ew	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
7	SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Ew		LMSW Ew	INVLPG
8					BT	BTS	BTR	BTC
9		CMPXCH 8BMq						
A			PSRL Pq, Ib		PSRA Pq, Ib		PSLL Pq, Ib	