AMD64 Technology

AMD64 Architecture Programmer's Manual: Volumes 1-5

Publication # **40332** Revision: **4.04**
Issue Date: **November 2021**

Trademarks

AMD64 Technology

AMD64 Architecture Programmer's Manual

Volume 1: Application Programming

Publication No.	Revision	Date
24592	3.23	October 2020

Advanced Micro Devices

Trademarks

Contents

Figures

Tables

Revision History

Date	Revision	Description
October 2020	3.23	Added Shadow Stack support. Preface: Added updates. Chapter 2: Memory Model. Added content. Chapter 3: General-Purpose Programming. Added content.
December 2017	3.22	Clarified Items in Notational Conventions in the Preface. Clarified Memory Fence, Serializing Instructions and Internal Caches in Chapter 3. Added Instruction Cache Coherency in Chapter 3. Removed redundant information Section 3.11. Corrected description of streaming instructions in section 4.12.4.
October 2013	3.21	Integrated the AVX2 instruction subset into Chapter 4, “Streaming SIMD Extensions Media and Scientific Programming,” on page 111.
May 2013	3.20	Clarified Section 3.11. “Cross-Modifying Code” on page 107.
March 2012	3.19	Added description of the MOVBE instruction to discussion of move instructions on page 45.
December 2011	3.18	Added corrections and clarifications to “Legacy Prefixes” on page 76. Corrected some formatting issues on figure titles in Chapter 4.
September 2011	3.17	Completed integration of extended SSE instruction set into application programming discussion.
May 2011	3.16	Updated application programming model to include the YMM Registers. Added descriptions for SSE4.1 and SSE4.2 instructions. Added F16C to Section 4.6.3. Added BMI and TBM instructions to Section 3.3. Added descriptive information for XOP instructions in appropriate section 4.5 locations. Added description of 256-bit data types to Section 4.4.
November 2009	3.15	Modified description of the Auxiliary Carry Flag. Clarified section, “Load Segment Registers.” Added section “Atomicity of accesses.” Revised section, “Cross-Modifying Code.”
September 2007	3.14	Incorporated minor clarifications and formatting changes.

Date	Revision	Description
July 2007	3.13	Revised rFLAGS register table. Added section “Cross-Modifying Code.” Added section “Feature Detection in a Virtualized Environment”. Merged table of MXCSR register reset values into Figure. Added “Misaligned Exception Mask (MM)”. Revised indefinite-value encodings in tables. Revised section “Precision.” Made minor editorial changes for purposes of clarification.
September 2006	3.12	Incorporated minor clarifications and formatting changes.
December 2005	3.11	Updated index entries.
February 2005	3.10	Clarified section “Self-Modifying Code.” Added general descriptions of SSE3 instructions to Chapter 4. Added description of the CMPXCHG16B instruction to Chapter 3. Elaborated explanation of PREFETCH/*evel* instructions.
September 2003	3.09	Corrected several factual errors.
September 2002	3.07	Corrected minor organizational problems in sections dealing with ‘Prefetch’ instructions in Chapters 3, 4, and 5. Clarified the general description of the operation of certain 128-bit media instructions in Chapter 1. Corrected a factual error in the description of the FNINIT/FINIT instructions in Chapter 6. Corrected operand descriptions for the CMOVcc instructions in Chapter 3. Added Revision History. Corrected marketing denotations.

Preface

About This Book

This book is part of a multivolume work entitled the *AMD64 Architecture Programmer's Manual*. This table lists each volume and its order number.

Title	Order No.
Volume 1: Application Programming	24592
Volume 2: System Programming	24593
Volume 3: General-Purpose and System Instructions	24594
Volume 4: 128-Bit and 256-Bit Media Instructions	26568
Volume 5: 64-Bit Media and x87 Floating-Point Instructions	26569

Audience

This volume is intended for programmers writing application programs, compilers, or assemblers. It assumes prior experience in microprocessor programming, although it does not assume prior experience with the legacy x86 or AMD64 microprocessor architecture.

This volume describes the AMD64 architecture's resources and functions that are accessible to application software, including memory, registers, instructions, operands, I/O facilities, and application-software aspects of control transfers (including interrupts and exceptions) and performance optimization.

System-programming topics—including the use of instructions running at a current privilege level (CPL) of 0 (most-privileged)—are described in Volume 2. Details about each instruction are described in Volumes 3, 4, and 5.

Organization

This volume begins with an overview of the architecture and its memory organization and is followed by chapters that describe the four application-programming models available in the AMD64 architecture:

- *General-Purpose Programming*—This model uses the integer general-purpose registers (GPRs). The chapter describing it also describes the basic application environment for exceptions, control transfers, I/O, and memory optimization that applies to all other application-programming models.

- *Streaming SIMD Extensions (SSE) Programming*—This model uses the SSE (YMM/XMM) registers and supports integer and floating-point operations on vector (packed) and scalar data types.
- *Multimedia Extensions (MMX™) Programming*—This model uses the 64-bit MMX registers and supports integer and floating-point operations on vector (packed) and scalar data types.
- *x87 Floating-Point Programming*—This model uses the 80-bit x87 registers and supports floating-point operations on scalar data types.

The index at the end of this volume cross-references topics within the volume. For other topics relating to the AMD64 architecture, see the tables of contents and indexes of the other volumes.

Conventions and Definitions

The following section **Notational Conventions** describes notational conventions used in this volume and in the remaining volumes of this *AMD64 Architecture Programmer's Manual.* This is followed by a **Definitions** section which lists a number of terms used in the manual along with their technical definitions. Some of these definitions assume knowledge of the legacy x86 architecture. See "Related Documents" on page xxx for further information about the legacy x86 architecture. Finally, the **Registers** section lists the registers which are a part of the application programming model.

Notational Conventions

#GP(0)

An instruction exception—in this example, a general-protection exception with error code of 0.

1011b

A binary value—in this example, a 4-bit value.

F0EA_0B02h

A hexadecimal value. Underscore characters may be inserted to improve readability.

128

Numbers without an alpha suffix are decimal unless the context indicates otherwise.

7:4

A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first. Commas may be inserted to indicate gaps.

CR0–CR4

A register range, from register CR0 through CR4, inclusive, with the low-order register first.

CR0[PE], CR0.PE

Notation for referring to a field within a register—in this case, the PE field of the CR0 register.

CR0[PE] = 1, CR0.PE = 1

The PE field of the CR0 register is set (contains the value 1).

EFER[LME] = 0, EFER.LME = 0

The LME field of the EFER register is cleared (contains a value of 0).

DS:SI

A far pointer or logical address. The real address or segment descriptor specified by the segment register (DS in this example) is combined with the offset contained in the second register (SI in this example) to form a real or virtual address.

RFLAGS[13:12]

A field within a register identified by its bit range. In this example, corresponding to the IOPL field.

Definitions

128-bit media instructions

Instructions that operate on the various 128-bit vector data types. Supported within both the *legacy SSE* and *extended SSE* instruction sets.

256-bit media instructions

Instructions that operate on the various 256-bit vector data types. Supported within the *extended SSE* instruction set.

64-bit media instructions

Instructions that operate on the 64-bit vector data types. These are primarily a combination of MMX and 3DNow!™ instruction sets and their extensions, with some additional instructions from the *SSE1* and *SSE2* instruction sets.

16-bit mode

Legacy mode or compatibility mode in which a 16-bit address size is active. See *legacy mode* and *compatibility mode*.

32-bit mode

Legacy mode or compatibility mode in which a 32-bit address size is active. See *legacy mode* and *compatibility mode*.

64-bit mode

A submode of *long mode*. In 64-bit mode, the default address size is 64 bits and new features, such as register extensions, are supported for system and application software.

absolute

Said of a displacement that references the base of a code segment rather than an instruction pointer. Contrast with *relative*.

AES

Advance Encryption Standard (AES) algorithm acceleration instructions; part of *Streaming SIMD Extensions (SSE)*.

ASID

Address space identifier.

AVX

Extension of the SSE instruction set supporting 128- and 256-bit vector (packed) operands. See *Streaming SIMD Extensions*.

AVX2

Extension of the AVX instruction subset that adds more support for 256-bit vector (mostly packed integer) operands and a few new SIMD instructions. See *Streaming SIMD Extensions*.

biased exponent

The sum of a floating-point value's exponent and a constant bias for a particular floating-point data type. The bias makes the range of the biased exponent always positive, which allows reciprocation without overflow.

byte

Eight bits.

clear

To write a bit value of 0. Compare *set*.

compatibility mode

A submode of *long mode*. In compatibility mode, the default address size is 32 bits, and legacy 16-bit and 32-bit applications run without modification.

commit

To irreversibly write, in program order, an instruction's result to software-visible storage, such as a register (including flags), the data cache, an internal write buffer, or memory.

CPL

Current privilege level.

direct

Referencing a memory location whose address is included in the instruction's syntax as an immediate operand. The address may be an absolute or relative address. Compare *indirect*.

dirty data

Data held in the processor's caches or internal buffers that is more recent than the copy held in main memory.

displacement

A signed value that is added to the base of a segment (absolute addressing) or an instruction pointer (relative addressing). Same as *offset*.

doubleword

Two words, or four bytes, or 32 bits.

double quadword

Eight words, or 16 bytes, or 128 bits. Also called *octword*.

effective address size

The address size for the current instruction after accounting for the default address size and any address-size override prefix.

effective operand size

The operand size for the current instruction after accounting for the default operand size and any operand-size override prefix.

element

See *vector*.

exception

An abnormal condition that occurs as the result of executing an instruction. The processor's response to an exception depends on the type of the exception. For all exceptions except SSE floating-point exceptions and x87 floating-point exceptions, control is transferred to the handler (or service routine) for that exception, as defined by the exception's vector. For floating-point exceptions defined by the IEEE 754 standard, there are both masked and unmasked responses. When unmasked, the exception handler is called, and when masked, a default response is provided instead of calling the handler.

extended SSE

Enhanced set of SIMD instructions supporting 256-bit vector data types and allowing the specification of up to four operands. A subset of the *Streaming SIMD Extensions (SSE).* Includes the *AVX*, *AVX2*, *FMA*, *FMA4*, and *XOP* instructions. Compare *legacy SSE*.

flush

An often ambiguous term meaning (1) writeback, if modified, and invalidate, as in "flush the cache line," or (2) invalidate, as in "flush the pipeline," or (3) change a value, as in "flush to zero."

FMA4

Fused Multiply Add, four operand. Part of the *extended SSE* instruction set.

FMA

Fused Multiply Add. Part of the *extended SSE* instruction set.

GDT
: Global descriptor table.

GIF
: Global interrupt flag.

IDT
: Interrupt descriptor table.

IGN
: Ignored. Value written is ignored by hardware. Value returned on a read is indeterminate. See *reserved*.

indirect
: Referencing a memory location whose address is in a register or other memory location. The address may be an absolute or relative address. Compare *direct*.

IRB
: The virtual-8086 mode interrupt-redirection bitmap.

IST
: The long-mode interrupt-stack table.

IVT
: The real-address mode interrupt-vector table.

LDT
: Local descriptor table.

legacy x86
: The legacy x86 architecture. See “Related Documents” on page xxx for descriptions of the legacy x86 architecture.

legacy mode
: An operating mode of the AMD64 architecture in which existing 16-bit and 32-bit applications and operating systems run without modification. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Legacy mode has three submodes, *real mode*, *protected mode*, and *virtual-8086 mode*.

legacy SSE
: A subset of the *Streaming SIMD Extensions (SSE)* composed of the *SSE1*, *SSE2*, *SSE3*, *SSSE3*, *SSE4.1*, *SSE4.2*, and *SSE4A* instruction sets. Compare *extended SSE*.

LIP
: Linear Instruction Pointer. LIP = (CS.base + rIP).

long mode

An operating mode unique to the AMD64 architecture. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Long mode has two submodes, *64-bit mode* and *compatibility mode*.

lsb

Least-significant bit.

LSB

Least-significant byte.

main memory

Physical memory, such as RAM and ROM (but not cache memory) that is installed in a particular computer system.

mask

(1) A control bit that prevents the occurrence of a floating-point exception from invoking an exception-handling routine. (2) A field of bits used for a control purpose.

MBZ

Must be zero. If software attempts to set an MBZ bit to 1, a general-protection exception (#GP) occurs. See *reserved*.

memory

Unless otherwise specified, *main memory*.

msb

Most-significant bit.

MSB

Most-significant byte.

multimedia instructions

Those instructions that operate simultaneously on multiple elements within a vector data type. Comprises the *256-bit media instructions*, *128-bit media instructions,* and *64-bit media instructions*.

octword

Same as *double quadword.*

offset

Same as *displacement*.

overflow

The condition in which a floating-point number is larger in magnitude than the largest, finite, positive or negative number that can be represented in the data-type format being used.

packed
See *vector*.

PAE
Physical-address extensions.

physical memory
Actual memory, consisting of *main memory* and cache.

probe
A check for an address in a processor's caches or internal buffers. *External probes* originate outside the processor, and *internal probes* originate within the processor.

procedure stack
A portion of a stack segment in memory that is used to link procedures. Also known as a *program stack.*

program stack
See *procedure stack.*

protected mode
A submode of *legacy mode*.

quadword
Four words, or eight bytes, or 64 bits.

RAZ
Read as zero. Value returned on a read is always zero (0) regardless of what was previously written. (See *reserved*)

real-address mode
See *real mode*.

real mode
A short name for *real-address mode,* a submode of *legacy mode*.

relative
Referencing with a displacement (also called offset) from an instruction pointer rather than the base of a code segment. Contrast with *absolute*.

reserved
Fields marked as reserved may be used at some future time.

To preserve compatibility with future processors, reserved fields require special handling when read or written by software. Software must not depend on the state of a reserved field (unless qualified as RAZ), nor upon the ability of such fields to return a previously written state.

If a field is marked reserved without qualification, software must not change the state of that field; it must reload that field with the same value returned from a prior read.

Reserved fields may be qualified as IGN, MBZ, RAZ, or SBZ (see definitions).

REX

An instruction encoding prefix that specifies a 64-bit operand size and provides access to additional registers.

RIP-relative addressing

Addressing relative to the 64-bit RIP instruction pointer.

SBZ

Should be zero. An attempt by software to set an SBZ bit to 1 results in undefined behavior. See *reserved*.

scalar

An atomic value existing independently of any specification of location, direction, etc., as opposed to *vectors*.

set

To write a bit value of 1. Compare *clear*.

shadow stack

A shadow stack is a separate, protected stack that is conceptually parallel to the procedure stack and used only by the shadow stack feature.

SIB

A byte following an instruction opcode that specifies address calculation based on scale (S), index (I), and base (B).

SIMD

Single instruction, multiple data. See *vector*.

Streaming SIMD Extensions (SSE)

Instructions that operate on scalar or vector (packed) integer and floating point numbers. The SSE instruction set comprises the *legacy SSE* and *extended SSE* instruction sets.

SSE1

Original SSE instruction set. Includes instructions that operate on vector operands in both the MMX and the XMM registers.

SSE2

Extensions to the SSE instruction set.

SSE3

Further extensions to the SSE instruction set.

SSSE3

Further extensions to the SSE instruction set.

SSE4.1

Further extensions to the SSE instruction set.

SSE4.2

Further extensions to the SSE instruction set.

SSE4A

A minor extension to the SSE instruction set adding the instructions EXTRQ, INSERTQ, MOVNTSS, and MOVNTSD.

sticky bit

A bit that is set or cleared by hardware and that remains in that state until explicitly changed by software.

TOP

The x87 top-of-stack pointer.

TSS

Task-state segment.

underflow

The condition in which a floating-point number is smaller in magnitude than the smallest nonzero, positive or negative number that can be represented in the data-type format being used.

vector

(1) A set of integer or floating-point values, called *elements*, that are packed into a single operand. Most of the media instructions support vectors as operands. Vectors are also called *packed* or *SIMD* (single-instruction multiple-data) operands.

(2) An index into an interrupt descriptor table (IDT), used to access exception handlers. Compare *exception*.

VEX

An instruction encoding escape prefix that opens a new extended instruction encoding space, specifies a 64-bit operand size, and provides access to additional registers. See *XOP prefix*.

virtual-8086 mode

A submode of *legacy mode*.

VMCB

Virtual machine control block.

VMM
: Virtual machine monitor.

word
: Two bytes, or 16 bits.

XOP instructions
: Part of the extended SSE instruction set using the XOP prefix. See *Streaming SIMD Extensions*.

XOP prefix
: Extended instruction identifier prefix, used by XOP instructions allowing the specification of up to four operands and 128 or 256-bit operand widths.

Registers

In the following list of registers, the names are used to refer either to a given register or to the contents of that register:

AH–DH
: The high 8-bit AH, BH, CH, and DH registers. Compare *AL–DL.*

AL–DL
: The low 8-bit AL, BL, CL, and DL registers. Compare *AH–DH.*

AL–r15B
: The low 8-bit AL, BL, CL, DL, SIL, DIL, BPL, SPL, and R8B–R15B registers, available in 64-bit mode.

BP
: Base pointer register.

CR*n*
: Control register number *n*.

CS
: Code segment register.

eAX–eSP
: The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers. Compare *rAX–rSP.*

EFER
: Extended features enable register.

eFLAGS
: 16-bit or 32-bit flags register. Compare *rFLAGS*.

EFLAGS
: 32-bit (extended) flags register.

eIP
: 16-bit or 32-bit instruction-pointer register. Compare *rIP*.

EIP
: 32-bit (extended) instruction-pointer register.

FLAGS
: 16-bit flags register.

GDTR
: Global descriptor table register.

GPRs
: General-purpose registers. For the 16-bit data size, these are AX, BX, CX, DX, DI, SI, BP, and SP. For the 32-bit data size, these are EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP. For the 64-bit data size, these include RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and R8–R15.

IDTR
: Interrupt descriptor table register.

IP
: 16-bit instruction-pointer register.

LDTR
: Local descriptor table register.

MSR
: Model-specific register.

r8–r15
: The 8-bit R8B–R15B registers, or the 16-bit R8W–R15W registers, or the 32-bit R8D–R15D registers, or the 64-bit R8–R15 registers.

rAX–rSP
: The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers, or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers, or the 64-bit RAX, RBX, RCX, RDX, RDI, RSI, RBP, and RSP registers. Replace the placeholder *r* with nothing for 16-bit size, “E” for 32-bit size, or “R” for 64-bit size.

RAX
: 64-bit version of the EAX register.

RBP
64-bit version of the EBP register.

RBX
64-bit version of the EBX register.

RCX
64-bit version of the ECX register.

RDI
64-bit version of the EDI register.

RDX
64-bit version of the EDX register.

rFLAGS
16-bit, 32-bit, or 64-bit flags register. Compare *RFLAGS*.

RFLAGS
64-bit flags register. Compare *rFLAGS*.

rIP
16-bit, 32-bit, or 64-bit instruction-pointer register. Compare *RIP*.

RIP
64-bit instruction-pointer register.

RSI
64-bit version of the ESI register.

RSP
64-bit version of the ESP register.

SP
Stack pointer register.

SS
Stack segment register.

SSP
Shadow-stack pointer register.

TPR
Task priority register (CR8), a new register introduced in the AMD64 architecture to speed interrupt management.

TR

Task register.

Endian Order

The x86 and AMD64 architectures address memory using little-endian byte-ordering. Multibyte values are stored with their least-significant byte at the lowest byte address, and they are illustrated with their least significant byte at the right side. Strings are illustrated in reverse order, because the addresses of their bytes increase from right to left.

Related Documents

- Peter Abel, *IBM PC Assembly Language and Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Rakesh Agarwal, *80x86 Architecture & Programming: Volume II*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- AMD data sheets and application notes for particular hardware implementations of the AMD64 architecture.
- AMD, *Software Optimization Guide for AMD Family 15h Processors,* order number 47414.
- AMD, *AMD-K6® MMX™ Enhanced Processor Multimedia Technology*, Sunnyvale, CA, 2000.
- AMD, *3DNow!™ Technology Manual*, Sunnyvale, CA, 2000.
- AMD, *AMD Extensions to the 3DNow!™ and MMX™ Instruction Sets*, Sunnyvale, CA, 2000.
- Don Anderson and Tom Shanley, *Pentium® Processor System Architecture*, Addison-Wesley, New York, 1995.
- Nabajyoti Barkakati and Randall Hyde, *Microsoft Macro Assembler Bible*, Sams, Carmel, Indiana, 1992.
- Barry B. Brey, *8086/8088, 80286, 80386, and 80486 Assembly Language Programming*, Macmillan Publishing Co., New York, 1994.
- Barry B. Brey, *Programming the 80286, 80386, 80486, and Pentium Based Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Ralf Brown and Jim Kyle, *PC Interrupts,* Addison-Wesley, New York, 1994.
- Penn Brumm and Don Brumm, *80386/80486 Assembly Language Programming*, Windcrest McGraw-Hill, 1993.
- Geoff Chappell, *DOS Internals,* Addison-Wesley, New York, 1994.
- Chips and Technologies, Inc. *Super386 DX Programmer's Reference Manual*, Chips and Technologies, Inc., San Jose, 1992.
- John Crawford and Patrick Gelsinger, *Programming the 80386*, Sybex, San Francisco, 1987.
- Cyrix Corporation, *5x86 Processor BIOS Writer's Guide*, Cyrix Corporation, Richardson, TX, 1995.

- Cyrix Corporation, *M1 Processor Data Book*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor MMX Extension Opcode Table*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor Data Book*, Cyrix Corporation, Richardson, TX, 1997.
- Ray Duncan, *Extending DOS: A Programmer's Guide to Protected-Mode DOS*, Addison Wesley, NY, 1991.
- William B. Giles, *Assembly Language Programming for the Intel 80xxx Family*, Macmillan, New York, 1991.
- Frank van Gilluwe, *The Undocumented PC,* Addison-Wesley, New York, 1994.
- John L. Hennessy and David A. Patterson, *Computer Architecture*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.
- Thom Hogan, *The Programmer's PC Sourcebook*, Microsoft Press, Redmond, WA, 1991.
- Hal Katircioglu, *Inside the 486, Pentium®, and Pentium Pro*, Peer-to-Peer Communications, Menlo Park, CA, 1997.
- IBM Corporation, *486SLC Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *486SLC2 Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *80486DX2 Processor Floating Point Instructions*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *80486DX2 Processor BIOS Writer's Guide*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *Blue Lightning 486DX2 Data Book*, IBM Corporation, Essex Junction, VT, 1994.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, ANSI/IEEE Std 854-1987.
- Muhammad Ali Mazidi and Janice Gillispie Mazidi, *80X86 IBM PC and Compatible Computers*, Prentice-Hall, Englewood Cliffs, NJ, 1997.
- Hans-Peter Messmer, *The Indispensable Pentium Book,* Addison-Wesley, New York, 1995.
- Karen Miller, *An Assembly Language Introduction to Computer Architecture: Using the Intel Pentium®*, Oxford University Press, New York, 1999.
- Stephen Morse, Eric Isaacson, and Douglas Albert, *The 80386/387 Architecture*, John Wiley & Sons, New York, 1987.
- NexGen Inc.*, Nx586 Processor Data Book*, NexGen Inc., Milpitas, CA, 1993.
- NexGen Inc.*, Nx686 Processor Data Book*, NexGen Inc., Milpitas, CA, 1994.

- Bipin Patwardhan, *Introduction to the Streaming SIMD Extensions in the Pentium® III*, www.x86.org/articles/sse_pt1/ simd1.htm, June, 2000.
- Peter Norton, Peter Aitken, and Richard Wilton, *PC Programmer's Bible,* Microsoft® Press, Redmond, WA, 1993.
- *PharLap 386|ASM Reference Manual*, Pharlap, Cambridge MA, 1993.
- *PharLap TNT DOS-Extender Reference Manual*, Pharlap, Cambridge MA, 1995.
- Sen-Cuo Ro and Sheau-Chuen Her, *i386/i486 Advanced Programming*, Van Nostrand Reinhold, New York, 1993.
- Jeffrey P. Royer, *Introduction to Protected Mode Programming*, course materials for an onsite class, 1992.
- Tom Shanley, *Protected Mode System Architecture*, Addison Wesley, NY, 1996.
- SGS-Thomson Corporation, *80486DX Processor SMM Programming Manual*, SGS-Thomson Corporation, 1995.
- Walter A. Triebel, *The 80386DX Microprocessor*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- John Wharton, *The Complete x86*, MicroDesign Resources, Sebastopol, California, 1994.
- Web sites and newsgroups:
  - www.amd.com
  - news.comp.arch
  - news.comp.lang.asm.x86
  - news.intel.microprocessors
  - news.microsoft

1 Overview of the AMD64 Architecture

1.1 Introduction

The AMD64 architecture is a simple yet powerful 64-bit, backward-compatible extension of the industry-standard (legacy) x86 architecture. It adds 64-bit addressing and expands register resources to support higher performance for recompiled 64-bit programs, while supporting legacy 16-bit and 32-bit applications and operating systems without modification or recompilation. It is the architectural basis on which new processors can provide seamless, high-performance support for both the vast body of existing software and 64-bit software required for higher-performance applications.

The need for a 64-bit x86 architecture is driven by applications that address large amounts of virtual and physical memory, such as high-performance servers, database management systems, and CAD tools. These applications benefit from both 64-bit addresses and an increased number of registers. The small number of registers available in the legacy x86 architecture limits performance in computation-intensive applications. Increasing the number of registers provides a performance boost to many such applications.

1.1.1 AMD64 Features

The AMD64 architecture includes these features:

- Register Extensions (see Figure 1-1 on page 2):
  - 8 additional general-purpose registers (GPRs).
  - All 16 GPRs are 64 bits wide.
  - 8 additional YMM/XMM registers.
  - Uniform byte-register addressing for all GPRs.
  - An instruction prefix (REX) accesses the extended registers.
- Long Mode (see Table 1-1 on page 2):
  - Up to 64 bits of virtual address.
  - 64-bit instruction pointer (RIP).
  - Instruction-pointer-relative data-addressing mode.
  - Flat address space.

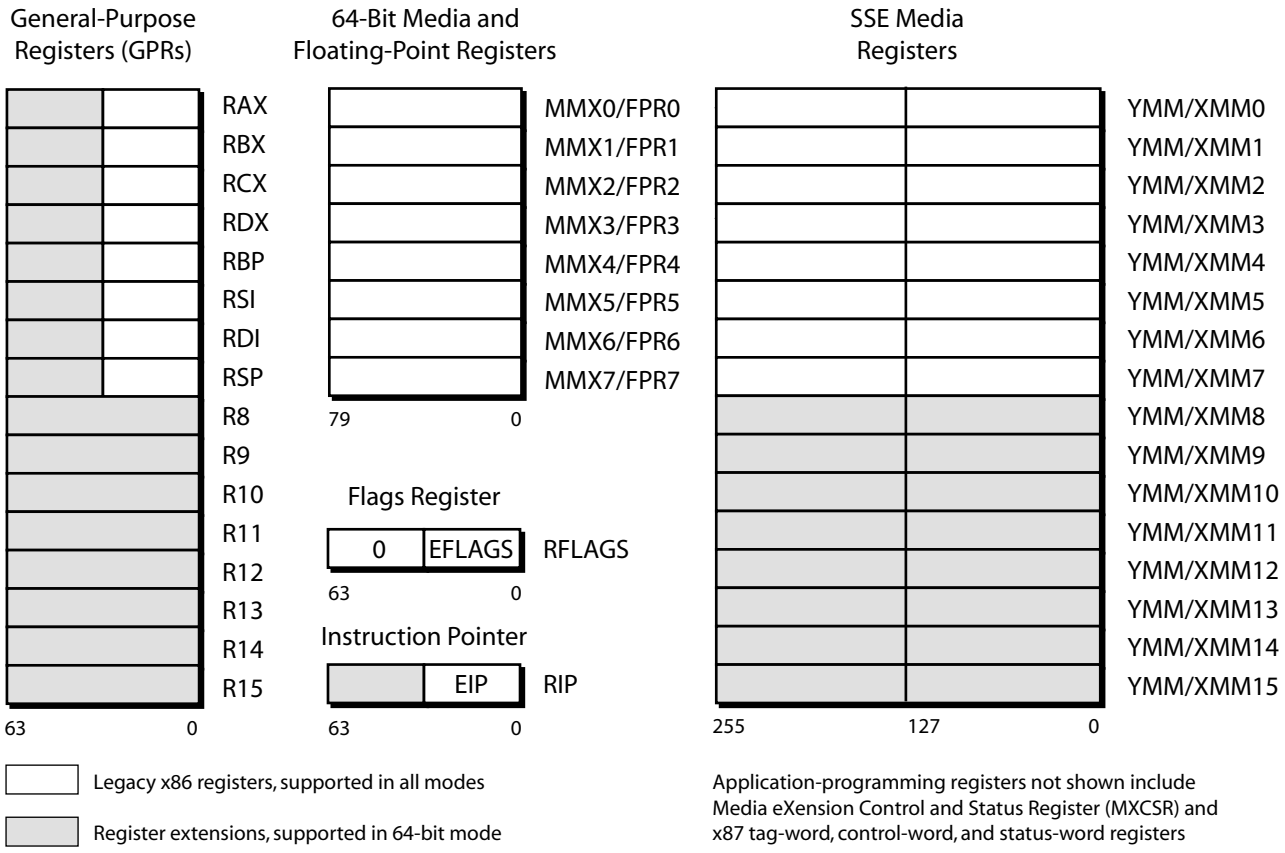


Figure 1-1. Application-Programming Register Set

Table 1-1. Operating Modes

Operating Mode		Operating System Required	Application Recompile Required	Defaults		Register Extensions	Typical
				Address Size (bits)	Operand Size (bits)		GPR Width (bits)
Long Mode	64-Bit Mode	64-bit OS	yes	64	32	yes	64
	Compatibility Mode		no	32		no	32
				16	16		16
Legacy Mode	Protected Mode	Legacy 32-bit OS	no	32	32	no	32
				16	16		
	Virtual-8086 Mode			16	16		16
	Real Mode	Legacy 16-bit OS					

1.1.2 Registers

Table 1-2 compares the register and stack resources available to application software, by operating mode. The left set of columns shows the legacy x86 resources, which are available in the AMD64 architecture's legacy and compatibility modes. The right set of columns shows the comparable resources in 64-bit mode. Gray shading indicates differences between the modes. These register differences (not including stack-width difference) represent the *register extensions* shown in Figure 1-1.

Table 1-2. Application Registers and Stack, by Operating Mode

Register or Stack	Legacy and Compatibility Modes			64-Bit Mode[1]		
	Name	Number	Size (bits)	Name	Number	Size (bits)
General-Purpose Registers (GPRs)[2]	EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP	8	32	RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8–R15	16	64
256-bit YMM Registers	YMM0–YMM7[3]	8	256	YMM0–YMM15[3]	16	256
128-Bit XMM Registers	XMM0–XMM7[3]	8	128	XMM0–XMM15[3]	16	128
64-Bit MMX Registers	MMX0–MMX7[4]	8	64	MMX0–MMX7[4]	8	64
x87 Registers	FPR0–FPR7[4]	8	80	FPR0–FPR7[4]	8	80
Instruction Pointer[2]	EIP	1	32	RIP	1	64
Flags[2]	EFLAGS	1	32	RFLAGS	1	64
Stack	—		16 or 32	—		64

Note:

1. *Gray-shaded entries indicate differences between the modes. These differences (except stack-width difference) are the AMD64 architecture's register extensions.*
2. *GPRs are listed using their full-width names. In legacy and compatibility modes,16-bit and 8-bit mappings of the registers are also accessible. In 64-bit mode, 32-bit, 16-bit, and 8-bit mappings of the registers are accessible. See Section 3.1. "Registers" on page 23.*
3. *The XMM registers overlay the lower octword of the YMM registers. See Section 4.2. "Registers" on page 113.*
4. *The MMX0–MMX7 registers are mapped onto the FPR0–FPR7 physical registers, as shown in Figure 1-1. The x87 stack registers, ST(0)–ST(7), are the logical mappings of the FPR0–FPR7 physical registers.*

As Table 1-2 shows, the legacy x86 architecture (called *legacy mode* in the AMD64 architecture) supports eight GPRs. In reality, however, the general use of at least four registers (EBP, ESI, EDI, and ESP) is compromised because they serve special purposes when executing many instructions. The AMD64 architecture's addition of eight GPRs—and the increased width of these registers from 32 bits to 64 bits—allows compilers to substantially improve software performance. Compilers have more flexibility in using registers to hold variables. Compilers can also minimize memory traffic—and thus boost performance—by localizing work within the GPRs.

1.1.3 Instruction Set

The AMD64 architecture supports the full legacy x86 instruction set, with additional instructions to support long mode (see Table 1-1 on page 2 for a summary of operating modes). The application-programming instructions are organized into four subsets, as follows:

- *General-Purpose Instructions*—These are the basic x86 integer instructions used in virtually all programs. Most of these instructions load, store, or operate on data located in the general-purpose registers (GPRs) or memory. Some of the instructions alter sequential program flow by branching to other program locations.
- *Streaming SIMD Extensions Instructions (SSE)*—These instructions load, store, or operate on data located primarily in the YMM/XMM registers. 128-bit media instructions operate on the lower half of the YMM registers. SSE instructions perform integer and floating-point operations on vector (packed) and scalar data types. Because the vector instructions can independently and simultaneously perform a single operation on multiple sets of data, they are called *single-instruction, multiple-data* (SIMD) instructions. They are useful for high-performance media and scientific applications that operate on blocks of data.
- *Multimedia Extension Instructions*—These include the MMX™ technology and AMD 3DNow!™ technology instructions. These instructions load, store, or operate on data located primarily in the 64-bit MMX registers which are mapped onto the 80-bit x87 floating-point registers. Like the SSE instructions, they perform integer and floating-point operations on vector (packed) and scalar data types. These instructions are useful in media applications that do not require high precision. Multimedia Extension Instructions use saturating mathematical operations that do not generate operation exceptions. AMD has de-emphasized the use of 3DNow! instructions, which have been superceded by their more efficient SSE counterparts. Relevant recommendations are provided in Chapter 5, “64-Bit Media Programming” on page 239, and in the *AMD64 Programmer’s Manual Volume 4: 64-Bit Media and x87 Floating-Point Instructions*.
- *x87 Floating-Point Instructions*—These are the floating-point instructions used in legacy x87 applications. They load, store, or operate on data located in the 80-bit x87 registers.

Some of these application-programming instructions bridge two or more of the above subsets. For example, there are instructions that move data between the general-purpose registers and the YMM/XMM or MMX registers, and many of the integer vector (packed) instructions can operate on either YMM/XMM or MMX registers, although not simultaneously. If instructions bridge two or more subsets, their descriptions are repeated in all subsets to which they apply.

1.1.4 Media Instructions

Media applications—such as image processing, music synthesis, speech recognition, full-motion video, and 3D graphics rendering—share certain characteristics:

- They process large amounts of data.
- They often perform the same sequence of operations repeatedly across the data.
- The data are often represented as small quantities, such as 8 bits for pixel values, 16 bits for audio samples, and 32 bits for object coordinates in floating-point format.

SSE and MMX instructions are designed to accelerate these applications. The instructions use a form of vector (or packed) parallel processing known as single-instruction, multiple data (SIMD) processing. This vector technology has the following characteristics:

- A single register can hold multiple independent pieces of data. For example, a single YMM register can hold 32 8-bit integer data elements, or eight 32-bit single-precision floating-point data elements.
- The vector instructions can operate on all data elements in a register, independently and simultaneously. For example, a PADDB instruction operating on byte elements of two vector operands in 128-bit XMM registers performs 16 simultaneous additions and returns 16 independent results in a single operation.

SSE and MMX instructions take SIMD vector technology a step further by including special instructions that perform operations commonly found in media applications. For example, a graphics application that adds the brightness values of two pixels must prevent the add operation from wrapping around to a small value if the result overflows the destination register, because an overflow result can produce unexpected effects such as a dark pixel where a bright one is expected. These instructions include saturating-arithmetic instructions to simplify this type of operation. A result that otherwise would wrap around due to overflow or underflow is instead forced to saturate at the largest or smallest value that can be represented in the destination register.

1.1.5 Floating-Point Instructions

The AMD64 architecture provides three floating-point instruction subsets, using three distinct register sets:

- SSE instructions support 32-bit single-precision and 64-bit double-precision floating-point operations, in addition to integer operations. Operations on both vector data and scalar data are supported, with a dedicated floating-point exception-reporting mechanism. These floating-point operations comply with the IEEE-754 standard.
- MMX Instructions support single-precision floating-point operations. Operations on both vector data and scalar data are supported, but these instructions do not support floating-point exception reporting.
- x87 Floating-Point Instructions support single-precision, double-precision, and 80-bit extended-precision floating-point operations. Only scalar data are supported, with a dedicated floating-point exception-reporting mechanism. The x87 floating-point instructions contain special instructions for performing trigonometric and logarithmic transcendental operations. The single-precision and double-precision floating-point operations comply with the IEEE-754 standard.

Maximum floating-point performance can be achieved using the 256-bit media instructions. One of these vector instructions can support up to eight single-precision (or four double-precision) operations in parallel. A total of 16 256-bit YMM registers, available in 64-bit mode, speeds up applications by providing more registers to hold intermediate results, thus reducing the need to store these results in memory. Fewer loads and stores results in better performance.

1.2 Modes of Operation

Table 1-1 on page 2 summarizes the modes of operation supported by the AMD64 architecture. In most cases, the default address and operand sizes can be overridden with instruction prefixes. The register extensions shown in the second-from-right column of Table 1-1 are those illustrated in Figure 1-1 on page 2.

1.2.1 Long Mode

Long mode is an extension of legacy protected mode. Long mode consists of two submodes: *64-bit mode* and *compatibility mode*. 64-bit mode supports all of the features and register extensions of the AMD64 architecture. Compatibility mode supports binary compatibility with existing 16-bit and 32-bit applications. Long mode does not support legacy real mode or legacy virtual-8086 mode, and it does not support hardware task switching.

Throughout this document, references to *long mode* refer to both *64-bit mode* and *compatibility mode*. If a function is specific to either of these submodes, then the name of the specific submode is used instead of the name *long mode*.

1.2.2 64-Bit Mode

64-bit mode—a submode of long mode—supports the full range of 64-bit virtual-addressing and register-extension features. This mode is enabled by the operating system on an individual code-segment basis. Because 64-bit mode supports a 64-bit virtual-address space, it requires a 64-bit operating system and tool chain. Existing application binaries can run without recompilation in compatibility mode, under an operating system that runs in 64-bit mode, or the applications can also be recompiled to run in 64-bit mode.

Addressing features include a 64-bit instruction pointer (RIP) and an RIP-relative data-addressing mode. This mode accommodates modern operating systems by supporting only a flat address space, with single code, data, and stack space.

Register Extensions. 64-bit mode implements register extensions through a group of instruction prefixes, called REX prefixes. These extensions add eight GPRs (R8–R15), widen all GPRs to 64 bits, and add eight YMM/XMM registers (YMM/XMM8–15).

The REX instruction prefixes also provide a byte-register capability that makes the low byte of any of the sixteen GPRs available for byte operations. This results in a uniform set of byte, word, doubleword, and quadword registers that is better suited to compiler register-allocation.

64-Bit Addresses and Operands. In 64-bit mode, the default virtual-address size is 64 bits (implementations can have fewer). The default operand size for most instructions is 32 bits. For most instructions, these defaults can be overridden on an instruction-by-instruction basis using instruction prefixes. REX prefixes specify the 64-bit operand size and register extensions.

RIP-Relative Data Addressing. 64-bit mode supports data addressing relative to the 64-bit instruction pointer (RIP). The legacy x86 architecture supports IP-relative addressing only in control-

transfer instructions. RIP-relative addressing improves the efficiency of position-independent code and code that addresses global data.

Opcodes. A few instruction opcodes and prefix bytes are redefined to allow register extensions and 64-bit addressing. These differences are described in Appendix B “General-Purpose Instructions in 64-Bit Mode” and Appendix C “Differences Between Long Mode and Legacy Mode” in Volume 3.

1.2.3 Compatibility Mode

Compatibility mode—the second submode of long mode—allows 64-bit operating systems to run existing 16-bit and 32-bit x86 applications. These legacy applications run in compatibility mode without recompilation.

Applications running in compatibility mode use 32-bit or 16-bit addressing and can access the first 4GB of virtual-address space. Legacy x86 instruction prefixes toggle between 16-bit and 32-bit address and operand sizes.

As with 64-bit mode, compatibility mode is enabled by the operating system on an individual code-segment basis. Unlike 64-bit mode, however, x86 segmentation functions the same as in the legacy x86 architecture, using 16-bit or 32-bit protected-mode semantics. From the application viewpoint, compatibility mode looks like the legacy x86 protected-mode environment. From the operating-system viewpoint, however, address translation, interrupt and exception handling, and system data structures use the 64-bit long-mode mechanisms.

1.2.4 Legacy Mode

Legacy mode preserves binary compatibility not only with existing 16-bit and 32-bit applications but also with existing 16-bit and 32-bit operating systems. Legacy mode consists of the following three submodes:

- *Protected Mode*—Protected mode supports 16-bit and 32-bit programs with memory segmentation, optional paging, and privilege-checking. Programs running in protected mode can access up to 4GB of memory space.
- *Virtual-8086 Mode*—Virtual-8086 mode supports 16-bit real-mode programs running as tasks under protected mode. It uses a simple form of memory segmentation, optional paging, and limited protection-checking. Programs running in virtual-8086 mode can access up to 1MB of memory space.
- *Real Mode*—Real mode supports 16-bit programs using simple register-based memory segmentation. It does not support paging or protection-checking. Programs running in real mode can access up to 1MB of memory space.

Legacy mode is compatible with existing 32-bit processor implementations of the x86 architecture. Processors that implement the AMD64 architecture boot in legacy real mode, just like processors that implement the legacy x86 architecture.

Throughout this document, references to *legacy mode* refer to all three submodes—*protected mode*, *virtual-8086 mode*, and *real mode*. If a function is specific to either of these submodes, then the name of the specific submode is used instead of the name *legacy mode*.

2 Memory Model

This chapter describes the memory characteristics that apply to application software in the various operating modes of the AMD64 architecture. These characteristics apply to all instructions in the architecture. Several additional system-level details about memory and cache management are described in Volume 2.

2.1 Memory Organization

2.1.1 Virtual Memory

Virtual memory consists of the entire address space available to programs. It is a large linear-address space that is translated by a combination of hardware and operating-system software to a smaller physical-address space, parts of which are located in memory and parts on disk or other external storage media.

Figure 2-1 on page 10 shows how the virtual-memory space is treated in the two submodes of long mode:

- *64-bit mode*—This mode uses a flat segmentation model of virtual memory. The 64-bit virtual-memory space is treated as a single, flat (unsegmented) address space. Program addresses access locations that can be anywhere in the linear 64-bit address space. The operating system can use separate selectors for code, stack, and data segments for memory-protection purposes, but the base address of all these segments is always 0. (For an exception to this general rule, see “FS and GS as Base of Address Calculation” on page 17.)
- *Compatibility mode*—This mode uses a protected, multi-segment model of virtual memory, just as in legacy protected mode. The 32-bit virtual-memory space is treated as a segmented set of address spaces for code, stack, and data segments, each with its own base address and protection parameters. A segmented space is specified by adding a segment selector to an address.

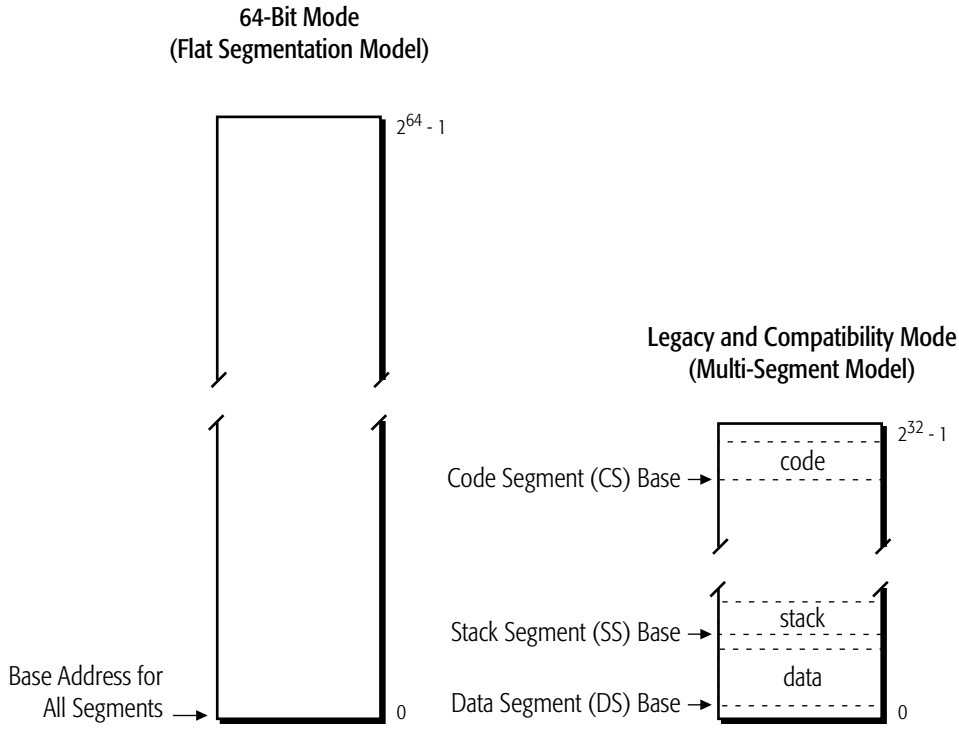


Figure 2-1. Virtual-Memory Segmentation

Operating systems have used segmented memory as a method to isolate programs from the data they used, in an effort to increase the reliability of systems running multiple programs simultaneously. However, most modern operating systems do not use the segmentation features available in the legacy x86 architecture. Instead, these operating systems handle segmentation functions entirely in software. For this reason, the AMD64 architecture dispenses with most of the legacy segmentation functions in 64-bit mode. This allows 64-bit operating systems to be coded more simply, and it supports more efficient management of multi-tasking environments than is possible in the legacy x86 architecture.

2.1.2 Segment Registers

Segment registers hold the selectors used to access memory segments. Figure 2-2 on page 11 shows the application-visible portion of the segment registers. In legacy and compatibility modes, all segment registers are accessible to software. In 64-bit mode, only the CS, FS, and GS segments are recognized by the processor, and software can use the FS and GS segment-base registers as base registers for address calculation, as described in “FS and GS as Base of Address Calculation” on page 17. For references to the DS, ES, or SS segments in 64-bit mode, the processor assumes that the base for each of these segments is zero, neither their segment limit nor attributes are checked, and the processor simply checks that all such addresses are in canonical form, as described in “64-Bit Canonical Addresses” on page 15.

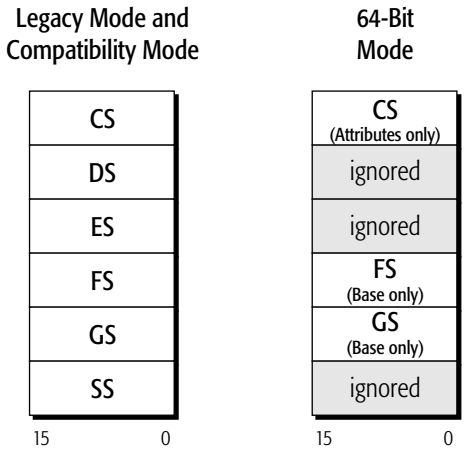


Figure 2-2. Segment Registers

For details on segmentation and the segment registers, see "Segmented Virtual Memory" in Volume 2.

2.1.3 Physical Memory

Physical memory is the installed memory (excluding cache memory) in a particular computer system that can be accessed through the processor's bus interface. The maximum size of the physical memory space is determined by the number of address bits on the bus interface. In a virtual-memory system, the large virtual-address space (also called *linear-address space*) is translated to a smaller physical-address space by a combination of segmentation and paging hardware and software.

Segmentation is illustrated in Figure 2-1 on page 10. Paging is a mechanism for translating linear (virtual) addresses into fixed-size blocks called *pages*, which the operating system can move, as needed, between memory and external storage media (typically disk). The AMD64 architecture supports an expanded version of the legacy x86 paging mechanism, one that is able to translate the full 64-bit virtual-address space into the physical-address space supported by the particular implementation.

2.1.4 Memory Management

Memory management strategies translate addresses generated by programs into addresses in physical memory using segmentation and/or paging. Memory management is not visible to application programs. It is handled by the operating system and processor hardware. The following description gives a very brief overview of these functions. Details are given in "System-Management Instructions" in Volume 2.

2.1.4.1 Long-Mode Memory Management

Figure 2-3 shows the flow, from top to bottom, of memory management functions performed in the two submodes of long mode.

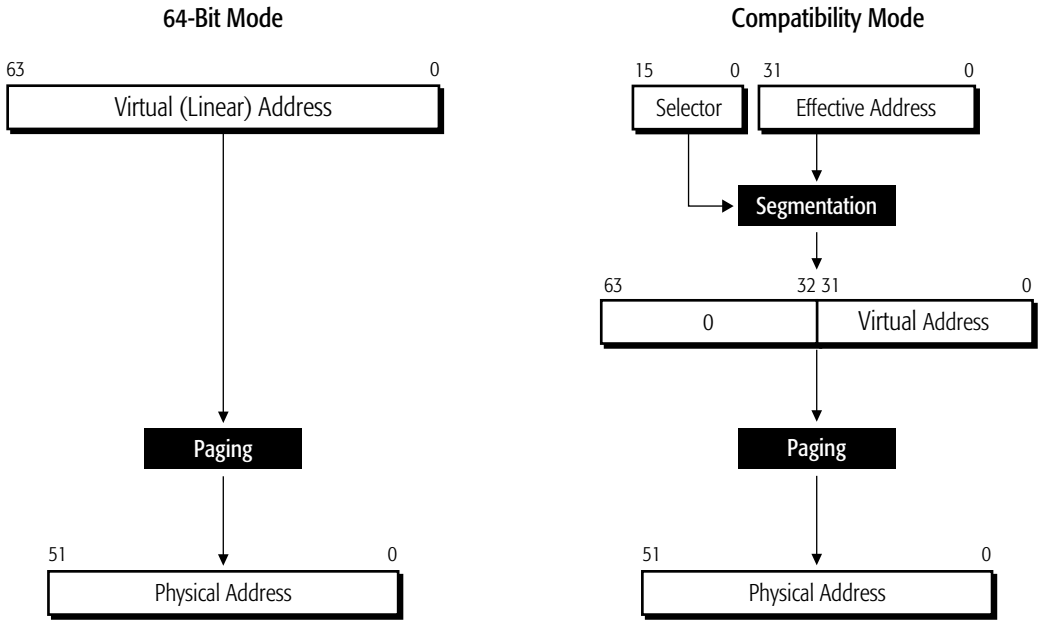


Figure 2-3. Long-Mode Memory Management

In 64-bit mode, programs generate virtual (linear) addresses that can be up to 64 bits in size. The virtual addresses are passed to the long-mode paging function, which generates physical addresses that can be up to 52 bits in size. (Specific implementations of the architecture can support smaller virtual-address and physical-address sizes.)

In compatibility mode, legacy 16-bit and 32-bit applications run using legacy x86 protected-mode segmentation semantics. The 16-bit or 32-bit effective addresses generated by programs are combined with their segments to produce 32-bit virtual (linear) addresses that are zero-extended to a maximum of 64 bits. The paging that follows is the same long-mode paging function used in 64-bit mode. It translates the virtual addresses into physical addresses. The combination of segment selector and effective address is also called a *logical address* or *far pointer*. The *virtual address* is also called the *linear address*.

2.1.4.2 Legacy-Mode Memory Management

Figure 2-4 on page 13 shows the memory-management functions performed in the three submodes of legacy mode.

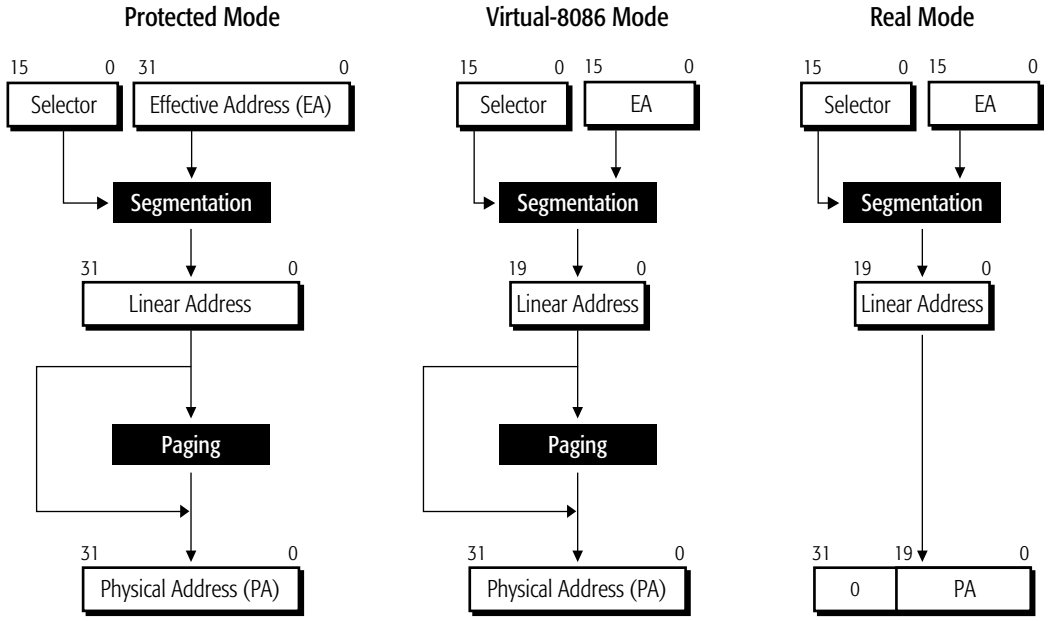


Figure 2-4. Legacy-Mode Memory Management

The memory-management functions differ, depending on the submode, as follows:

- *Protected Mode*—Protected mode supports 16-bit and 32-bit programs with table-based memory segmentation, paging, and privilege-checking. The segmentation function takes 32-bit effective addresses and 16-bit segment selectors and produces 32-bit linear addresses into one of 16K memory segments, each of which can be up to 4GB in size. Paging is optional. The 32-bit physical addresses are either produced by the paging function or the linear addresses are used without modification as physical addresses.
- *Virtual-8086 Mode*—Virtual-8086 mode supports 16-bit programs running as tasks under protected mode. 20-bit linear addresses are formed in the same way as in real mode, but they can optionally be translated through the paging function to form 32-bit physical addresses that access up to 4GB of memory space.
- *Real Mode*—Real mode supports 16-bit programs using register-based shift-and-add segmentation, but it does not support paging. Sixteen-bit effective addresses are zero-extended and added to a 16-bit segment-base address that is left-shifted four bits, producing a 20-bit linear address. The linear address is zero-extended to a 32-bit physical address that can access up to 1MB of memory space.

2.2 Memory Addressing

2.2.1 Byte Ordering

Instructions and data are stored in memory in *little-endian* byte order. Little-endian ordering places the least-significant byte of the instruction or data item at the lowest memory address and the most-significant byte at the highest memory address.

Figure 2-5 shows a generalization of little-endian memory and register images of a quadword data type. The least-significant byte is at the lowest address in memory and at the right-most byte location of the register image.

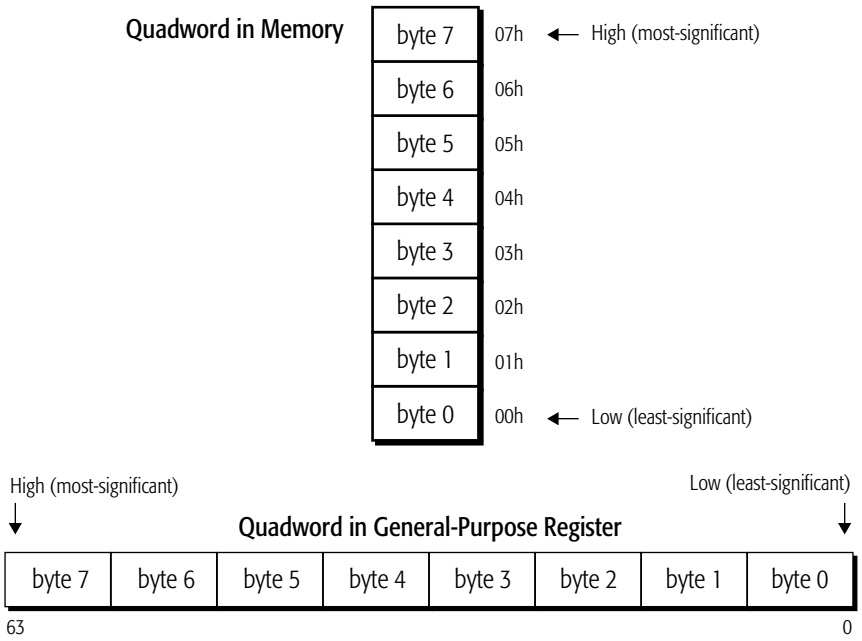


Figure 2-5. Byte Ordering

Figure 2-6 on page 15 shows the memory image of a 10-byte instruction. Instructions are byte data types. They are read from memory one byte at a time, starting with the least-significant byte (lowest address). For example, the following instruction specifies the 64-bit instruction MOV RAX, 1122334455667788 instruction that consists of the following ten bytes:

```
48 B8 8877665544332211
```

48 is a REX instruction prefix that specifies a 64-bit operand size, B8 is the opcode that—together with the REX prefix—specifies the 64-bit RAX destination register, and 8877665544332211 is the 8-byte immediate value to be moved, where 88 represents the eighth (least-significant) byte and 11 represents the first (most-significant) byte. In memory, the REX prefix byte (48) would be stored at the lowest address, and the first immediate byte (11) would be stored at the highest instruction address.

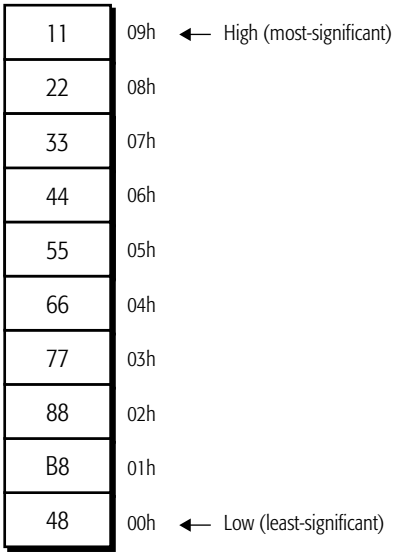


Figure 2-6. Example of 10-Byte Instruction in Memory

2.2.2 64-Bit Canonical Addresses

Long mode defines 64 bits of virtual address, but implementations of the AMD64 architecture may support fewer bits of virtual address. Although implementations might not use all 64 bits of the virtual address, they check bits 63 through the most-significant implemented bit to see if those bits are all zeros or all ones. An address that complies with this property is said to be in *canonical address form*. If a virtual-memory reference is not in canonical form, the implementation causes a general-protection exception or stack fault.

2.2.3 Effective Addresses

Programs provide effective addresses to the hardware prior to segmentation and paging translations. Long-mode effective addresses are a maximum of 64 bits wide, as shown in Figure 2-3 on page 12. Programs running in compatibility mode generate (by default) 32-bit effective addresses, which the hardware zero-extends to 64 bits. Legacy-mode effective addresses, with no address-size override, are 32 or 16 bits wide, as shown in Figure 2-4 on page 13. These sizes can be overridden with an address-size instruction prefix, as described in “Instruction Prefixes” on page 76.

There are five methods for generating effective addresses, depending on the specific instruction encoding:

- *Absolute Addresses*—These addresses are given as displacements (or offsets) from the base address of a data segment. They point directly to a memory location in the data segment.
- *Instruction-Relative Addresses*—These addresses are given as displacements (or offsets) from the current instruction pointer (IP), also called the program counter (PC). They are generated by control-transfer instructions. A displacement in the instruction encoding, or one read from

memory, serves as an offset from the address that follows the transfer. See “RIP-Relative Addressing” on page 18 for details about RIP-relative addressing in 64-bit mode.

- *Indexed Register-Indirect Addresses*—These addresses are calculated off a base address contained in a general-purpose register specified by the instruction (base). Different encodings allow offsets from this base using a signed displacement or using the sum of the displacement and a scaled index value. Instruction encodings may utilize up to ten bytes—the ModRM byte, the optional SIB (scale, index, base) byte and a variable length displacement—to specify the values to be used in the effective address calculation. The base and index values are contained in general-purpose registers specified by the SIB byte. The scale and displacement values are specified directly in the instruction encoding. Figure 2-7 shows the components of the address calculation. The resultant effective address is added to the data-segment base address to form a linear address, as described in “Segmented Virtual Memory” in Volume 2. “Instruction Formats” in Volume 3 gives further details on specifying this form of address.

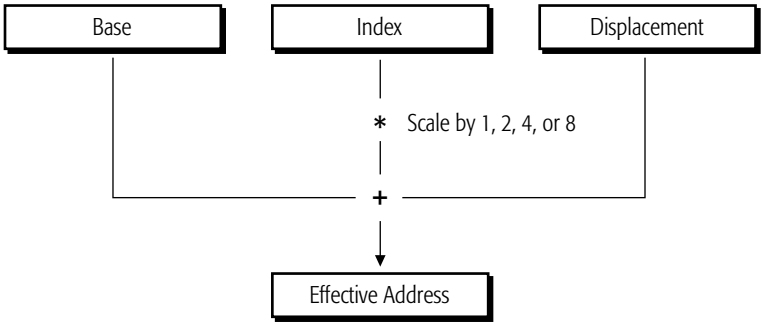


Figure 2-7. Complex Address Calculation (Protected Mode)

- *Stack Addresses*—PUSH, POP, CALL, RET, IRET, and INT instructions implicitly use the stack pointer, which contains the address of the procedure stack. See “Stack Operation” on page 19 for details about the size of the stack pointer.
- *String Addresses*—String instructions generate sequential addresses using the rDI and rSI registers, as described in “Implicit Uses of GPRs” on page 30.

In 64-bit mode, with no address-size override, the size of effective-address calculations is 64 bits. An effective-address calculation uses 64-bit base and index registers and sign-extends displacements to 64 bits. Due to the flat address space in 64-bit mode, virtual addresses are equal to effective addresses. (For an exception to this general rule, see “FS and GS as Base of Address Calculation” on page 17.)

2.2.3.1 Long-Mode Zero-Extension of 16-Bit and 32-Bit Addresses

In long mode, all 16-bit and 32-bit address calculations are zero-extended to form 64-bit addresses. Address calculations are first truncated to the effective-address size of the current mode (64-bit mode or compatibility mode), as overridden by any address-size prefix. The result is then zero-extended to the full 64-bit address width.

Because of this, 16-bit and 32-bit applications running in compatibility mode can access only the low 4GB of the long-mode virtual-address space. Likewise, a 32-bit address generated in 64-bit mode can access only the low 4GB of the long-mode virtual-address space.

2.2.3.2 Displacements and Immediates

In general, the maximum size of address displacements and immediate operands is 32 bits. They can be 8, 16, or 32 bits in size, depending on the instruction or, for displacements, the effective address size. In 64-bit mode, displacements are sign-extended to 64 bits during use, but their actual size (for value representation) remains a maximum of 32 bits. The same is true for immediates in 64-bit mode, when the operand size is 64 bits. However, support is provided in 64-bit mode for some 64-bit displacement and immediate forms of the MOV instruction.

2.2.3.3 FS and GS as Base of Address Calculation

In 64-bit mode, the FS and GS segment-base registers (unlike the DS, ES, and SS segment-base registers) can be used as non-zero data-segment base registers for address calculations, as described in "Segmented Virtual Memory" in Volume 2. 64-bit mode assumes all other data-segment registers (DS, ES, and SS) have a base address of 0.

2.2.4 Address-Size Prefix

The default address size of an instruction is determined by the default-size (D) bit and long-mode (L) bit in the current code-segment descriptor (for details, see "Segmented Virtual Memory" in Volume 2). Application software can override the default address size in any operating mode by using the 67h address-size instruction prefix byte. The address-size prefix allows mixing 32-bit and 64-bit addresses on an instruction-by-instruction basis.

Table 2-1 on page 18 shows the effects of using the address-size prefix in all operating modes. In 64-bit mode, the default address size is 64 bits. The address size can be overridden to 32 bits. 16-bit addresses are not supported in 64-bit mode. In compatibility and legacy modes, the address-size prefix works the same as in the legacy x86 architecture.

Table 2-1. Address-Size Prefixes

Operating Mode		Default Address Size (Bits)	Effective Address Size (Bits)	Address-Size Prefix (67h)[1] Required?
Long Mode	64-Bit Mode	64	64	no
			32	yes
	Compatibility Mode	32	32	no
			16	yes
		16	32	yes
			16	no
Legacy Mode (Protected, Virtual-8086, or Real Mode)		32	32	no
			16	yes
		16	32	yes
			16	no

Note:
1. *"No" indicates that the default address size is used.*

2.2.5 RIP-Relative Addressing

RIP-relative addressing—that is, addressing relative to the 64-bit instruction pointer (also called program counter)—is available in 64-bit mode. The effective address is formed by adding the displacement to the 64-bit RIP of the next instruction.

In the legacy x86 architecture, addressing relative to the instruction pointer (IP or EIP) is available only in control-transfer instructions. In the 64-bit mode, any instruction that uses ModRM addressing (see "ModRM and SIB Bytes" in Volume 3) can use RIP-relative addressing. The feature is particularly useful for addressing data in position-independent code and for code that addresses global data.

Programs usually have many references to data, especially global data, that are not register-based. To load such a program, the loader typically selects a location for the program in memory and then adjusts the program's references to global data based on the load location. RIP-relative addressing of data makes this adjustment unnecessary.

2.2.5.1 Range of RIP-Relative Addressing

Without RIP-relative addressing, instructions encoded with a ModRM byte address memory relative to zero. With RIP-relative addressing, instructions with a ModRM byte can address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of ±2 GBytes from the RIP.

2.2.5.2 Effect of Address-Size Prefix on RIP-Relative Addressing

RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. Conversely, use of the address-size prefix does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits, like any other addressing mode.

2.2.5.3 Encoding

For details on instruction encoding of RIP-relative addressing, see in "Encoding for RIP-Relative Addressing" in Volume 3.

2.3 Pointers

Pointers are variables that contain addresses rather than data. They are used by instructions to reference memory. Instructions access data using near and far pointers. Stack pointers locate the current stack.

2.3.1 Near and Far Pointers

Near pointers contain only an effective address, which is used as an offset into the current segment. Far pointers contain both an effective address and a segment selector that specifies one of several segments. Figure 2-8 illustrates the two types of pointers.

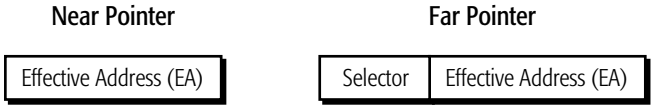


Figure 2-8. Near and Far Pointers

In 64-bit mode, the AMD64 architecture supports only the flat-memory model in which there is only one data segment, so the effective address is used as the virtual (linear) address and far pointers are not needed. In compatibility mode and legacy protected mode, the AMD64 architecture supports multiple memory segments, so effective addresses can be combined with segment selectors to form far pointers, and the terms *logical address* (segment selector and effective address) and *far pointer* are synonyms. Near pointers can also be used in compatibility mode and legacy mode.

2.4 Stack Operation

A procedure stack (also known as a '*program stack*') is a portion of a stack segment in memory that is used to link procedures. Software conventions typically define stacks using a *stack frame*, which consists of two registers—a *stack-frame base pointer* (rBP) and a *stack pointer* (rSP)—as shown in Figure 2-9 on page 20. These stack pointers can be either near pointers or far pointers.

The stack-segment (SS) register, points to the base address of the current stack segment. The stack pointers contain offsets from the base address of the current stack segment. All instructions that address memory using the rBP or rSP registers cause the processor to access the current stack segment.

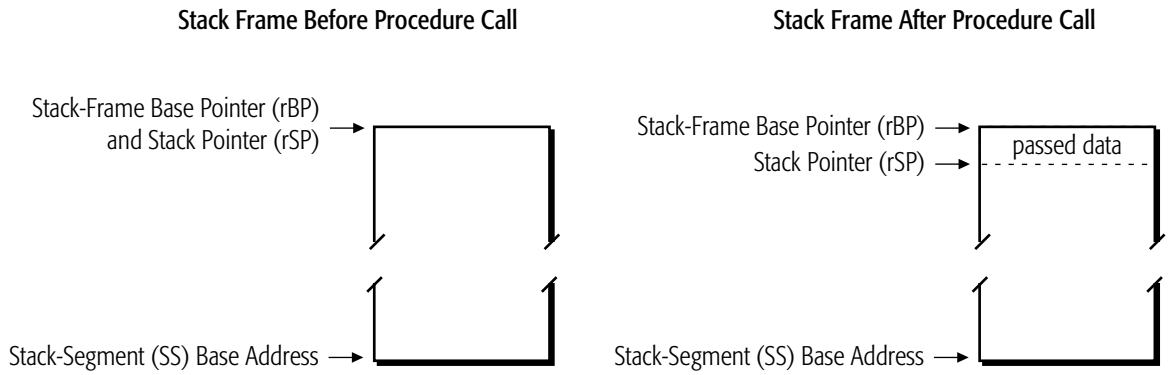


Figure 2-9. Stack Pointer Mechanism

In typical APIs, the stack-frame base pointer and the stack pointer point to the same location before a procedure call (the top-of-stack of the prior stack frame). After data is pushed onto the procedure stack, the stack-frame base pointer remains where it was and the stack pointer advances downward to the address below the pushed data, where it becomes the new top-of-stack.

In legacy and compatibility modes, the default stack pointer size is 16 bits (SP) or 32 bits (ESP), depending on the default-size (B) bit in the stack-segment descriptor, and multiple stacks can be maintained in separate stack segments. In 64-bit mode, stack pointers are always 64 bits wide (RSP).

Further application-programming details on the procedure stack mechanism are described in “Control Transfers” on page 80. System-programming details on the stack segments are described in “Segmented Virtual Memory” in Volume 2.

A shadow stack is a separate, protected stack that is conceptually parallel to the procedure stack and used only by the shadow stack feature. When enabled by system software, the shadow stack feature provides, in a manner that is transparent to application software, protection against a class of computer exploit known as 'return oriented programming'. System-programming details on the shadow stack feature are described in “Shadow Stacks” in Volume 2.

2.5 Instruction Pointer

The instruction pointer is used in conjunction with the code-segment (CS) register to locate the next instruction in memory. The instruction-pointer register contains the displacement (offset)—from the base address of the current CS segment, or from address 0 in 64-bit mode—to the next instruction to be executed. The pointer is incremented sequentially, except for branch instructions, as described in “Control Transfers” on page 80.

In legacy and compatibility modes, the instruction pointer is a 16-bit (IP) or 32-bit (EIP) register. In 64-bit mode, the instruction pointer is extended to a 64-bit (RIP) register to support 64-bit offsets. The case-sensitive acronym, *rIP,* is used to refer to any of these three instruction-pointer sizes, depending on the software context.

Figure 2-10 on page 21 shows the relationship between RIP, EIP, and IP. The 64-bit RIP can be used for RIP-relative addressing, as described in "RIP-Relative Addressing" on page 18.

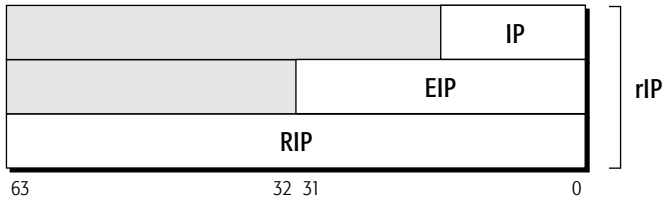


Figure 2-10. Instruction Pointer (rIP) Register

The contents of the rIP are not directly readable by software. However, the rIP is pushed onto the stack by a call instruction.

The memory model described in this chapter is used by all of the programming environments that make up the AMD64 architecture. The next four chapters of this volume describe the application programming environments, which include:

- General-purpose programming (Chapter 3 on page 23).
- Streaming SIMD extensions used in media and scientific programming (Chapter 4 on page 111).
- 64-bit media programming (Chapter 5 on page 239).
- x87 floating-point programming (Chapter 6 on page 285).

3 General-Purpose Programming

The general-purpose programming model includes the general-purpose registers (GPRs), integer instructions and operands that use the GPRs, program-flow control methods, memory optimization methods, and I/O. This programming model includes the original x86 integer-programming architecture, plus 64-bit extensions and a few additional instructions. Only the application-programming instructions and resources are described in this chapter. Integer instructions typically used in system programming, including all of the privileged instructions, are described in Volume 2, along with other system-programming topics.

The general-purpose programming model is used to some extent by almost all programs, including programs consisting primarily of 256-bit or 128-bit media instructions, 64-bit media instructions, x87 floating-point instructions, or system instructions. For this reason, an understanding of the general-purpose programming model is essential for any programming work using the AMD64 instruction set architecture.

3.1 Registers

Figure 3-1 on page 24 shows an overview of the registers used in general-purpose application programming. They include the general-purpose registers (GPRs), segment registers, flags register, and instruction-pointer register. The number and width of available registers depends on the operating mode.

The registers and register ranges shaded *light gray* in Figure 3-1 on page 24 are available only in 64-bit mode. Those shaded *dark gray* are available only in legacy mode and compatibility mode. Thus, in 64-bit mode, the 32-bit general-purpose, flags, and instruction-pointer registers available in legacy mode and compatibility mode are extended to 64-bit widths, eight new GPRs are available, and the DS, ES, and SS segment registers are ignored.

When naming registers, if reference is made to multiple register widths, a lower-case *r* notation is used. For example, the notation *rAX* refers to the 16-bit AX, 32-bit EAX, or 64-bit RAX register, depending on an instruction's effective operand size.

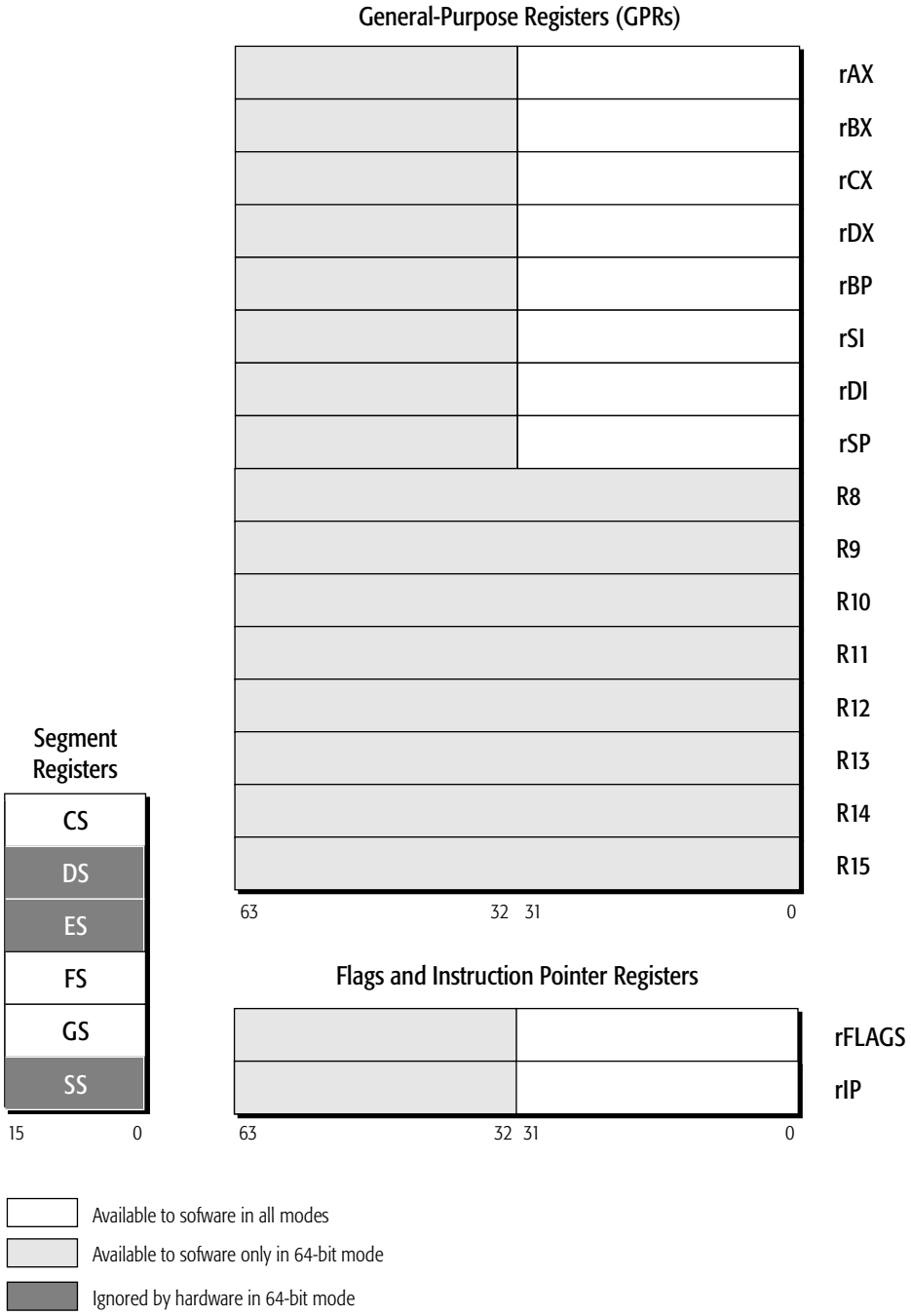


Figure 3-1. General-Purpose Programming Registers

3.1.1 Legacy Registers

In legacy and compatibility modes, all of the legacy x86 registers are available. Figure 3-2 on page 25 shows a detailed view of the GPR, flag, and instruction-pointer registers.

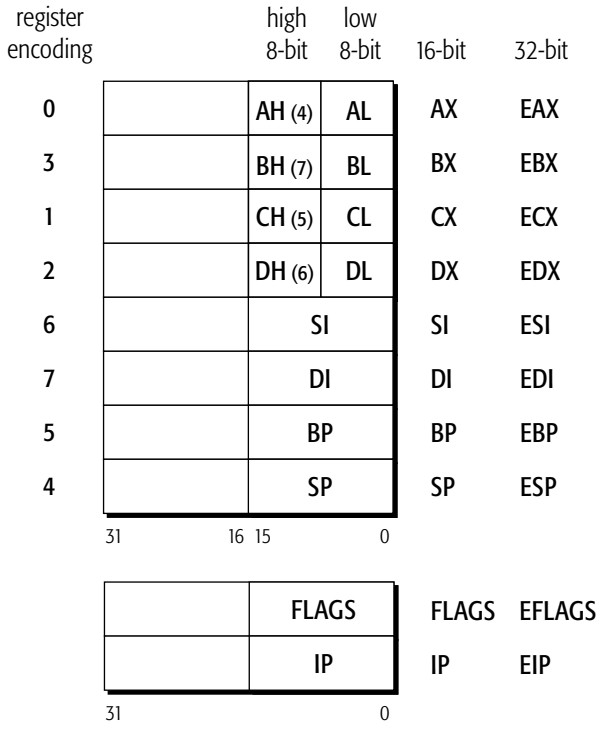


Figure 3-2. General Registers in Legacy and Compatibility Modes

The legacy GPRs include:

- Eight 8-bit registers (AH, AL, BH, BL, CH, CL, DH, DL).
- Eight 16-bit registers (AX, BX, CX, DX, DI, SI, BP, SP).
- Eight 32-bit registers (EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP).

The size of register used by an instruction depends on the effective operand size or, for certain instructions, the opcode, address size, or stack size. The 16-bit and 32-bit registers are encoded as 0 through 7 in Figure 3-2. For opcodes that specify a byte operand, registers encoded as 0 through 3 refer to the low-byte registers (AL, BL, CL, DL) and registers encoded as 4 through 7 refer to the high-byte registers (AH, BH, CH, DH).

The 16-bit FLAGS register, which is also the low 16 bits of the 32-bit EFLAGS register, shown in Figure 3-2, contains control and status bits accessible to application software, as described in Section 3.1.4, “Flags Register,” on page 34. The 16-bit IP or 32-bit EIP instruction-pointer register contains the address of the next instruction to be executed, as described in Section 2.5, “Instruction Pointer,” on page 20.

3.1.2 64-Bit-Mode Registers

In 64-bit mode, eight new GPRs are added to the eight legacy GPRs, all 16 GPRs are 64 bits wide, and the low bytes of all registers are accessible. Figure 3-3 on page 27 shows the GPRs, flags register, and instruction-pointer register available in 64-bit mode. The GPRs include:

- Sixteen 8-bit low-byte registers (AL, BL, CL, DL, SIL, DIL, BPL, SPL, R8B, R9B, R10B, R11B, R12B, R13B, R14B, R15B).
- Four 8-bit high-byte registers (AH, BH, CH, DH), addressable only when no REX prefix is used.
- Sixteen 16-bit registers (AX, BX, CX, DX, DI, SI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W).
- Sixteen 32-bit registers (EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D).
- Sixteen 64-bit registers (RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15).

The size of register used by an instruction depends on the effective operand size or, for certain instructions, the opcode, address size, or stack size. For most instructions, access to the extended GPRs requires a REX prefix (Section 3.5.2, “REX Prefixes,” on page 79). The four high-byte registers (AH, BH, CH, DH) available in legacy mode are not addressable when a REX prefix is used.

In general, byte and word operands are stored in the low 8 or 16 bits of GPRs without modifying their high 56 or 48 bits, respectively. Doubleword operands, however, are normally stored in the low 32 bits of GPRs and zero-extended to 64 bits.

The 64-bit RFLAGS register, shown in Figure 3-3 on page 27, contains the legacy EFLAGS in its low 32-bit range. The high 32 bits are reserved. They can be written with anything but they always read as zero (RAZ). The 64-bit RIP instruction-pointer register contains the address of the next instruction to be executed, as described in Section 3.1.5, “Instruction Pointer Register,” on page 36.

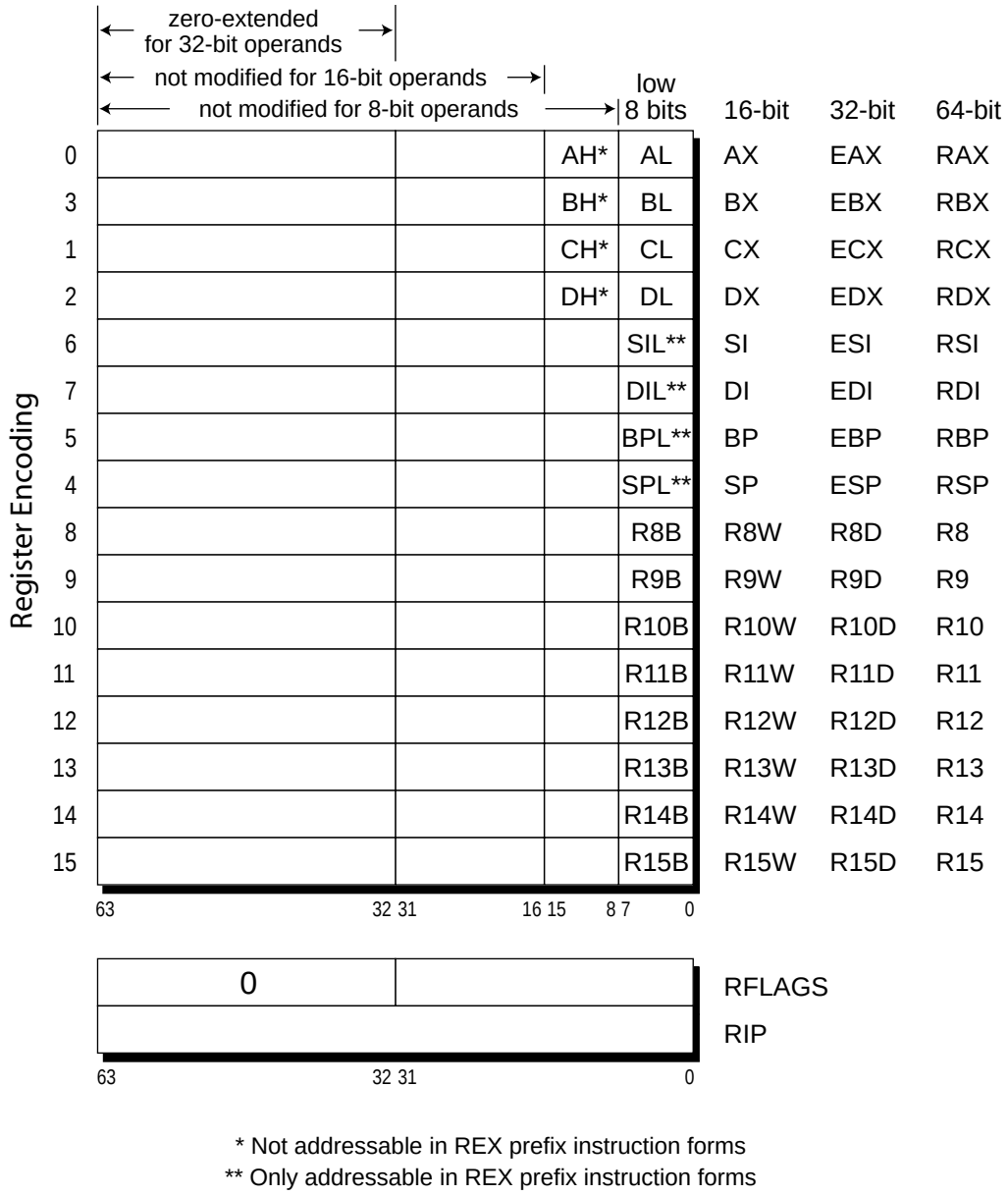


Figure 3-3. General Purpose Registers in 64-Bit Mode

Figure 3-4 on page 28 illustrates another way of viewing the 64-bit-mode GPRs, showing how the legacy GPRs overlay the extended GPRs. Gray-shaded bits are not modified in 64-bit mode.

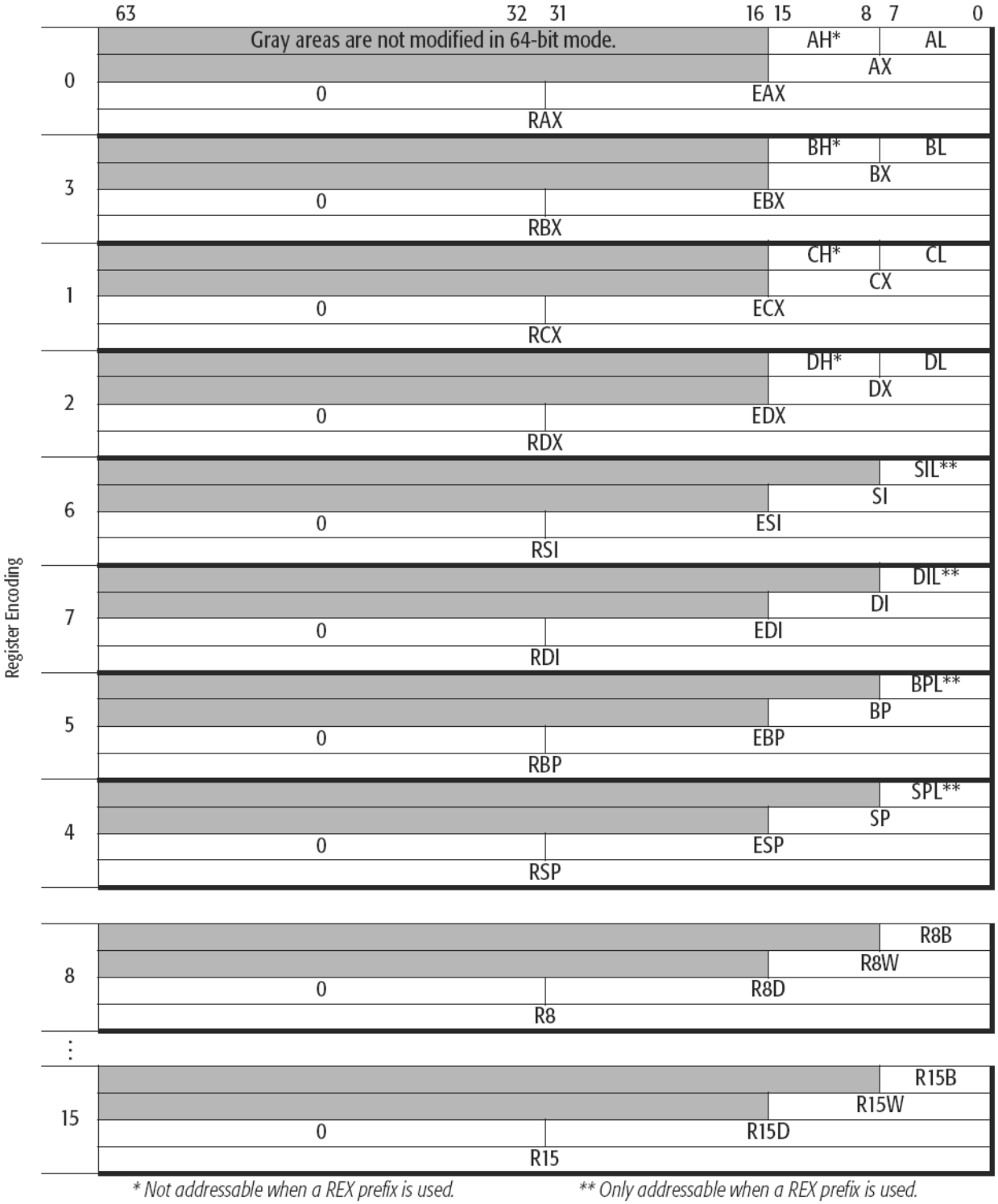


Figure 3-4. GPRs in 64-Bit Mode

3.1.2.1 Default Operand Size

For most instructions, the default operand size in 64-bit mode is 32 bits. To access 16-bit operand sizes, an instruction must contain an operand-size prefix (66h), as described in Section 3.2.3, "Operand Sizes and Overrides," on page 41. To access the full 64-bit operand size, most instructions must contain a REX prefix.

For details on operand size, see Section 3.2.3, "Operand Sizes and Overrides," on page 41.

3.1.2.2 Byte Registers

64-bit mode provides a uniform set of low-byte, low-word, low-doubleword, and quadword registers that is well-suited for register allocation by compilers. Access to the four new low-byte registers in the legacy-GPR range (SIL, DIL, BPL, SPL), or any of the low-byte registers in the extended registers (R8B–R15B), requires a REX instruction prefix. However, the legacy high-byte registers (AH, BH, CH, DH) are not accessible when a REX prefix is used.

3.1.2.3 Zero-Extension of 32-Bit Results

As Figure 3-3 on page 27 and Figure 3-4 on page 28 show, when performing 32-bit operations with a GPR destination in 64-bit mode, the processor zero-extends the 32-bit result into the full 64-bit destination. 8-bit and 16-bit operations on GPRs preserve all unwritten upper bits of the destination GPR. This is consistent with legacy 16-bit and 32-bit semantics for partial-width results.

Software should explicitly sign-extend the results of 8-bit, 16-bit, and 32-bit operations to the full 64-bit width before using the results in 64-bit address calculations.

The following four code examples show how 64-bit, 32-bit, 16-bit, and 8-bit ADDs work. In these examples, "48" is a REX prefix specifying 64-bit operand size, and "01C3" and "00C3" are the opcode and ModRM bytes of each instruction (see "Opcode Syntax" in Volume 3 for details on the opcode and ModRM encoding).

Example 1: 64-bit Add:

```
Before:RAX =0002_0001_8000_2201
       RBX =0002_0002_0123_3301

       48 01C3 ADD RBX,RAX ;48 is a REX prefix for size.

Result:RBX = 0004_0003_8123_5502
```

Example 2: 32-bit Add:

```
Before:RAX = 0002_0001_8000_2201
       RBX = 0002_0002_0123_3301

       01C3 ADD EBX,EAX ;32-bit add

 Result:RBX = 0000_0000_8123_5502
       (32-bit result is zero extended)
```

Example 3: 16-bit Add:

```
Before:RAX = 0002_0001_8000_2201
       RBX = 0002_0002_0123_3301

       66 01C3 ADD BX,AX ;66 is 16-bit size override

Result:RBX = 0002_0002_0123_5502
       (bits 63:16 are preserved)
```

Example 4: 8-bit Add:

```
Before:RAX = 0002_0001_8000_2201
       RBX = 0002_0002_0123_3301

       00C3 ADD BL,AL ;8-bit add

 Result:RBX = 0002_0002_0123_3302
       (bits 63:08 are preserved)
```

3.1.2.4 GPR High 32 Bits Across Mode Switches

The processor does not preserve the upper 32 bits of the 64-bit GPRs across switches from 64-bit mode to compatibility or legacy modes. When using 32-bit operands in compatibility or legacy mode, the high 32 bits of GPRs are undefined. Software must not rely on these undefined bits, because they can change from one implementation to the next or even on a cycle-to-cycle basis within a given implementation. The undefined bits are not a function of the data left by any previously running process.

3.1.3 Implicit Uses of GPRs

Most instructions can use any of the GPRs for operands. However, as Figure 3-1 on page 31 shows, some instructions use some GPRs implicitly. Details about implicit use of GPRs are described in "General-Purpose Instructions in 64-Bit Mode" in Volume 3.

Table 3-1 on page 31 shows implicit register uses only for application instructions. Certain system instructions also make implicit use of registers. These system instructions are described in "System Instruction Reference" in Volume 3.

Table 3-1. Implicit Uses of GPRs

Registers[1]				Name	Implicit Uses
Low 8-Bit	16-Bit	32-Bit	64-Bit		
AL	AX	EAX	RAX[2]	Accumulator	• Operand for decimal arithmetic, multiply, divide, string, compare-and-exchange, table-translation, and I/O instructions. • Special accumulator encoding for ADD, XOR, and MOV instructions. • Used with EDX to hold double-precision operands. • CPUID processor-feature information.
BL	BX	EBX	RBX[2]	Base	• Address generation in 16-bit code. • Memory address for XLAT instruction. • CPUID processor-feature information.
CL	CX	ECX	RCX[2]	Count	• Bit index for shift and rotate instructions. • Iteration count for loop and repeated string instructions. • Jump conditional if zero. • CPUID processor-feature information.
DL	DX	EDX	RDX[2]	I/O Address	• Operand for multiply and divide instructions. • Port number for I/O instructions. • Used with EAX to hold double-precision operands. • CPUID processor-feature information.
SIL[2]	SI	ESI	RSI[2]	Source Index	• Memory address of source operand for string instructions. • Memory index for 16-bit addresses.

Note:
1. *Gray-shaded registers have no implicit uses.*
2. *Accessible only in 64-bit mode.*

Table 3-1. Implicit Uses of GPRs (continued)

Registers[1]				Name	Implicit Uses
Low 8-Bit	16-Bit	32-Bit	64-Bit		
DIL[2]	DI	EDI	RDI[2]	Destination Index	• Memory address of destination operand for string instructions. • Memory index for 16-bit addresses.
BPL[2]	BP	EBP	RBP[2]	Base Pointer	• Memory address of stack-frame base pointer.
SPL[2]	SP	ESP	RSP[2]	Stack Pointer	• Memory address of last stack entry (top of stack).
R8B–R10B[2]	R8W–R10W[2]	R8D–R10D[2]	R8–R10[2]	None	No implicit uses
R11B[2]	R11W[2]	R11D[2]	R11[2]	None	• Holds the value of RFLAGS on SYSCALL/SYSRET.
R12B–R15B[2]	R12W–R15W[2]	R12D–R15D[2]	R12–R15[2]	None	No implicit uses

Note:
1. *Gray-shaded registers have no implicit uses.*
2. *Accessible only in 64-bit mode.*

3.1.3.1 Arithmetic Operations

Several forms of the add, subtract, multiply, and divide instructions use AL or rAX implicitly. The multiply and divide instructions also use the concatenation of rDX:rAX for double-sized results (multiplies) or quotient and remainder (divides).

3.1.3.2 Sign-Extensions

The instructions that double the size of operands by sign extension (for example, CBW, CWDE, CDQE, CWD, CDQ, CQO) use rAX register implicitly for the operand. The CWD, CDQ, and CQO instructions also uses the rDX register.

3.1.3.3 Special MOVs

The MOV instruction has several opcodes that implicitly use the AL or rAX register for one operand.

3.1.3.4 String Operations

Many types of string instructions use the accumulators implicitly. Load string, store string, and scan string instructions use AL or rAX for data and rDI or rSI for the offset of a memory address.

3.1.3.5 I/O-Address-Space Operations.

The I/O and string I/O instructions use rAX to hold data that is received from or sent to a device located in the I/O-address space. DX holds the device I/O-address (the port number).

3.1.3.6 Table Translations

The table translate instruction (XLATB) uses AL for an memory index and rBX for memory base address.

3.1.3.7 Compares and Exchanges

Compare and exchange instructions (CMPXCHG) use the AL or rAX register for one operand.

3.1.3.8 Decimal Arithmetic

The decimal arithmetic instructions (AAA, AAD, AAM, AAS, DAA, DAS) that adjust binary-coded decimal (BCD) operands implicitly use the AL and AH register for their operations.

3.1.3.9 Shifts and Rotates

Shift and rotate instructions can use the CL register to specify the number of bits an operand is to be shifted or rotated.

3.1.3.10 Conditional Jumps

Special conditional-jump instructions use the rCX register instead of flags. The JCXZ and JrCXZ instructions check the value of the rCX register and pass control to the target instruction when the value of rCX register reaches 0.

3.1.3.11 Repeated String Operations

With the exception of I/O string instructions, all string operations use rSI as the source-operand pointer and rDI as the destination-operand pointer. I/O string instructions use rDX to specify the input-port or output-port number. For repeated string operations (those preceded with a repeat-instruction prefix), the rSI and rDI registers are incremented or decremented as the string elements are moved from the source location to the destination. Repeat-string operations also use rCX to hold the string length, and decrement it as data is moved from one location to the other.

3.1.3.12 Stack Operations

Stack operations make implicit use of the rSP register, and in some cases, the rBP register. The rSP register is used to hold the top-of-stack pointer (or simply, stack pointer). rSP is decremented when items are pushed onto the stack, and incremented when they are popped off the stack. The ENTER and LEAVE instructions use rBP as a stack-frame base pointer. Here, rBP points to the last entry in a data structure that is passed from one block-structured procedure to another.

The use of rSP or rBP as a base register in an address calculation implies the use of SS (stack segment) as the default segment. Using any other GPR as a base register without a segment-override prefix implies the use of the DS data segment as the default segment.

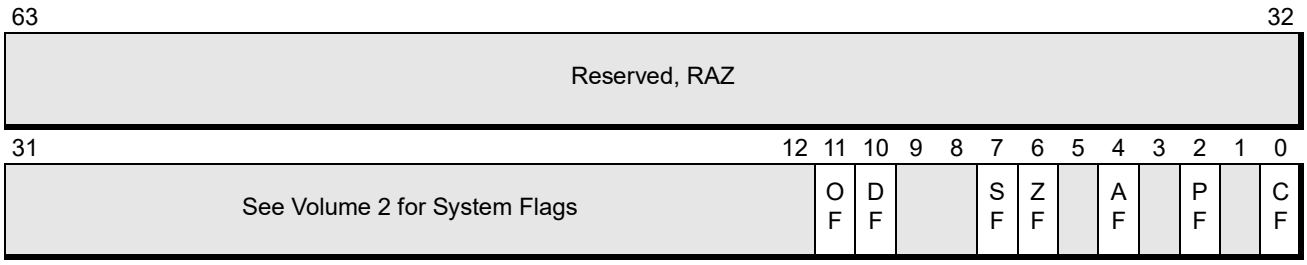
The push all and pop all instructions (PUSHA, PUSHAD, POPA, POPAD) implicitly use all of the GPRs.

3.1.3.13 CPUID Information

The CPUID instruction makes implicit use of the EAX, EBX, ECX, and EDX registers. Software loads a function code into EAX and, for some function codes, a sub-function code in ECX, executes the CPUID instruction, and then reads the associated processor-feature information in EAX, EBX, ECX, and EDX.

3.1.4 Flags Register

Figure 3-5 on page 34 shows the 64-bit RFLAGS register and the flag bits visible to application software. Bits 15:0 are the FLAGS register (accessed in legacy real and virtual-8086 modes), bits 31:0 are the EFLAGS register (accessed in legacy protected mode and compatibility mode), and bits 63:0 are the RFLAGS register (accessed in 64-bit mode). The name *rFLAGS* refers to any of the three register widths, depending on the current software context.



Bits	Mnemonic	Description	R/W
11	OF	Overflow Flag	R/W
10	DF	Direction Flag	R/W
7	SF	Sign Flag	R/W
6	ZF	Zero Flag	R/W
4	AF	Auxiliary Carry Flag	R/W
2	PF	Parity Flag	R/W
0	CF	Carry Flag	R/W

Figure 3-5. rFLAGS Register—Flags Visible to Application Software

The low 16 bits (FLAGS portion) of rFLAGS are accessible by application software and hold the following flags:

- One control flag (the direction flag DF).
- Six status flags (carry flag CF, parity flag PF, auxiliary carry flag AF, zero flag ZF, sign flag SF, and overflow flag OF).

The direction flag (DF) controls the direction of string operations. The status flags provide result information from logical and arithmetic operations and control information for conditional move and jump instructions.

Bits 31:16 of the rFLAGS register contain flags that are accessible only to system software. These flags are described in "System Registers" in Volume 2. The highest 32 bits of RFLAGS are reserved. In 64-bit mode, writes to these bits are ignored. They are read as zeros (RAZ). The rFLAGS register is initialized to 02h on reset, so that all of the programmable bits are cleared to zero.

The effects that rFLAGS bit-values have on instructions are summarized in the following places:

- Conditional Moves (CMOV*cc*)—Table 3-4 on page 46.
- Conditional Jumps (J*cc*)—Table 3-5 on page 60.
- Conditional Sets (SET*cc*)—Table 3-6 on page 64.

The effects that instructions have on rFLAGS bit-values are summarized in "Instruction Effects on RFLAGS" in Volume 3.

The sections below describe each application-visible flag. All of these flags are readable and writable. For example, the POPF, POPFD, POPFQ, IRET, IRETD, and IRETQ instructions write all flags. The carry and direction flags are writable by dedicated application instructions. Other application-visible flags are written indirectly by specific instructions. Reserved bits and bits whose writability is prevented by the current values of system flags, current privilege level (CPL), or the current operating mode, are unaffected by the POPF*x* instructions.

Carry Flag (CF). Bit 0. Hardware sets the carry flag to 1 if the last integer addition or subtraction operation resulted in a carry (for addition) or a borrow (for subtraction) out of the most-significant bit position of the result. Otherwise, hardware clears the flag to 0.

The increment and decrement instructions—unlike the addition and subtraction instructions—do not affect the carry flag. The bit shift and bit rotate instructions shift bits of operands into the carry flag. Logical instructions like AND, OR, XOR clear the carry flag. Bit-test instructions (BTx) set the value of the carry flag depending on the value of the tested bit of the operand.

Software can set or clear the carry flag with the STC and CLC instructions, respectively. Software can complement the flag with the CMC instruction.

Parity Flag (PF). Bit 2. Hardware sets the parity flag to 1 if there is an even number of 1 bits in the least-significant byte of the last result of certain operations. Otherwise (i.e., for an odd number of 1 bits), hardware clears the flag to 0. Software can read the flag to implement parity checking.

Auxiliary Carry Flag (AF). Bit 4. Hardware sets the auxiliary carry flag if an arithmetic operation or a binary-coded decimal (BCD) operation generates a carry (in the case of an addition) or a borrow (in the case of a subtraction) out of bit 3 of the result. Otherwise, AF is cleared to zero.

The main application of this flag is to support decimal arithmetic operations. Most commonly, this flag is used internally by correction commands for decimal addition (AAA) and subtraction (AAS).

Zero Flag (ZF). Bit 6. Hardware sets the zero flag to 1 if the last arithmetic operation resulted in a value of zero. Otherwise (for a non-zero result), hardware clears the flag to 0. The compare and test instructions also affect the zero flag.

The zero flag is typically used to test whether the result of an arithmetic or logical operation is zero, or to test whether two operands are equal.

Sign Flag (SF). Bit 7. Hardware sets the sign flag to 1 if the last arithmetic operation resulted in a negative value. Otherwise (for a positive-valued result), hardware clears the flag to 0. Thus, in such operations, the value of the sign flag is set equal to the value of the most-significant bit of the result. Depending on the size of operands, the most-significant bit is bit 7 (for bytes), bit 15 (for words), bit 31 (for doublewords), or bit 63 (for quadwords).

Direction Flag (DF). Bit 10. The direction flag determines the order in which strings are processed. Software can set the direction flag to 1 to specify decrementing the data pointer for the next string instruction (LODSx, STOSx, MOVSx, SCASx, CMPSx, OUTSx, or INSx). Clearing the direction flag to 0 specifies incrementing the data pointer. The pointers are stored in the rSI or rDI register. Software can set or clear the flag with the STD and CLD instructions, respectively.

Overflow Flag (OF). Bit 11. Hardware sets the overflow flag to 1 to indicate that the most-significant (sign) bit of the result of the last signed integer operation differed from the signs of both source operands. Otherwise, hardware clears the flag to 0. A set overflow flag means that the magnitude of the positive or negative result is too big (overflow) or too small (underflow) to fit its defined data type.

The OF flag is undefined after the DIV instruction and after a shift of more than one bit. Logical instructions clear the overflow flag.

3.1.5 Instruction Pointer Register

The instruction pointer register—IP, EIP, or RIP, or simply rIP for any of the three depending on the context—is used in conjunction with the code-segment (CS) register to locate the next instruction in memory. See Section 2.5, “Instruction Pointer,” on page 20 for details.

3.2 Operands

Operands are either referenced by an instruction's encoding or included as an immediate value in the instruction encoding. Depending on the instruction, referenced operands can be located in registers, memory locations, or I/O ports.

3.2.1 Fundamental Data Types

At the most fundamental level, a datum is an ordered string of a specific length composed of binary digits (bits). Bits are indexed from *0* to *length-1*. While technically the size of a datum is not restricted, for convenience in storing and manipulating data the Architecture defines a finite number of data objects of specific size and names them.

A datum of length 1 is simply a *bit*. A datum of length 4 is a *nibble*, a datum of length 8 is a *byte*, a datum of length 16 is a *word*, a datum of length 32 is a *doubleword*, a datum of length 64 is a *quadword*, a datum of length 128 is a *double quadword* (also called an *octword*), a datum of length 256 is a *double octword*.

For instructions that move or reorder data, the significance of each bit within the datum is immaterial. An instruction of this type may operate on bits, bytes, words, doublewords, and so on. The majority of instructions, however, expect operand data to be of a specific format. The format assigns a particular significance to each bit based on its position within the datum. This assignment of significance or meaning to each bit is called data typing.

The Architecture defines the following fundamental data types:

- Untyped data objects
  - bit
  - nibble (4 bits)
  - byte (8 bits)
  - word (16 bits)
  - doubleword (32 bits)
  - quadword (64 bits)
  - double quadword (octword) (128 bits)
  - double octword (256 bits)
- Unsigned integers
  - 8-bit (byte) unsigned integer
  - 16-bit (word) unsigned integer
  - 32-bit (doubleword) unsigned integer
  - 64-bit (quadword) unsigned integer
  - 128-bit (octword) unsigned integer
- Signed (two's-complement) integers
  - 8-bit (byte) signed integer
  - 16-bit (word) signed integer
  - 32-bit (doubleword) signed integer
  - 64-bit (quadword) signed integer
  - 128-bit (octword) signed integer
- Binary coded decimal (BCD) digits
- Floating-point data types
  - half-precision floating point (16 bits)
  - single-precision floating point (32 bits)
  - double-precision floating point (64 bits)

These fundamental data types may be aggregated into composite data types. The defined composite data types are:

- strings

  - character strings (composed of bytes or words)
  - doubleword and quadword
- packed BCD
- packed signed and unsigned integers (also called integer vectors)
- packed single- or double-precision floating point (also called floating-point vectors)

Integer, BCD, and string data types are described in the following section. The floating-point and vector data types are discussed in Section 4.3.3, “SSE Instruction Data Types,” on page 121.

3.2.2 General-Purpose Instruction Data types

The following data types are supported in the general-purpose programming environment:

- Signed (two's-complement) integers.
- Unsigned integers.
- BCD digits.
- Packed BCD digits.
- Strings, including bit strings.
- Untyped data objects.

Figure 3-6 on page 39 illustrates the data types used by most general-purpose instructions. Software can define data types in ways other than those shown, but the AMD64 architecture does not directly support such interpretations and software must handle them entirely on its own. Note that the bit positions are numbered from right to left starting with *0* and ending with *length-1*. The untyped data objects bit, nibble, byte, word, doubleword, quadword, and octword are not shown.

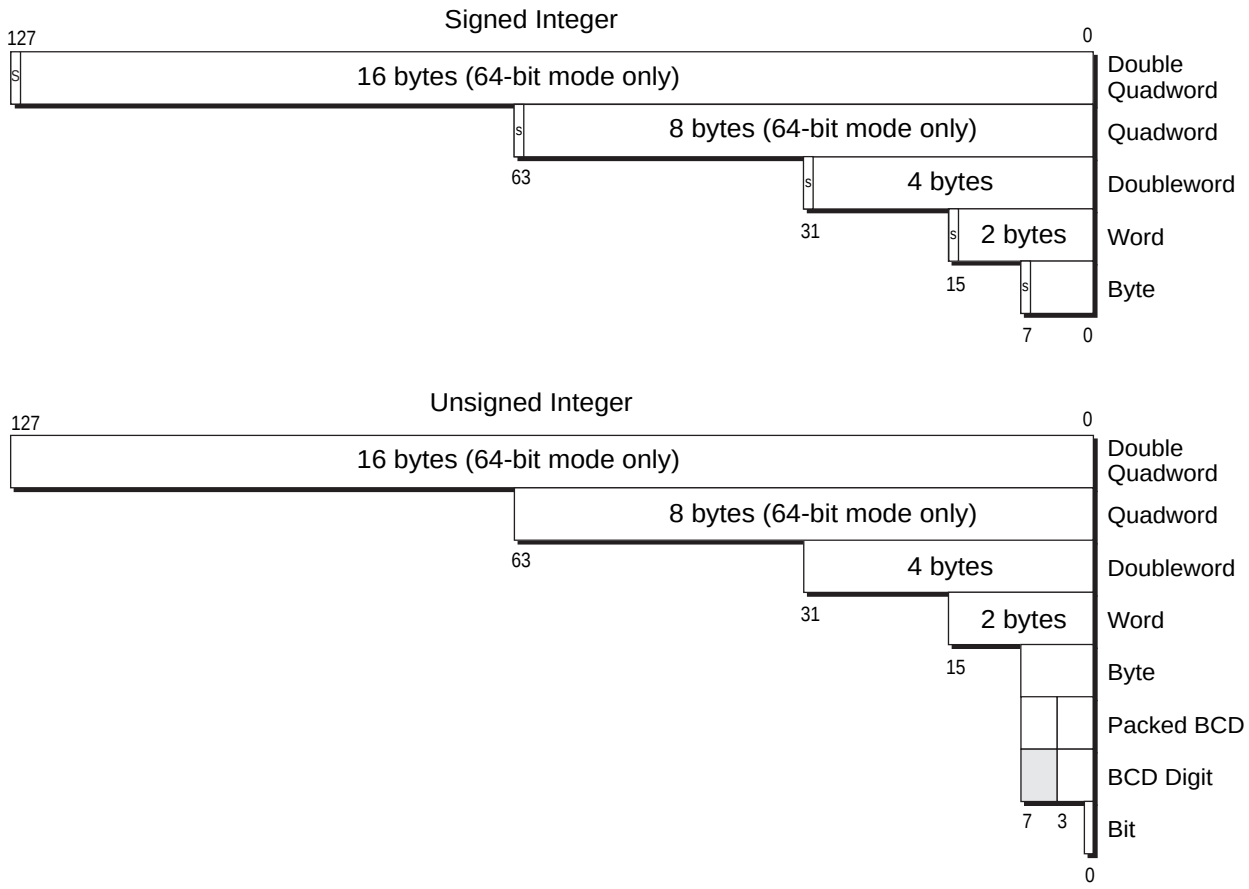


Figure 3-6. General-Purpose Data Types

3.2.2.1 Signed and Unsigned Integers

The architecture supports signed and unsigned 1-byte, 2-byte, 4- byte, 8-byte, and 16-byte integers. The sign bit (S) occupies the most significant bit (datum bit position *length-1*). Signed integers are represented in two's complement format. S = 0 represents positive numbers and S = 1 negative numbers.

The table below presents the representable range of values for each integer data type and the BCD data types discussed in the following section:

Table 3-2. Representable Values of General-Purpose Data Types

Data Type	Byte	Word	Doubleword	Quadword	Double Quadword[2]
Signed Integers[1]	-2^7 to $+(2^7-1)$	-2^{15} to $+(2^{15}-1)$	-2^{31} to $+(2^{31}-1)$	-2^{63} to $+(2^{63}-1)$	-2^{127} to $+(2^{127}-1)$
Unsigned Integers	0 to $+2^8-1$ (0 to 255)	0 to $+2^{16}-1$ (0 to 65,535)	0 to $+2^{32}-1$ (0 to 4.29×10^9)	0 to $+2^{64}-1$ (0 to 1.84×10^{19})	0 to $+2^{128}-1$ (0 to 3.40×10^{38})
Packed BCD Digits	00 to 99	multiple packed BCD-digit bytes			
BCD Digit	0 to 9	multiple BCD-digit bytes			

Note:
1. *The sign bit is the most-significant bit (e.g., bit 7 for a byte, bit 15 for a word, etc.).*
2. *The double quadword data type is supported in the RDX:RAX registers by the MUL, IMUL, DIV, IDIV, and CQO instructions.*

In 64-bit mode, the double quadword (octword) integer data type is supported in the RDX:RAX registers by the MUL, IMUL, DIV, IDIV, and CQO instructions.

3.2.2.2 Binary-Coded-Decimal (BCD) Digits

BCD digits have values ranging from 0 to 9. These values can be represented in binary encoding with four bits. For example, 0000b represents the decimal number 0 and 1001b represents the decimal number 9. Values ranging from 1010b to 1111b are invalid for this data type. Because a byte contains eight bits, two BCD digits can be stored in a single byte. This is referred to as *packed-BCD*. If a single BCD digit is stored per byte, it is referred to as *unpacked-BCD*. In the x87 floating-point programming environment (described in Section 6, "x87 Floating-Point Programming," on page 285) an 80-bit packed BCD data type is also supported, along with conversions between floating-point and BCD data types, so that data expressed in the BCD format can be operated on as floating-point values.

Integer add, subtract, multiply, and divide instructions can be used to operate on single (unpacked) BCD digits. The result must be adjusted to produce a correct BCD representation. For unpacked BCD numbers, the ASCII-adjust instructions are provided to simplify that correction. In the case of division, the adjustment must be made prior to executing the integer-divide instruction.

Similarly, integer add and subtract instructions can be used to operate on packed-BCD digits. The result must be adjusted to produce a correct packed-BCD representation. Decimal-adjust instructions are provided to simplify packed-BCD result corrections.

3.2.2.3 Strings

Strings are a continuous sequence of a single data type. The string instructions can be used to operate on byte, word, doubleword, or quadword data types. The maximum length of a string of any data type is $2^{32}-1$ bytes, in legacy or compatibility modes, or $2^{64}-1$ bytes in 64-bit mode. One of the more common types of strings used by applications are byte data-type strings known as ASCII strings, which can be used to represent character data.

Bit strings are also supported by instructions that operate specifically on bit strings. In general, bit strings can start and end at any bit location within any byte, although the BT*x* bit-string instructions assume that strings start on a byte boundary. The length of a bit string can range in size from a single bit up to 2^{32}–1 bits, in legacy or compatibility modes, or 2^{64}–1 bits in 64-bit mode.

3.2.2.4 Untyped Data Objects

Move instructions: register to register, memory to register (load) or register to memory (store); pack, unpack, swap, permutate, and merge instructions operate on data without regard to data type.

SIMD instructions operate on vector data types based on the fundamental data types described above. See Section 4.3. “Operands” on page 118 for a discussion of vector data types

3.2.3 Operand Sizes and Overrides

3.2.3.1 Default Operand Size

In legacy and compatibility modes, the default operand size is either 16 bits or 32 bits, as determined by the default-size (D) bit in the current code-segment descriptor (for details, see “Segmented Virtual Memory” in Volume 2). In 64-bit mode, the default operand size for most instructions is 32 bits.

Application software can override the default operand size by using an operand-size instruction prefix. Table 3-3 shows the instruction prefixes for operand-size overrides in all operating modes. In 64-bit mode, the default operand size for most instructions is 32 bits. A REX prefix (see Section 3.5.2, “REX Prefixes,” on page 79) specifies a 64-bit operand size, and a 66h prefix specifies a 16-bit operand size. The REX prefix takes precedence over the 66h prefix.

Table 3-3. Operand-Size Overrides

Operating Mode		Default Operand Size (Bits)	Effective Operand Size (Bits)	Instruction Prefix	
				66h[1]	REX
Long Mode	64-Bit Mode	32[2]	64	x	yes
			32	no	no
			16	yes	no
	Compatibility Mode	32	32	no	Not Applicable
			16	yes	
		16	32	yes	
			16	no	
Legacy Mode (Protected, Virtual-8086, or Real Mode)		32	32	no	
			16	yes	
		16	32	yes	
			16	no	

Note:
1. *A "no" indicates that the default operand size is used. An "x" means "don't care."*
2. *Near branches, instructions that implicitly reference the stack pointer, and certain other instructions default to 64-bit operand size. See "General-Purpose Instructions in 64-Bit Mode" in Volume 3*

There are several exceptions to the 32-bit operand-size default in 64-bit mode, including near branches and instructions that implicitly reference the RSP stack pointer. For example, the near CALL, near JMP, J*cc*, LOOP*cc*, POP, and PUSH instructions all default to a 64-bit operand size in 64-bit mode. Such instructions do not need a REX prefix for the 64-bit operand size. For details, see "General-Purpose Instructions in 64-Bit Mode" in Volume 3.

3.2.3.2 Effective Operand Size

The term *effective operand size* describes the operand size for the current instruction, after accounting for the instruction's default operand size and any operand-size override or REX prefix that is used with the instruction.

3.2.3.3 Immediate Operand Size

In legacy mode and compatibility modes, the size of immediate operands can be 8, 16, or 32 bits, depending on the instruction. In 64-bit mode, the maximum size of an immediate operand is also 32 bits, except that 64-bit immediates can be copied into a 64-bit GPR using the MOV instruction.

When the operand size of a MOV instruction is 64 bits, the processor sign-extends immediates to 64 bits before using them. Support for true 64-bit immediates is accomplished by expanding the semantics of the `MOV reg, imm16/32` instructions. In legacy and compatibility modes, these instructions—opcodes B8h through BFh—copy a 16-bit or 32-bit immediate (depending on the

effective operand size) into a GPR. In 64-bit mode, if the operand size is 64 bits (requires a REX prefix), these instructions can be used to copy a true 64-bit immediate into a GPR.

3.2.4 Operand Addressing

Operands for general-purpose instructions are referenced by the instruction's syntax or they are incorporated in the instruction as an immediate value. Referenced operands can be in registers, memory, or I/O ports.

3.2.4.1 Register Operands

Most general-purpose instructions that take register operands reference the general-purpose registers (GPRs). A few general-purpose instructions reference operands in the RFLAGS register, XMM registers, or MMX™ registers.

The type of register addressed is specified in the instruction syntax. When addressing GPRs or XMM registers, the REX instruction prefix can be used to access the extended GPRs or XMM registers, as described in Section 3.5, “Instruction Prefixes,” on page 76.

3.2.4.2 Memory Operands

Many general-purpose instructions can access operands in memory. Section 2.2, “Memory Addressing,” on page 14 describes the general methods and conditions for addressing memory operands.

3.2.4.3 I/O Ports

Operands in I/O ports are referenced according to the conventions described in Section 3.8, “Input/Output,” on page 95.

3.2.4.4 Immediate Operands

In certain instructions, a source operand—called an *immediate operand*, or simply *immediate*—is included as part of the instruction rather than being accessed from a register or memory location. For details on the size of immediate operands, see “Immediate Operand Size” on page 42.

3.2.5 Data Alignment

A data access is *aligned* if its address is a multiple of its operand size, in bytes. The following examples illustrate this definition:

- *Byte* accesses are always aligned. Bytes are the smallest addressable parts of memory.
- *Word* (two-byte) accesses are aligned if their address is a multiple of 2.
- *Doubleword* (four-byte) accesses are aligned if their address is a multiple of 4.
- *Quadword* (eight-byte) accesses are aligned if their address is a multiple of 8.

The AMD64 architecture does not impose data-alignment requirements for accessing data in memory. However, depending on the location of the misaligned operand with respect to the width of the data

bus and other aspects of the hardware implementation (such as store-to-load forwarding mechanisms), a misaligned memory access can require more bus cycles than an aligned access. For maximum performance, avoid misaligned memory accesses.

Performance on many hardware implementations will benefit from observing the following operand-alignment and operand-size conventions:

- Avoid misaligned data accesses.
- Maintain consistent use of operand size across all loads and stores. Larger operand sizes (doubleword and quadword) tend to make more efficient use of the data bus and any data-forwarding features that are implemented by the hardware.
- When using word or byte stores, avoid loading data from the same doubleword of memory, other than the identical start addresses of the stores.

3.3 Instruction Summary

This section summarizes the functions of the general-purpose instructions. The instructions are organized by functional group—such as, data-transfer instructions, arithmetic instructions, and so on. Details on individual instructions are given in the alphabetically organized "General-Purpose Instructions in 64-Bit Mode" in Volume 3.

3.3.1 Syntax

Each instruction has a *mnemonic syntax* used by assemblers to specify the operation and the operands to be used for source and destination (result) data. Figure 3-7 shows an example of the mnemonic syntax for a compare (CMP) instruction. In this example, the CMP mnemonic is followed by two operands, a 32-bit register or memory operand and an 8-bit immediate operand.

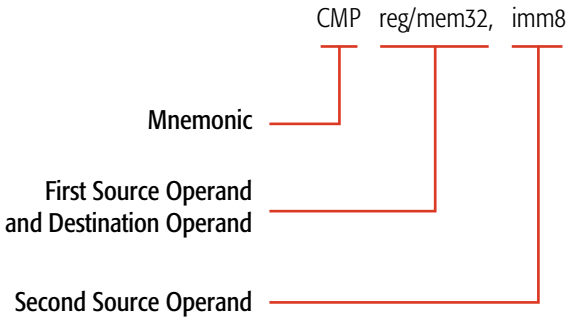


Figure 3-7. Mnemonic Syntax Example

In most instructions that take two operands, the first (left-most) operand is both a source operand and the destination operand. The second (right-most) operand serves only as a source. Instructions can have one or more prefixes that modify default instruction functions or operand properties. These

prefixes are summarized in Section 3.5, “Instruction Prefixes,” on page 76. Instructions that access 64-bit operands in a general-purpose register (GPR) or any of the extended GPR or XMM registers require a REX instruction prefix.

Unless otherwise stated in this section, the word *register* means a general-purpose register (GPR). Several instructions affect the flag bits in the RFLAGS register. “Instruction Effects on RFLAGS” in Volume 3 summarizes the effects that instructions have on rFLAGS bits.

3.3.2 Data Transfer

The data-transfer instructions copy data between registers and memory.

Move

- MOV—Move
- MOVBE—Move Big-Endian
- MOVSX—Move with Sign-Extend
- MOVZX—Move with Zero-Extend
- MOVD—Move Doubleword or Quadword
- MOVNTI—Move Non-temporal Doubleword or Quadword

The move instructions copy a byte, word, doubleword, or quadword from a register or memory location to a register or memory location. The source and destination cannot both be memory locations. For MOVBE, both operands cannot be registers and the operand size must be greater than one byte. MOVBE performs a reordering of the bytes within the source operand as it is copied.

An immediate constant can be used as a source operand with the MOV instruction. For most move instructions, the destination must be of the same size as the source, but the MOVSX and MOVZX instructions copy values of smaller size to a larger size by using sign-extension or zero-extension respectively. The MOVD instruction copies a doubleword or quadword between a general-purpose register or memory and an XMM or MMX register.

The MOV instruction is in many aspects similar to the assignment operator in high-level languages. The simplest example of their use is to initialize variables. To initialize a register to 0, rather than using a MOV instruction it may be more efficient to use the XOR instruction with identical destination and source operands.

The MOVNTI instruction stores a doubleword or quadword from a register into memory as “non-temporal” data, which assumes a single access (as opposed to frequent subsequent accesses of “temporal data”). The operation therefore minimizes cache pollution. The exact method by which cache pollution is minimized depends on the hardware implementation of the instruction. For further information, see Section 3.9, “Memory Optimization,” on page 98.

Conditional Move

- CMOV*cc*—Conditional Move If *condition*

The CMOV*cc* instructions conditionally copy a word, doubleword, or quadword from a register or memory location to a register location. The source and destination must be of the same size.

The CMOV*cc* instructions perform the same task as MOV but work conditionally, depending on the state of status flags in the RFLAGS register. If the condition is not satisfied, the instruction has no effect and control is passed to the next instruction. The mnemonics of CMOV*cc* instructions indicate the condition that must be satisfied. Several mnemonics are often used for one opcode to make the mnemonics easier to remember. For example, CMOVE (conditional move if equal) and CMOVZ (conditional move if zero) are aliases and compile to the same opcode. Table 3-4 shows the RFLAGS values required for each CMOV*cc* instruction.

In assembly languages, the conditional move instructions correspond to small conditional statements like:

```
IF a = b THEN x = y
```

CMOV*cc* instructions can replace two instructions—a conditional jump and a move. For example, to perform a high-level statement like:

```
IF ECX = 5 THEN EAX = EBX
```

without a CMOV*cc* instruction, the code would look like:

```
cmp ecx, 5          ; test if ecx equals 5
jnz Continue        ; test condition and skip if not met
mov eax, ebx        ; move
Continue:           ; continuation
```

but with a CMOV*cc* instruction, the code would look like:

```
cmp ecx, 5          ; test if ecx equals to 5
cmovz eax, ebx      ; test condition and move
```

Replacing conditional jumps with conditional moves also has the advantage that it can avoid branch-prediction penalties that may be caused by conditional jumps.

Support for CMOV*cc* instructions depends on the processor implementation. To find out if a processor is able to perform CMOV*cc* instructions, use the CPUID instruction. For more information on using the CPUID instruction, see Section 3.6, “Feature Detection,” on page 79.

Table 3-4. rFLAGS for CMOV*cc* Instructions

Mnemonic	Required Flag State	Description
CMOVO	OF = 1	Conditional move if overflow
CMOVNO	OF = 0	Conditional move if not overflow
CMOVB CMOVC CMOVNAE	CF = 1	Conditional move if below Conditional move if carry Conditional move if not above or equal

Table 3-4. rFLAGS for CMOV*cc* Instructions (continued)

Mnemonic	Required Flag State	Description
CMOVAE CMOVNB CMOVNC	CF = 0	Conditional move if above or equal Conditional move if not below Conditional move if not carry
CMOVE CMOVZ	ZF = 1	Conditional move if equal Conditional move if zero
CMOVNE CMOVNZ	ZF = 0	Conditional move if not equal Conditional move if not zero
CMOVBE CMOVNA	CF = 1 or ZF = 1	Conditional move if below or equal Conditional move if not above
CMOVA CMOVNBE	CF = 0 and ZF = 0	Conditional move if not below or equal Conditional move if not below or equal
CMOVS	SF = 1	Conditional move if sign
CMOVNS	SF = 0	Conditional move if not sign
CMOVP CMOVPE	PF = 1	Conditional move if parity Conditional move if parity even
CMOVNP CMOVPO	PF = 0	Conditional move if not parity Conditional move if parity odd
CMOVL CMOVNGE	SF <> OF	Conditional move if less Conditional move if not greater or equal
CMOVGE CMOVNL	SF = OF	Conditional move if greater or equal Conditional move if not less
CMOVLE CMOVNG	ZF = 1 or SF <> OF	Conditional move if less or equal Conditional move if not greater
CMOVG CMOVNLE	ZF = 0 and SF = OF	Conditional move if greater Conditional move if not less or equal

Stack Operations

- POP—Pop Stack
- POPA—Pop All to GPR Words
- POPAD—Pop All to GPR Doublewords
- PUSH—Push onto Stack
- PUSHA—Push All GPR Words onto Stack
- PUSHAD—Push All GPR Doublewords onto Stack
- ENTER—Create Procedure Stack Frame
- LEAVE—Delete Procedure Stack Frame

PUSH copies the specified register, memory location, or immediate value to the top of stack. This instruction decrements the stack pointer by 2, 4, or 8, depending on the operand size, and then copies the operand into the memory location pointed to by SS:rSP.

POP copies a word, doubleword, or quadword from the memory location pointed to by the SS:rSP registers (the top of stack) to a specified register or memory location. Then, the rSP register is incremented by 2, 4, or 8. After the POP operation, rSP points to the new top of stack.

PUSHA or PUSHAD stores eight word-sized or doubleword-sized registers onto the stack: eAX, eCX, eDX, eBX, eSP, eBP, eSI and eDI, in that order. The stored value of eSP is sampled at the moment when the PUSHA instruction started. The resulting stack-pointer value is decremented by 16 or 32.

POPA or POPAD extracts eight word-sized or doubleword-sized registers from the stack: eDI, eSI, eBP, eSP, eBX, eDX, eCX and eAX, in that order (which is the reverse of the order used in the PUSHA instruction). The stored eSP value is ignored by the POPA instruction. The resulting stack pointer value is incremented by 16 or 32.

It is a common practice to use PUSH instructions to pass parameters (via the stack) to functions and subroutines. The typical instruction sequence used at the beginning of a subroutine looks like:

```
push  ebp          ; save current EBP
mov   ebp, esp     ; set stack frame pointer value
sub   esp, N       ; allocate space for local variables
```

The rBP register is used as a *stack frame pointer*—a base address of the stack area used for parameters passed to subroutines and local variables. Positive offsets of the stack frame pointed to by rBP provide access to parameters passed while negative offsets give access to local variables. This technique allows creating re-entrant subroutines.

The ENTER and LEAVE instructions provide support for procedure calls, and are mainly used in high-level languages. The ENTER instruction is typically the first instruction of the procedure, and the LEAVE instruction is the last before the RET instruction.

The ENTER instruction creates a stack frame for a procedure. The first operand, *size*, specifies the number of bytes allocated in the stack. The second operand, *depth*, specifies the number of stack-frame pointers copied from the calling procedure's stack (i.e., the nesting level). The depth should be an integer in the range 0–31.

Typically, when a procedure is called, the stack contains the following four components:

- Parameters passed to the called procedure (created by the calling procedure).
- Return address (created by the CALL instruction).
- Array of stack-frame pointers (pointers to stack frames of procedures with smaller nesting-level depth) which are used to access the local variables of such procedures.
- Local variables used by the called procedure.

All these data are called the *stack frame*. The ENTER instruction simplifies management of the last two components of a stack frame. First, the current value of the rBP register is pushed onto the stack. The value of the rSP register at that moment is a *frame pointer* for the current procedure: positive offsets from this pointer give access to the parameters passed to the procedure, and negative offsets give access to the local variables which will be allocated later. During procedure execution, the value of the frame pointer is stored in the rBP register, which at that moment contains a frame pointer of the

calling procedure. This frame pointer is saved in a temporary register. If the depth operand is greater than one, the array of *depth-1* frame pointers of procedures with smaller nesting level is pushed onto the stack. This array is copied from the stack frame of the calling procedure, and it is addressed by the rBP register from the calling procedure. If the depth operand is greater than zero, the saved frame pointer of the current procedure is pushed onto the stack (forming an array of *depth* frame pointers). Finally, the saved value of the frame pointer is copied to the rBP register, and the rSP register is decremented by the value of the first operand, allocating space for local variables used in the procedure. See “Stack Operations” on page 47 for a parameter-passing instruction sequence using PUSH that is equivalent to ENTER.

The LEAVE instruction removes local variables and the array of frame pointers, allocated by the previous ENTER instruction, from the stack frame. This is accomplished by the following two steps: first, the value of the frame pointer is copied from the rBP register to the rSP register. This releases the space allocated by local variables and an array of frame pointers of procedures with smaller nesting levels. Second, the rBP register is popped from the stack, restoring the previous value of the frame pointer (or simply the value of the rBP register, if the depth operand is zero). Thus, the LEAVE instruction is equivalent to the following code:

```
mov rSP, rBP
pop rBP
```

3.3.3 Data Conversion

The data-conversion instructions perform various transformations of data, such as operand-size doubling by sign extension, conversion of little-endian to big-endian format, extraction of sign masks, searching a table, and support for operations with decimal numbers.

Sign Extension

- CBW—Convert Byte to Word
- CWDE—Convert Word to Doubleword
- CDQE—Convert Doubleword to Quadword
- CWD—Convert Word to Doubleword
- CDQ—Convert Doubleword to Quadword
- CQO—Convert Quadword to Octword

The CBW, CWDE, and CDQE instructions sign-extend the AL, AX, or EAX register to the upper half of the AX, EAX, or RAX register, respectively. By doing so, these instructions create a double-sized destination operand in rAX that has the same numerical value as the source operand. The CBW, CWDE, and CDQE instructions have the same opcode, and the action taken depends on the effective operand size.

The CWD, CDQ and CQO instructions sign-extend the AX, EAX, or RAX register to all bit positions of the DX, EDX, or RDX register, respectively. By doing so, these instructions create a double-sized destination operand in rDX:rAX that has the same numerical value as the source operand. The CWD,

CDQ, and CQO instructions have the same opcode, and the action taken depends on the effective operand size.

Flags are not affected by these instructions. The instructions can be used to prepare an operand for signed division (performed by the IDIV instruction) by doubling its storage size.

Extract Sign Mask

- (V)MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask
- (V)MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask

The MOVMSKPS instruction moves the sign bits of four packed single-precision floating-point values in an XMM register to the four low-order bits of a general-purpose register, with zero-extension. MOVMSKPD does a similar operation for two packed double-precision floating-point values: it moves the two sign bits to the two low-order bits of a general-purpose register, with zero-extension. The result of either instruction is a sign-bit mask.

Translate

- XLAT—Translate Table Index

The XLAT instruction replaces the value stored in the AL register with a table element. The initial value in AL serves as an unsigned index into the table, and the start (base) of table is specified by the DS:rBX registers (depending on the effective address size).

This instruction is not recommended. The following instruction serves to replace it:

```
MOV AL,[rBX + AL]
```

ASCII Adjust

- AAA—ASCII Adjust After Addition
- AAD—ASCII Adjust Before Division
- AAM—ASCII Adjust After Multiply
- AAS—ASCII Adjust After Subtraction

The AAA, AAD, AAM, and AAS instructions perform corrections of arithmetic operations with non-packed BCD values (i.e., when the decimal digit is stored in a byte register). There are no instructions which directly operate on decimal numbers (either packed or non-packed BCD). However, the ASCII-adjust instructions correct decimal-arithmetic results. These instructions assume that an arithmetic instruction, such as ADD, was performed on two BCD operands, and that the result was stored in the AL or AX register. This result can be incorrect or it can be a non-BCD value (for example, when a decimal carry occurs). After executing the proper ASCII-adjust instruction, the AX register contains a correct BCD representation of the result. (The AAD instruction is an exception to this, because it should be applied *before* a DIV instruction, as explained below). All of the ASCII-adjust instructions are able to operate with multiple-precision decimal values.

AAA should be applied after addition of two non-packed decimal digits. AAS should be applied after subtraction of two non-packed decimal digits. AAM should be applied after multiplication of two non-

packed decimal digits. AAD should be applied *before* the division of two non-packed decimal numbers.

Although the base of the numeration for ASCII-adjust instructions is assumed to be 10, the AAM and AAD instructions can be used to correct multiplication and division with other bases.

BCD Adjust

- DAA—Decimal Adjust after Addition
- DAS—Decimal Adjust after Subtraction

The DAA and DAS instructions perform corrections of addition and subtraction operations on packed BCD values. (Packed BCD values have two decimal digits stored in a byte register, with the higher digit in the higher four bits, and the lower one in the lower four bits.) There are no instructions for correction of multiplication and division with packed BCD values.

DAA should be applied after addition of two packed-BCD numbers. DAS should be applied after subtraction of two packed-BCD numbers.

DAA and DAS can be used in a loop to perform addition or subtraction of two multiple-precision decimal numbers stored in packed-BCD format. Each loop cycle would operate on corresponding bytes (containing two decimal digits) of operands.

Endian Conversion

- BSWAP—Byte Swap

The BSWAP instruction changes the byte order of a doubleword or quadword operand in a register, as shown in Figure 3-8. In a doubleword, bits 7:0 are exchanged with bits 31:24, and bits 15:8 are exchanged with bits 23:16. In a quadword, bits 7:0 are exchanged with bits 63:56, bits 15:8 with bits 55:48, bits 23:16 with bits 47:40, and bits 31:24 with bits 39:32. See the following illustration.

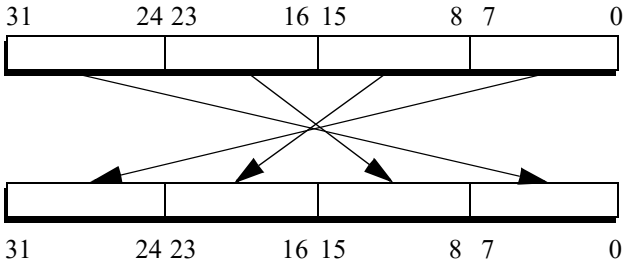


Figure 3-8. BSWAP Doubleword Exchange

A second application of the BSWAP instruction to the same operand restores its original value. The result of applying the BSWAP instruction to a 16-bit register is undefined. To swap bytes of a 16-bit register, use the XCHG instruction.

The BSWAP instruction is used to convert data between little-endian and big-endian byte order.

3.3.4 Load Segment Registers

These instructions load segment registers.

- LDS, LES, LFS, LGS, LSS—Load Far Pointer
- MOV segReg—Move Segment Register
- POP segReg—Pop Stack Into Segment Register

The LDS, LES, LFD, LGS, and LSS instructions atomically (with respect to interrupts only, not contending memory accesses) load the two parts of a far pointer into a segment register and a general-purpose register. A far pointer is a 16-bit segment selector and a 16-bit or 32-bit offset. The load copies the segment-selector portion of the pointer from memory into the segment register and the offset portion of the pointer from memory into a general-purpose register.

The effective operand size determines the size of the offset loaded by the LDS, LES, LFD, LGS, and LSS instructions. The instructions load not only the software-visible segment selector into the segment register, but they also cause the hardware to load the associated segment-descriptor information into the software-invisible (hidden) portion of that segment register.

The MOV segReg and POP segReg instructions load a segment selector from a general-purpose register or memory (for MOV segReg) or from the top of the stack (for POP segReg) to a segment register. These instructions not only load the software-visible segment selector into the segment register but also cause the hardware to load the associated segment-descriptor information into the software-invisible (hidden) portion of that segment register.

In 64-bit mode, the POP DS, POP ES, and POP SS instructions are invalid.

3.3.5 Load Effective Address

- LEA—Load Effective Address

The LEA instruction calculates and loads the effective address (offset within a given segment) of a source operand and places it in a general-purpose register.

LEA is related to MOV, which copies data from a memory location to a register, but LEA takes the address of the source operand, whereas MOV takes the contents of the memory location specified by the source operand. In the simplest cases, LEA can be replaced with MOV. For example:

```
lea eax, [ebx]
```

has the same effect as:

```
mov eax, ebx
```

However, LEA allows software to use any valid addressing mode for the source operand. For example:

```
lea eax, [ebx+edi]
```

loads the sum of EBX and EDI registers into the EAX register. This could not be accomplished by a single MOV instruction.

LEA has a limited capability to perform multiplication of operands in general-purpose registers using scaled-index addressing. For example:

```
lea eax, [ebx+ebx*8]
```

loads the value of the EBX register, multiplied by 9, into the EAX register.

3.3.6 Arithmetic

The arithmetic instructions perform basic arithmetic operations, such as addition, subtraction, multiplication, and division on integer operands.

Add and Subtract

- ADC—Add with Carry
- ADD—Signed or Unsigned Add
- SBB—Subtract with Borrow
- SUB—Subtract
- NEG—Two's Complement Negation

The ADD instruction performs addition of two integer operands. There are opcodes that add an immediate value to a byte, word, doubleword, or quadword register or a memory location. In these opcodes, if the size of the immediate is smaller than that of the destination, the immediate is first sign-extended to the size of the destination operand. The arithmetic flags (OF, SF, ZF, AF, CF, PF) are set according to the resulting value of the destination operand.

The ADC instruction performs addition of two integer operands, plus 1 if the carry flag (CF) is set.

The SUB instruction performs subtraction of two integer operands.

The SBB instruction performs subtraction of two integer operands, and it also subtracts an additional 1 if the carry flag is set.

The ADC and SBB instructions simplify addition and subtraction of multiple-precision integer operands, because they correctly handle carries (and borrows) between parts of a multiple-precision operand.

The NEG instruction performs negation of an integer operand. The value of the operand is replaced with the result of subtracting the operand from zero.

Multiply and Divide

- MUL—Multiply Unsigned
- IMUL—Signed Multiply
- DIV—Unsigned Divide
- IDIV—Signed Divide

The MUL instruction performs multiplication of unsigned integer operands. The size of operands can be byte, word, doubleword, or quadword. The product is stored in a destination which is double the size of the source operands (multiplicand and factor).

The MUL instruction's mnemonic has only one operand, which is a factor. The multiplicand operand is always assumed to be an accumulator register. For byte-sized multiplies, AL contains the multiplicand, and the result is stored in AX. For word-sized, doubleword-sized, and quadword-sized multiplies, rAX contains the multiplicand, and the result is stored in rDX and rAX. In 64-bit mode

The IMUL instruction performs multiplication of signed integer operands. There are forms of the IMUL instruction with one, two, and three operands, and it is thus more powerful than the MUL instruction. The one-operand form of the IMUL instruction behaves similarly to the MUL instruction, except that the operands and product are signed integer values. In the two-operand form of IMUL, the multiplicand and product use the same register (the first operand), and the factor is specified in the second operand. In the three-operand form of IMUL, the product is stored in the first operand, the multiplicand is specified in the second operand, and the factor is specified in the third operand.

The DIV instruction performs division of unsigned integers. The instruction divides a double-sized dividend in AH:AL or rDX:rAX by the divisor specified in the operand of the instruction. It stores the quotient in AL or rAX and the remainder in AH or rDX.

The IDIV instruction performs division of signed integers. It behaves similarly to DIV, with the exception that the operands are treated as signed integer values.

Division is the slowest of all integer arithmetic operations and should be avoided wherever possible. One possibility for improving performance is to replace division with multiplication, such as by replacing *i/j/k* with *i/(j*k)*. This replacement is possible if no overflow occurs during the computation of the product. This can be determined by considering the possible ranges of the divisors.

Increment and Decrement

- DEC—Decrement by 1
- INC—Increment by 1

The INC and DEC instructions are used to increment and decrement, respectively, an integer operand by one. For both instructions, an operand can be a byte, word, doubleword, or quadword register or memory location.

These instructions behave in all respects like the corresponding ADD and SUB instructions, with the second operand as an immediate value equal to 1. The only exception is that the carry flag (CF) is not affected by the INC and DEC instructions.

Apart from their obvious arithmetic uses, the INC and DEC instructions are often used to modify addresses of operands. In this case it can be desirable to preserve the value of the carry flag (to use it later), so these instructions do not modify the carry flag.

3.3.7 Rotate and Shift

The rotate and shift instructions perform cyclic rotation or non-cyclic shift, by a given number of bits (called the *count*), in a given byte-sized, word-sized, doubleword-sized or quadword-sized operand.

When the count is greater than 1, the result of the rotate and shift instructions can be considered as an iteration of the same 1-bit operation by *count* number of times. Because of this, the descriptions below describe the result of 1-bit operations.

The count can be 1, the value of the CL register, or an immediate 8-bit value. To avoid redundancy and make rotation and shifting quicker, the count is masked to the 5 or 6 least-significant bits, depending on the effective operand size, so that its value does not exceed 31 or 63 before the rotation or shift takes place.

Rotate

- RCL—Rotate Through Carry Left
- RCR—Rotate Through Carry Right
- ROL—Rotate Left
- ROR—Rotate Right

The RC*x* instructions rotate the bits of the first operand to the left or right by the number of bits specified by the source (count) operand. The bits rotated out of the destination operand are rotated into the carry flag (CF) and the carry flag is rotated into the opposite end of the first operand.

The RO*x* instructions rotate the bits of the first operand to the left or right by the number of bits specified by the source operand. Bits rotated out are rotated back in at the opposite end. The value of the CF flag is determined by the value of the last bit rotated out. In single-bit left-rotates, the overflow flag (OF) is set to the XOR of the CF flag after rotation and the most-significant bit of the result. In single-bit right-rotates, the OF flag is set to the XOR of the two most-significant bits. Thus, in both cases, the OF flag is set to 1 if the single-bit rotation changed the value of the most-significant bit (sign bit) of the operand. The value of the OF flag is undefined for multi-bit rotates.

Bit-rotation instructions provide many ways to reorder bits in an operand. This can be useful, for example, in character conversion, including cryptography techniques.

Shift

- SAL—Shift Arithmetic Left
- SAR—Shift Arithmetic Right
- SHL—Shift Left
- SHR—Shift Right
- SHLD—Shift Left Double
- SHRD—Shift Right Double

The SH*x* instructions (including SHxD) perform shift operations on unsigned operands. The SA*x* instructions operate with signed operands.

SHL and SAL instructions effectively perform multiplication of an operand by a power of 2, in which case they work as more-efficient alternatives to the MUL instruction. Similarly, SHR and SAR instructions can be used to divide an operand (signed or unsigned, depending on the instruction used) by a power of 2.

Although the SAR instruction divides the operand by a power of 2, the behavior is different from the IDIV instruction. For example, shifting –11 (FFFFFFF5h) by two bits to the right (i.e. divide –11 by 4), gives a result of FFFFFFFDh, or –3, whereas the IDIV instruction for dividing –11 by 4 gives a result of –2. This is because the IDIV instruction rounds off the quotient to zero, whereas the SAR instruction rounds off the remainder to zero for positive dividends, and to negative infinity for negative dividends. This means that, for positive operands, SAR behaves like the corresponding IDIV instruction, and for negative operands, it gives the same result if and only if all the shifted-out bits are zeroes, and otherwise the result is smaller by 1.

The SAR instruction treats the most-significant bit (msb) of an operand in a special way: the msb (the sign bit) is not changed, but is copied to the next bit, preserving the sign of the result. The least-significant bit (lsb) is shifted out to the CF flag. In the SAL instruction, the msb is shifted out to CF flag, and the lsb is cleared to 0.

The SH*x* instructions perform *logical shift*, i.e. without special treatment of the sign bit. SHL is the same as SAL (in fact, their opcodes are the same). SHR copies 0 into the most-significant bit, and shifts the least-significant bit to the CF flag.

The SHxD instructions perform a double shift. These instructions perform left and right shift of the destination operand, taking the bits to copy into the most-significant bit (for the SHRD instruction) or into the least-significant bit (for the SHLD instruction) from the source operand. These instructions behave like SHx, but use bits from the source operand instead of zero bits to shift into the destination operand. The source operand is not changed.

3.3.8 Bit Manipulation

The bit manipulation instructions manipulate individual bits in a register for purposes such as controlling low-level devices, correcting algorithms, and detecting errors. Following are descriptions of supported bit manipulation instructions.

Extract Bit Field

- BEXTR—Bit Field Extract (register form is a BMI instruction)
- BEXTR—Bit Field Extract (immediate version is a TBM instruction)

The BEXTR instruction (register form and immediate version) extracts a contiguous field of bits from the first source operand, as specified by the control field setting in the second source operand and puts the extracted field into the least significant bit positions of the destination. The remaining bits in the destination register are cleared to 0.

Fill Bit

- BLCFILL—Fill From Lowest Clear Bit
- BLSFILL—Fill From Lowest Set Bit

The BLCFILL instruction finds the least significant zero bit in the source operand, clears all bits below that bit to 0 and writes the result to the destination. If there is no zero bit in the source operand, the destination is written with all zeros.

The BLSFILL instruction finds the least significant one bit in the source operand, sets all bits below that bit to 1 and writes the result to the destination. If there is no one bit in the source operand, the destination is written with all ones.

Isolate Bit

- BLSI— Isolate Lowest Set Bit
- BLCI—Isolate Lowest Clear Bit
- BLCIC—Bit Lowest Clear Isolate Complemented
- BLCS—Set Lowest Clear Bit
- BLSIC— Isolate Lowest Set Bit and Complement

The BLSI instruction clears all bits in the source operand except for the least significant bit that is set to 1 and writes the result to the destination.

The BLCI instruction finds the least significant zero bit in the source operand, sets all other bits to 1 and writes the result to the destination. If there is no zero bit in the source operand, the destination is written with all ones.

The BLCIC instruction finds the least significant zero bit in the source operand, sets that bit to 1, clears all other bits to 0 and writes the result to the destination. If there is no zero bit in the source operand, the destination is written with all zeros.

The BLCS instruction finds the least significant zero bit in the source operand, sets that bit to 1 and writes the result to the destination. If there is no zero bit in the source operand, the source is copied to the destination (and CF in rFLAGS is set to 1).

The BLSIC instruction finds the least significant bit that is set to 1 in the source operand, clears that bit to 0, sets all other bits to 1 and writes the result to the destination. If there is no one bit in the source operand, the destination is written with all ones.

Mask Bit

- BLSMSK—Mask from Lowest Set Bit
- BLCMSK—Mask From Lowest Clear Bit
- T1MSKC — Inverse Mask From Trailing Ones
- TZMSK—Mask From Trailing Zeros

The BLSMSK instruction forms a mask with bits set to 1 from bit 0 up to and including the least significant bit position that is set to 1 in the source operand and writes the mask to the destination.

The BLCMSK instruction finds the least significant zero bit in the source operand, sets that bit to 1, clears all bits above that bit to 0 and writes the result to the destination. If there is no zero bit in the source operand, the destination is written with all ones.

The T1MSKC instruction finds the least significant zero bit in the source operand, clears all bits below that bit to 0, sets all other bits to 1 (including the found bit) and writes the result to the destination. If the least significant bit of the source operand is 0, the destination is written with all ones.

The TZMSK instruction finds the least significant one bit in the source operand, sets all bits below that bit to 1, clears all other bits to 0 (including the found bit) and writes the result to the destination. If the least significant bit of the source operand is 1, the destination is written with all zeros.

Population and Zero Counts

- POPCNT—Bit Population Count
- LZCNT—Count Leading Zeros
- TZCNT—Trailing Zero Count

The POPCNT instruction counts the number of bits having a value of 1 in the source operand and places the total in the destination register.

The LZCNT instruction counts the number of leading zero bits in the 16-, 32-, or 64-bit general purpose register or memory source operand. Counting starts downward from the most significant bit and stops when the highest bit having a value of 1 is encountered or when the least significant bit is encountered. The count is written to the destination register.

The TZCNT instruction counts the number of trailing zero bits in the 16-, 32-, or 64-bit general purpose register or memory source operand. Counting starts upward from the least significant bit and stops when the lowest bit having a value of 1 is encountered or when the most significant bit is encountered. The count is written to the destination register.

Reset Bit

- BLSR—Reset Lowest Set Bit

The BLSR instruction clears the least-significant bit that is set to 1 in the input operand and writes the modified operand to the destination.

Scan Bit

- BSF—Bit Scan Forward
- BSR—Bit Scan Reverse

The BSF and BSR instructions search a source operand for the least-significant (BSF) or most-significant (BSR) bit that is set to 1. If a set bit is found, its bit index is loaded into the destination

operand, and the zero flag (ZF) is set. If no set bit is found, the zero flag is cleared and the contents of the destination are undefined.

3.3.9 Compare and Test

The compare and test instructions perform arithmetic and logical comparison of operands and set corresponding flags, depending on the result of comparison. These instruction are used in conjunction with conditional instructions such as J*cc* or SET*cc* to organize branching and conditionally executing blocks in programs. Assembler equivalents of conditional operators in high-level languages (do…while, if…then…else, and similar) also include compare and test instructions.

Compare

- CMP—Compare

The CMP instruction performs subtraction of the second operand (source) from the first operand (destination), like the SUB instruction, but it does not store the resulting value in the destination operand. It leaves both operands intact. The only effect of the CMP instruction is to set or clear the arithmetic flags (OF, SF, ZF, AF, CF, PF) according to the result of subtraction.

The CMP instruction is often used together with the conditional jump instructions (J*cc*), conditional SET instructions (SET*cc*) and other instructions such as conditional loops (LOOP*cc*) whose behavior depends on flag state.

Test

- TEST—Test Bits

The TEST instruction is in many ways similar to the AND instruction: it performs logical conjunction of the corresponding bits of both operands, but unlike the AND instruction it leaves the operands unchanged. The purpose of this instruction is to update flags for further testing.

The TEST instruction is often used to test whether one or more bits in an operand are zero. In this case, one of the instruction operands would contain a mask in which all bits are cleared to zero except the bits being tested. For more advanced bit testing and bit modification, use the BT*x* instructions.

Bit Test

- BT—Bit Test
- BTC—Bit Test and Complement
- BTR—Bit Test and Reset
- BTS—Bit Test and Set

The BT*x* instructions copy a specified bit in the first operand to the carry flag (CF) and leave the source bit unchanged (BT), or complement the source bit (BTC), or clear the source bit to 0 (BTR), or set the source bit to 1 (BTS).

These instructions are useful for implementing semaphore arrays. Unlike the XCHG instruction, the BT*x* instructions set the carry flag, so no additional test or compare instruction is needed. Also,

because these instructions operate directly on bits rather than larger data types, the semaphore arrays can be smaller than is possible when using XCHG. In such semaphore applications, bit-test instructions should be preceded by the LOCK prefix.

Set Byte on Condition

- SET*cc*—Set Byte if *condition*

The SET*cc* instructions store a 1 or 0 value to their byte operand depending on whether their condition (represented by certain rFLAGS bits) is true or false, respectively. Table 3-5 shows the rFLAGS values required for each SET*cc* instruction.

Table 3-5. rFLAGS for SET*cc* Instructions

Mnemonic	Required Flag State	Description
SETO	OF = 1	Set byte if overflow
SETNO	OF = 0	Set byte if not overflow
SETB SETC SETNAE	CF = 1	Set byte if below Set byte if carry Set byte if not above or equal (unsigned operands)
SETAE SETNB SETNC	CF = 0	Set byte if above or equal Set byte if not below Set byte if not carry (unsigned operands)
SETE SETZ	ZF = 1	Set byte if equal Set byte if zero
SETNE SETNZ	ZF = 0	Set byte if not equal Set byte if not zero
SETBE SETNA	CF = 1 or ZF = 1	Set byte if below or equal Set byte if not above (unsigned operands)
SETA SETNBE	CF = 0 and ZF = 0	Set byte if not below or equal Set byte if not below or equal (unsigned operands)
SETS	SF = 1	Set byte if sign
SETNS	SF = 0	Set byte if not sign
SETP SETPE	PF = 1	Set byte if parity Set byte if parity even
SETNP SETPO	PF = 0	Set byte if not parity Set byte if parity odd
SETL SETNGE	SF <> OF	Set byte if less Set byte if not greater or equal (signed operands)
SETGE SETNL	SF = OF	Set byte if greater or equal Set byte if not less (signed operands)
SETLE SETNG	ZF = 1 or SF <> OF	Set byte if less or equal Set byte if not greater (signed operands)
SETG SETNLE	ZF = 0 and SF = OF	Set byte if greater Set byte if not less or equal (signed operands)

SET*cc* instructions are often used to set logical indicators. Like CMOV*cc* instructions (page 45), SET*cc* instructions can replace two instructions—a conditional jump and a move. Replacing conditional jumps with conditional sets can help avoid branch-prediction penalties that may be caused by conditional jumps.

If the logical value True (logical 1) is represented in a high-level language as an integer with all bits set to 1, software can accomplish such representation by first executing the opposite SET*cc* instruction—for example, the opposite of SETZ is SETNZ—and then decrementing the result.

Bounds

- BOUND—Check Array Bounds

The BOUND instruction checks whether the value of the first operand, a signed integer index into an array, is within the minimal and maximal bound values pointed to by the second operand. The values of array bounds are often stored at the beginning of the array. If the bounds of the range are exceeded, the processor generates a bound-range exception.

The primary disadvantage of using the BOUND instruction is its use of the time-consuming exception mechanism to signal a failure of the bounds test.

3.3.10 Logical

The logical instructions perform bitwise operations.

- AND—Logical AND
- OR—Logical OR
- XOR—Exclusive OR
- NOT—One’s Complement Negation
- ANDN—And Not

The AND, OR, and XOR instructions perform their respective logical operations on the corresponding bits of both operands and store the result in the first operand. The CF flag and OF flag are cleared to 0, and the ZF flag, SF flag, and PF flag are set according to the resulting value of the first operand.

The NOT instruction performs logical inversion of all bits of its operand. Each zero bit becomes one and vice versa. All flags remain unchanged.

The ANDN instruction performs a bitwise AND of the second source operand and the one's complement of the first source operand and stores the result into the destination operand.

Apart from performing logical operations, AND and OR can test a register for a zero or non-zero value, sign (negative or positive), and parity status of its lowest byte. To do this, both operands must be the same register. The XOR instruction with two identical operands is an efficient way of loading the value 0 into a register.

3.3.11 String

The string instructions perform common string operations such as copying, moving, comparing, or searching strings. These instructions are widely used for processing text.

Compare Strings

- CMPS—Compare Strings
- CMPSB—Compare Strings by Byte
- CMPSW—Compare Strings by Word
- CMPSD—Compare Strings by Doubleword
- CMPSQ—Compare Strings by Quadword

The CMPS*x* instructions compare the values of two implicit operands of the same size located at *seg*:[rSI] and ES:[rDI]. After the copy, both the rSI and rDI registers are auto-incremented (if the DF flag is 0) or auto-decremented (if the DF flag is 1).

Scan String

- SCAS—Scan String
- SCASB—Scan String as Bytes
- SCASW—Scan String as Words
- SCASD—Scan String as Doubleword
- SCASQ—Scan String as Quadword

The SCAS*x* instructions compare the values of a memory operands in ES:rDI to a value of the same size in the AL/rAX register. Bits in rFLAGS are set to indicate the outcome of the comparison. After the comparison, the rDI register is auto-incremented (if the DF flag is 0) or auto-decremented (if the DF flag is 1).

Move String

- MOVS—Move String
- MOVSB—Move String Byte
- MOVSW—Move String Word
- MOVSD—Move String Doubleword
- MOVSQ—Move String Quadword

The MOVS*x* instructions copy an operand from the memory location *seg*:[rSI] to the memory location ES:[rDI]. After the copy, both the rSI and rDI registers are auto-incremented (if the DF flag is 0) or auto-decremented (if the DF flag is 1).

Load String

- LODS—Load String

- LODSB—Load String Byte
- LODSW—Load String Word
- LODSD—Load String Doubleword
- LODSQ—Load String Quadword

The LODS*x* instructions load a value from the memory location *seg*:[rSI] to the accumulator register (AL or rAX). After the load, the rSI register is auto-incremented (if the DF flag is 0) or auto-decremented (if the DF flag is 1).

Store String

- STOS—Store String
- STOSB—Store String Bytes
- STOSW—Store String Words
- STOSD—Store String Doublewords
- STOSQ—Store String Quadword

The STOS*x* instructions copy the accumulator register (AL or rAX) to a memory location ES:[rDI]. After the copy, the rDI register is auto-incremented (if the DF flag is 0) or auto-decremented (if the DF flag is 1).

3.3.12 Control Transfer

Control-transfer instructions, or branches, are used to iterate through loops and move through conditional program logic.

Jump

- JMP—Jump

JMP performs an unconditional jump to the specified address. There are several ways to specify the target address.

- *Relative Short Jump* and *Relative Near Jump*—The target address is determined by adding an 8-bit (short jump) or 16-bit or 32-bit (near jump) signed displacement to the rIP of the instruction following the JMP. The jump is performed within the current code segment (CS).
- *Register-Indirect* and *Memory-Indirect Near Jump*—The target rIP value is contained in a register or in a memory location. The jump is performed within the current CS.
- *Direct Far Jump*—For all far jumps, the target address is outside the current code segment. Here, the instruction specifies the 16-bit target-address code segment and the 16-bit or 32-bit offset as an immediate value. The direct far jump form is invalid in 64-bit mode.
- *Memory-Indirect Far Jump*—For this form, the target address (CS:rIP) is in a address outside the current code segment. A 32-bit or 48-bit far pointer in a specified memory location points to the target address.

The size of the target rIP is determined by the effective operand size for the JMP instruction.

For far jumps, the target selector can specify a code-segment selector, in which case it is loaded into CS, and a 16-bit or 32-bit target offset is loaded into rIP. The target selector can also be a call-gate selector or a task-state-segment (TSS) selector, used for performing task switches. In these cases, the target offset of the JMP instruction is ignored, and the new values loaded into CS and rIP are taken from the call gate or from the TSS.

Conditional Jump

- J*cc*—Jump if *condition*

Conditional jump instructions jump to an instruction specified by the operand, depending on the state of flags in the rFLAGS register. The operands specifies a signed relative offset from the current contents of the rIP. If the state of the corresponding flags meets the condition, a conditional jump instruction passes control to the target instruction, otherwise control is passed to the instruction following the conditional jump instruction. The flags tested by a specific J*cc* instruction depend on the opcode. In several cases, multiple mnemonics correspond to one opcode.

Table 3-6 shows the rFLAGS values required for each J*cc* instruction.

Table 3-6. rFLAGS for J*cc* Instructions

Mnemonic	Required Flag State	Description
JO	OF = 1	Jump near if overflow
JNO	OF = 0	Jump near if not overflow
JB JC JNAE	CF = 1	Jump near if below Jump near if carry Jump near if not above or equal
JNB JNC JAE	CF = 0	Jump near if not below Jump near if not carry Jump near if above or equal
JZ JE	ZF = 1	Jump near if 0 Jump near if equal
JNZ JNE	ZF = 0	Jump near if not zero Jump near if not equal
JNA JBE	CF = 1 or ZF = 1	Jump near if not above Jump near if below or equal
JNBE JA	CF = 0 and ZF = 0	Jump near if not below or equal Jump near if above
JS	SF = 1	Jump near if sign
JNS	SF = 0	Jump near if not sign
JP JPE	PF = 1	Jump near if parity Jump near if parity even
JNP JPO	PF = 0	Jump near if not parity Jump near if parity odd
JL JNGE	SF <> OF	Jump near if less Jump near if not greater or equal

Table 3-6. rFLAGS for J*cc* Instructions (continued)

Mnemonic	Required Flag State	Description
JGE JNL	SF = OF	Jump near if greater or equal Jump near if not less
JNG JLE	ZF = 1 or SF <> OF	Jump near if not greater Jump near if less or equal
JNLE JG	ZF = 0 and SF = OF	Jump near if not less or equal Jump near if greater

Unlike the unconditional jump (JMP), conditional jump instructions have only two forms—*near conditional jumps* and *short conditional jumps*. To create a far-conditional-jump code sequence corresponding to a high-level language statement like:

```
IF A = B THEN GOTO FarLabel
```

where FarLabel is located in another code segment, use the opposite condition in a conditional short jump before the unconditional far jump. For example:

```
      cmp   A,B               ; compare operands
      jne   NextInstr         ; continue program if not equal
      jmp   far ptr WhenNE    ; far jump if operands are equal
NextInstr:                    ; continue program
```

Three special conditional jump instructions use the rCX register instead of flags. The JCXZ, JECXZ, and JRCXZ instructions check the value of the CX, ECX, and RCX registers, respectively, and pass control to the target instruction when the value of rCX register reaches 0. These instructions are often used to control safe cycles, preventing execution when the value in rCX reaches 0.

Loop

- LOOP*cc*—Loop if *condition*

The LOOP*cc* instructions include LOOPE, LOOPNE, LOOPNZ, and LOOPZ. These instructions decrement the rCX register by 1 without changing any flags, and then check to see if the loop condition is met. If the condition is met, the program jumps to the specified target code.

LOOPE and LOOPZ are synonyms. Their loop condition is met if the value of the rCX register is non-zero and the zero flag (ZF) is set to 1 when the instruction starts. LOOPNE and LOOPNZ are also synonyms. Their loop condition is met if the value of the rCX register is non-zero and the ZF flag is cleared to 0 when the instruction starts. LOOP, unlike the other mnemonics, does not check the ZF flag. Its loop condition is met if the value of the rCX register is non-zero.

Call

- CALL—Procedure Call

The CALL instruction performs a call to a procedure whose address is specified in the operand. The return address is placed on the stack by the CALL, and points to the instruction immediately following

the CALL. When the called procedure finishes execution and is exited using a return instruction, control is transferred to the return address saved on the stack.

The CALL instruction has the same forms as the JMP instruction, except that CALL lacks the short-relative (1-byte offset) form.

- *Relative Near Call*—These specify an offset relative to the instruction following the CALL instruction. The operand is an immediate 16-bit or 32-bit offset from the called procedure, within the same code segment.
- *Register-Indirect* and *Memory-Indirect Near Call*—These specify a target address contained in a register or memory location.
- *Direct Far Call*—These specify a target address outside the current code segment. The address is pointed to by a 32-bit or 48-bit far-pointer specified by the instruction, which consists of a 16-bit code selector and a 16-bit or 32-bit offset. The direct far call form is invalid in 64-bit mode.
- *Memory-Indirect Far Call*—These specify a target address outside the current code segment. The address is pointed to by a 32-bit or 48-bit far pointer in a specified memory location.

The size of the rIP is in all cases determined by the operand-size attribute of the CALL instruction. CALLs push the return address to the stack. The data pushed on the stack depends on whether a near or far call is performed, and whether a privilege change occurs. See Section 3.7.5, “Procedure Calls,” on page 83 for further information.

For far CALLs, the selector portion of the target address can specify a code-segment selector (in which case the selector is loaded into the CS register), or a call-gate selector, (used for calls that change privilege level), or a task-state-segment (TSS) selector (used for task switches). In the latter two cases, the offset portion of the CALL instruction’s target address is ignored, and the new values loaded into CS and rIP are taken from the call gate or TSS.

Return

- RET—Return from Call

The RET instruction returns from a procedure originally called using the CALL instruction. CALL places a return address (which points to the instruction following the CALL) on the stack. RET takes the return address from the stack and transfers control to the instruction located at that address.

Like CALL instructions, RET instructions have both a near and far form. An optional immediate operand for the RET specifies the number of bytes to be popped from the procedure stack for parameters placed on the stack. See Section 3.7.6, “Returning from Procedures,” on page 86 for additional information.

Interrupts and Exceptions

- INT—Interrupt to Vector Number
- INTO—Interrupt to Overflow Vector
- IRET—Interrupt Return Word
- IRETD—Interrupt Return Doubleword

- IRETQ—Interrupt Return Quadword

The INT instruction implements a *software interrupt* by calling an interrupt handler. The operand of the INT instruction is an immediate byte value specifying an index in the interrupt descriptor table (IDT), which contains addresses of interrupt handlers (see Section 3.7.10, “Interrupts and Exceptions,” on page 91 for further information on the IDT).

The 1-byte INTO instruction calls interrupt 4 (the overflow exception, #OF), if the overflow flag in RFLAGS is set to 1, otherwise it does nothing. Signed arithmetic instructions can be followed by the INTO instruction if the result of the arithmetic operation can potentially overflow. (The 1-byte INT 3 instruction is considered a system instruction and is therefore not described in this volume).

IRET, IRETD, and IRETQ perform a return from an interrupt handler. The mnemonic specifies the operand size, which determines the format of the return addresses popped from the stack (IRET for 16-bit operand size, IRETD for 32-bit operand size, and IRETQ for 64-bit operand size). However, some assemblers can use the IRET mnemonic for all operand sizes. Actions performed by IRET are opposite to actions performed by an interrupt or exception. In real and protected mode, IRET pops the rIP, CS, and RFLAGS contents from the stack, and it pops SS:rSP if a privilege-level change occurs or if it executes from 64-bit mode. In protected mode, the IRET instruction can also cause a task switch if the nested task (NT) bit in the RFLAGS register is set. For details on using IRET to switch tasks, see “Task Management” in Volume 2.

3.3.13 Flags

The flags instructions read and write bits of the RFLAGS register that are visible to application software. “Flags Register” on page 34 illustrates the RFLAGS register.

Push and Pop Flags

- POPF—Pop to FLAGS Word
- POPFD—Pop to EFLAGS Doubleword
- POPFQ—Pop to RFLAGS Quadword
- PUSHF—Push FLAGS Word onto Stack
- PUSHFD—Push EFLAGS Doubleword onto Stack
- PUSHFQ—Push RFLAGS Quadword onto Stack

The push and pop flags instructions copy data between the rFLAGS register and the stack. POPF and PUSHF copy 16 bits of data between the stack and the FLAGS register (the low 16 bits of EFLAGS), leaving the high 48 bits of RFLAGS unchanged. POPFD and PUSHFD copy 32 bits between the stack and the RFLAGS register. POPFQ and PUSHFQ copy 64 bits between the stack and the RFLAGS register. Only the bits illustrated in Figure 3-5 on page 34 are affected. Reserved bits and bits that are write protected by the current values of system flags, current privilege level (CPL), or current operating mode are unaffected by the POPF, POPFQ, and POPFD instructions.

For details on stack operations, see “Control Transfers” on page 80.

Set and Clear Flags

- CLC—Clear Carry Flag
- CMC—Complement Carry Flag
- STC—Set Carry Flag
- CLD—Clear Direction Flag
- STD—Set Direction Flag
- CLI—Clear Interrupt Flag
- STI—Set Interrupt Flag

These instructions change the value of a flag in the rFLAGS register that is visible to application software. Each instruction affects only one specific flag.

The CLC, CMC, and STC instructions change the carry flag (CF). CLC clears the flag to 0, STC sets the flag to 1, and CMC inverts the flag. These instructions are useful prior to executing instructions whose behavior depends on the CF flag—for example, shift and rotate instructions.

The CLD and STD instructions change the direction flag (DF) and influence the function of string instructions (CMPSx, SCASx, MOVSx, LODSx, STOSx, INSx, OUTSx). CLD clears the flag to 0, and STD sets the flag to 1. A cleared DF flag indicates the *forward* direction in string sequences, and a set DF flag indicates the *backward* direction. Thus, in string instructions, the rSI and/or rDI register values are auto-incremented when DF = 0 and auto-decremented when DF = 1.

Two other instructions, CLI and STI, clear and set the interrupt flag (IF). CLI clears the flag, causing the processor to ignore external maskable interrupts. STI sets the flag, allowing the processor to recognize maskable external interrupts. These instructions are used primarily by system software—especially, interrupt handlers—and are described in “Exceptions and Interrupts” in Volume 2.

Load and Store Flags

- LAHF—Load Status Flags into AH Register
- SAHF—Store AH into Flags

LAHF loads the lowest byte of the RFLAGS register into the AH register. This byte contains the carry flag (CF), parity flag (PF), auxiliary flag (AF), zero flag (ZF), and sign flag (SF). SAHF stores the AH register into the lowest byte of the RFLAGS register.

3.3.14 Input/Output

The I/O instructions perform reads and writes of bytes, words, and doublewords from and to the *I/O address space*. This address space can be used to access and manage external devices, and is independent of the main-memory address space. By contrast, *memory-mapped I/O* uses the main-memory address space and is accessed using the MOV instructions rather than the I/O instructions.

When operating in legacy protected mode or in long mode, the RFLAGS register's I/O privilege level (IOPL) field and the I/O-permission bitmap in the current task-state segment (TSS) are used to control access to the I/O addresses (called *I/O ports*). See "Input/Output" on page 95 for further information.

General I/O

- IN—Input from Port
- OUT—Output to Port

The IN instruction reads a byte, word, or doubleword from the I/O port address specified by the source operand, and loads it into the accumulator register (AL or eAX). The source operand can be an immediate byte or the DX register.

The OUT instruction writes a byte, word, or doubleword from the accumulator register (AL or eAX) to the I/O port address specified by the destination operand, which can be either an immediate byte or the DX register.

If the I/O port address is specified with an immediate operand, the range of port addresses accessible by the IN and OUT instructions is limited to ports 0 through 255. If the I/O port address is specified by a value in the DX register, all 65,536 ports are accessible.

String I/O

- INS—Input String
- INSB—Input String Byte
- INSW—Input String Word
- INSD—Input String Doubleword
- OUTS—Output String
- OUTSB—Output String Byte
- OUTSW—Output String Word
- OUTSD—Output String Doubleword

The INS*x* instructions (INSB, INSW, INSD) read a byte, word, or doubleword from the I/O port specified by the DX register, and load it into the memory location specified by ES:[rDI].

The OUTS*x* instructions (OUTSB, OUTSW, OUTSD) write a byte, word, or doubleword from an implicit memory location specified by *seg*:[rSI], to the I/O port address stored in the DX register.

The INS*x* and OUTS*x* instructions are commonly used with a repeat prefix to transfer blocks of data. The memory pointer address is not incremented or decremented. This usage is intended for peripheral I/O devices that are expecting a stream of data.

3.3.15 Semaphores

The semaphore instructions support the implementation of reliable signaling between processors in a multi-processing environment, usually for the purpose of sharing resources.

- CMPXCHG—Compare and Exchange
- CMPXCHG8B—Compare and Exchange Eight Bytes
- CMPXCHG16B—Compare and Exchange Sixteen Bytes
- XADD—Exchange and Add
- XCHG—Exchange

The CMPXCHG instruction compares a value in the AL or rAX register with the first (destination) operand, and sets the arithmetic flags (ZF, OF, SF, AF, CF, PF) according to the result. If the compared values are equal, the source operand is loaded into the destination operand. If they are not equal, the first operand is loaded into the accumulator. CMPXCHG can be used to try to intercept a semaphore, i.e. test if its state is *free*, and if so, load a new value into the semaphore, making its state *busy*. The test and load are performed atomically, so that concurrent processes or threads which use the semaphore to access a shared object will not conflict.

The CMPXCHG8B instruction compares the 64-bit values in the EDX:EAX registers with a 64-bit memory location. If the values are equal, the zero flag (ZF) is set, and the ECX:EBX value is copied to the memory location. Otherwise, the ZF flag is cleared, and the memory value is copied to EDX:EAX.

The CMPXCHG16B instruction compares the 128-bit value in the RDX:RAX and RCX:RBX registers with a 128-bit memory location. If the values are equal, the zero flag (ZF) is set, and the RCX:RBX value is copied to the memory location. Otherwise, the ZF flag is cleared, and the memory value is copied to rDX:rAX.

The XADD instruction exchanges the values of its two operands, then it stores their sum in the first (destination) operand.

A LOCK prefix can be used to make the CMPXCHG, CMPXCHG8B and XADD instructions atomic if one of the operands is a memory location.

The XCHG instruction exchanges the values of its two operands. If one of the operands is in memory, the processor's bus-locking mechanism is engaged automatically during the exchange, whether or not the LOCK prefix is used.

3.3.16 Processor Information

- CPUID—Processor Identification

The CPUID instruction returns information about the processor implementation and its support for instruction subsets and architectural features. Software operating at any privilege level can execute the CPUID instruction to read this information. After the information is read, software can select procedures that optimize performance for a particular hardware implementation.

Some processor implementations may not support the CPUID instruction. Support for the CPUID instruction is determined by testing the RFLAGS.ID bit. If software can write this bit, then the CPUID instruction is supported by the processor implementation. Otherwise, execution of CPUID results in an invalid-opcode exception.

See Section 3.6, “Feature Detection,” on page 79 for details about using the CPUID instruction.

3.3.17 Cache and Memory Management

Applications can use the cache and memory-management instructions to control memory reads and writes to influence the caching of read/write data. “Memory Optimization” on page 98 describes how these instructions interact with the memory subsystem.

- LFENCE—Load Fence
- SFENCE—Store Fence
- MFENCE—Memory Fence
- PREFETCH*level*—Prefetch Data to Cache Level *level*
- PREFETCH—Prefetch L1 Data-Cache Line
- PREFETCHW—Prefetch L1 Data-Cache Line for Write
- CLFLUSH—Cache Line Invalidate
- CLWB—Cache Line Writeback

The LFENCE, SFENCE, and MFENCE instructions can be used to force ordering on memory accesses. The order of memory accesses can be important when the reads and writes are to a memory-mapped I/O device, and in multiprocessor environments where memory synchronization is required. LFENCE affects ordering on memory reads, but not writes. SFENCE affects ordering on memory writes, but not reads. MFENCE orders both memory reads and writes. These instructions do not take operands. They are simply inserted between the memory references that are to be ordered. For details about the fence instructions, see “Forcing Memory Order” on page 100.

The PREFETCH*level*, PREFETCH, and PREFETCHW instructions load data from memory into one or more cache levels. PREFETCH*level* loads a memory block into a specified level in the data-cache hierarchy (including a non-temporal caching level). The size of the memory block is implementation dependent. PREFETCH loads a cache line into the L1 data cache. PREFETCHW loads a cache line into the L1 data cache and sets the cache line’s memory-coherency state to *modified*, in anticipation of subsequent data writes to that line. (Both PREFETCH and PREFETCHW are 3DNow!™ instructions.) For details about the prefetch instructions, see “Cache-Control Instructions” on page 105. For a description of MOESI memory-coherency states, see “Memory System” in Volume 2.

The CLFLUSH instruction writes unsaved data back to memory for the specified cache line from all processor caches, invalidates the specified cache, and causes the processor to send a bus cycle which signals external caching devices to write back and invalidate their copies of the cache line. CLFLUSH provides a finer-grained mechanism than the WBINVD instruction, which writes back and invalidates all cache lines. Moreover, CLFLUSH can be used at all privilege levels, unlike WBINVD which can be used only by system software running at privilege level 0.

Similarly, the unprivileged CLWB instruction can be used to force individual modified cache lines to be written to memory without invalidating them in the cache (leaving them in non-modified state), whereas the privileged WBNOINVD instruction operates on entire caches.

3.3.18 No Operation

- NOP—No Operation

The NOP instructions performs no operation (except incrementing the instruction pointer rIP by one). It is an alternative mnemonic for the XCHG rAX, rAX instruction. Depending on the hardware implementation, the NOP instruction may use one or more cycles of processor time.

3.3.19 System Calls

System Call and Return

- SYSENTER—System Call
- SYSEXIT—System Return
- SYSCALL—Fast System Call
- SYSRET—Fast System Return

The SYSENTER and SYSCALL instructions perform a call to a routine running at current privilege level (CPL) 0—for example, a kernel procedure—from a user level program (CPL 3). The addresses of the target procedure and (for SYSENTER) the target stack are specified implicitly through the model-specific registers (MSRs). Control returns from the operating system to the caller when the operating system executes a SYSEXIT or SYSRET instruction. SYSEXIT are SYSRET are privileged instructions and thus can be issued only by a privilege-level-0 procedure.

The SYSENTER and SYSEXIT instructions form a complementary pair, as do SYSCALL and SYSRET. SYSENTER and SYSEXIT are invalid in 64-bit mode. In this case, use the faster SYSCALL and SYSRET instructions.

For details on these on other system-related instructions, see “System-Management Instructions” in Volume 2 and “System Instruction Reference” in Volume 3.

3.3.20 Application-Targeted Accelerator Instructions

- CRC32—Provides hardware acceleration to calculate cyclic redundancy checks for fast and efficient implementation of data integrity protocols.
- POPCNT—Accelerates software performance in the searching of bit patterns. This instruction calculates the number of bits set to 1 in the second operand (source) and returns the count in the first operand (destination register).

3.4 General Rules for Instructions in 64-Bit Mode

This section provides details of the general-purpose instructions in 64-bit mode, and how they differ from the same instructions in legacy and compatibility modes. The differences apply only to general-purpose instructions. Most of them do not apply to SIMD or x87 floating-point instructions.

3.4.1 Address Size

In 64-bit mode, the following rules apply to address size:

- Defaults to 64 bits.
- Can be overridden to 32 bits (by means of opcode prefix 67h).
- Can't be overridden to 16 bits.

3.4.2 Canonical Address Format

Bits 63 through the most-significant implemented virtual-address bit must be all zeros or all ones in any memory reference. See "64-Bit Canonical Addresses" on page 15 for details. (This rule applies to long mode, which includes both 64-bit mode and compatibility mode.)

3.4.3 Branch-Displacement Size

Branch-address displacements are 8 bits or 32 bits, as in legacy mode, but are sign-extended to 64 bits prior to using them for address computations. See "Displacements and Immediates" on page 17 for details.

3.4.4 Operand Size

In 64-bit mode, the following rules apply to operand size:

- **64-Bit Operand Size Option**: If an instruction's operand size (16-bit or 32-bit) in legacy mode depends on the default-size (D) bit in the current code-segment descriptor and the operand-size prefix, then the operand-size choices in 64-bit mode are extended from 16-bit and 32-bit to include 64 bits (with a REX prefix), or the operand size is fixed at 64 bits. See "General-Purpose Instructions in 64-Bit Mode" in Volume 3 for details.
- **Default Operand Size**: The default operand size for most instructions is 32 bits, and a REX prefix must be used to change the operand size to 64 bits. However, two groups of instructions default to 64-bit operand size and do not need a REX prefix: (1) near branches and (2) all instructions, except far branches, that implicitly reference the RSP. See "General-Purpose Instructions in 64-Bit Mode" in Volume 3 for details.
- **Fixed Operand Size**: If an instruction's operand size is fixed in legacy mode, that operand size is usually fixed at the same size in 64-bit mode. (There are some exceptions.) For example, the CPUID instruction always operates on 32-bit operands, irrespective of attempts to override the operand size. See "General-Purpose Instructions in 64-Bit Mode" in Volume 3 for details.
- **Immediate Operand Size**: The maximum size of immediate operands is 32 bits, as in legacy mode, except that 64-bit immediates can be MOVed into 64-bit GPRs. When the operand size is 64 bits, immediates are sign-extended to 64 bits prior to using them. See "Immediate Operand Size" on page 42 for details.
- **Shift-Count and Rotate-Count Operand Size**: When the operand size is 64 bits, shifts and rotates use one additional bit (6 bits total) to specify shift-count or rotate-count, allowing 64-bit shifts and rotates.

3.4.5 High 32 Bits

In 64-bit mode, the following rules apply to extension of results into the high 32 bits when results smaller than 64 bits are written:

- **Zero-Extension of 32-Bit Results**: 32-bit results are zero-extended into the high 32 bits of 64-bit GPR destination registers.
- **No Extension of 8-Bit and 16-Bit Results**: 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit GPR destination registers unchanged.
- **Undefined High 32 Bits After Mode Change**: The processor does not preserve the upper 32 bits of the 64-bit GPRs across changes from 64-bit mode to compatibility or legacy modes. In compatibility and legacy mode, the upper 32 bits of the GPRs are undefined and not accessible to software.

3.4.6 Invalid and Reassigned Instructions

The following general-purpose instructions are invalid in 64-bit mode:

- AAA—ASCII Adjust After Addition
- AAD—ASCII Adjust Before Division
- AAM—ASCII Adjust After Multiply
- AAS—ASCII Adjust After Subtraction
- BOUND—Check Array Bounds
- CALL (far absolute)—Procedure Call Far
- DAA—Decimal Adjust after Addition
- DAS—Decimal Adjust after Subtraction
- INTO—Interrupt to Overflow Vector
- JMP (far absolute)—Jump Far
- POP DS—Pop Stack into DS Segment
- POP ES—Pop Stack into ES Segment
- POP SS—Pop Stack into SS Segment
- POPA, POPAD—Pop All to GPR Words or Doublewords
- PUSH CS—Push CS Segment Selector onto Stack
- PUSH DS—Push DS Segment Selector onto Stack
- PUSH ES—Push ES Segment Selector onto Stack
- PUSH SS—Push SS Segment Selector onto Stack
- PUSHA, PUSHAD—Push All to GPR Words or Doublewords

The following general-purpose instructions are invalid in long mode (64-bit mode and compatibility mode):

- SYSENTER—System Call (use SYSCALL instead)
- SYSEXIT—System Exit (use SYSRET instead)

The opcodes for the following general-purpose instructions are reassigned in 64-bit mode:

- ARPL—Adjust Requestor Privilege Level. Opcode becomes the MOVSXD instruction.
- DEC (one-byte opcode only)—Decrement by 1. Opcode becomes a REX prefix. Use the two-byte DEC opcode instead.
- INC (one-byte opcode only)—Increment by 1. Opcode becomes a REX prefix. Use the two-byte INC opcode instead.
- LDS—Load DS Segment Register
- LES—Load ES Segment Register

3.4.7 Instructions with 64-Bit Default Operand Size

Most instructions default to 32-bit operand size in 64-bit mode. However, the following near branches instructions and instructions that implicitly reference the stack pointer (RSP) default to 64-bit operand size in 64-bit mode:

- *Near Branches:*
  - J*cc*—Jump Conditional Near
  - JMP—Jump Near
  - LOOP—Loop
  - LOOP*cc*—Loop Conditional
- *Instructions That Implicitly Reference RSP:*
  - ENTER—Create Procedure Stack Frame
  - LEAVE—Delete Procedure Stack Frame
  - POP *reg/mem*—Pop Stack (register or memory)
  - POP *reg*—Pop Stack (register)
  - POP FS—Pop Stack into FS Segment Register
  - POP GS—Pop Stack into GS Segment Register
  - POPF, POPFD, POPFQ—Pop to rFLAGS Word, Doubleword, or Quadword
  - PUSH *imm32*—Push onto Stack (sign-extended doubleword)
  - PUSH *imm8*—Push onto Stack (sign-extended byte)
  - PUSH *reg/mem*—Push onto Stack (register or memory)
  - PUSH *reg*—Push onto Stack (register)
  - PUSH FS—Push FS Segment Register onto Stack
  - PUSH GS—Push GS Segment Register onto Stack
  - PUSHF, PUSHFD, PUSHFQ—Push rFLAGS Word, Doubleword, or Quadword onto Stack

The default 64-bit operand size eliminates the need for a REX prefix with these instructions when registers RAX–RSP (the first set of eight GPRs) are used as operands. A REX prefix is still required if R8–R15 (the extended set of eight GPRs) are used as operands, because the prefix is required to address the extended registers.

The 64-bit default operand size can be overridden to 16 bits using the 66h operand-size override. However, it is not possible to override the operand size to 32 bits, because there is no 32-bit operand-size override prefix for 64-bit mode. For details on the operand-size prefix, see "Legacy Instruction Prefixes" in Volume 3.

For details on near branches, see "Near Branches in 64-Bit Mode" on page 90. For details on instructions that implicitly reference RSP, see "Stack Operand-Size in 64-Bit Mode" on page 82.

For details on opcodes and operand-size overrides, see "General-Purpose Instructions in 64-Bit Mode" in Volume 3.

3.5 Instruction Prefixes

An instruction prefix is a byte that precedes an instruction's opcode and modifies the instruction's operation or operands. Instruction prefixes are of three types:

- Legacy Prefixes
- REX Prefixes
- Extended Prefixes

Legacy prefixes are organized into five groups, in which each prefix has a unique value. REX prefixes, which enable use of the AMD64 register extensions in 64-bit mode, are organized as a single group in which the value of the prefix indicates the combination of register-extension features to be enabled. The extended prefixes provide an escape mechanism that opens entirely new instruction encoding spaces for instructions with new capabilities. Currently there are two sets of extended prefixes—VEX and XOP. VEX is used to encode the AVX instructions and XOP is used to encode the XOP instructions.

3.5.1 Legacy Prefixes

Table 3-7 on page 77 shows the legacy prefixes. These are organized into five groups, as shown in the left-most column of the table. Each prefix has a unique hexadecimal value. The legacy prefixes can appear in any order in the instruction, but only one prefix from each of the five groups can be used in a single instruction. The result of using multiple prefixes from a single group is undefined.

There are several restrictions on the use of prefixes. For example, the address-size override prefix (67h) changes the address size used in the read or write access of a single memory operand and applies only to the instruction immediately following the prefix. In general, the operand-size prefix cannot be used with x87 floating-point instructions. When used in the encoding of SSE or 64-bit media instructions, the 66h prefix is repurposed to modify the opcode. The repeat prefixes cause repetition only with certain string instructions. When used in the encoding of SSE or 64-bit media instructions,

the prefixes are repurposed to modify the opcode. The lock prefix can be used with only a small number of general-purpose instructions.

Table 3-7 on page 77 summarizes the functionality of instruction prefixes. Details about the prefixes and their restrictions are given in “Legacy Instruction Prefixes” in Volume 3.

Table 3-7. Legacy Instruction Prefixes

Prefix Group	Mnemonic	Prefix Code (Hex)	Description
Operand-Size Override	none	66[1]	Changes the default operand size of a memory or register operand, as shown in Table 3-3 on page 42.
Address-Size Override	none	67	Changes the default address size of a memory operand, as shown in Table 2-1 on page 18.
Segment Override	CS	2E	Forces use of the CS segment for memory operands.
	DS	3E	Forces use of the DS segment for memory operands.
	ES	26	Forces use of the ES segment for memory operands.
	FS	64	Forces use of the FS segment for memory operands.
	GS	65	Forces use of the GS segment for memory operands.
	SS	36	Forces use of the SS segment for memory operands.
Lock	LOCK	F0	Causes certain read-modify-write instructions on memory to occur atomically.
Repeat	REP	F3[1]	Repeats a string operation (INS, MOVS, OUTS, LODS, and STOS) until the rCX register equals 0.
	REPE or REPZ		Repeats a compare-string or scan-string operation (CMPS*x* and SCAS*x*) until the rCX register equals 0 or the zero flag (ZF) is cleared to 0.
	REPNE or REPNZ	F2[1]	Repeats a compare-string or scan-string operation (CMPS*x* and SCAS*x*) until the rCX register equals 0 or the zero flag (ZF) is set to 1.

Note:
1. *When used with SSE or 64-bit media instructions, this prefix is repurposed to modify the opcode.*

3.5.1.1 Operand-Size and Address-Size Prefixes

The operand-size and address-size prefixes allow mixing of data and address sizes on an instruction-by-instruction basis. An instruction’s default address size can be overridden in any operating mode by using the 67h address-size prefix.

Table 3-3 on page 42 shows the operand-size overrides for all operating modes. In 64-bit mode, the default operand size for most general-purpose instructions is 32 bits. A REX prefix (described in “REX Prefixes” on page 79) specifies a 64-bit operand size, and a 66h prefix specifies a 16-bit operand size. The REX prefix takes precedence over the 66h prefix.

Table 2-1 on page 18 shows the address-size overrides for all operating modes. In 64-bit mode, the default address size is 64 bits. The address size can be overridden to 32 bits. 16-bit addresses are not

supported in 64-bit mode. In compatibility mode, the address-size prefix works the same as in the legacy x86 architecture.

For further details on these prefixes, see “Operand-Size Override Prefix” and “Address-Size Override Prefix” in Volume 3.

3.5.1.2 Segment Override Prefix

The DS segment is the default segment for most memory operands. Many instructions allow this default data segment to be overridden using one of the six segment-override prefixes shown in Table 3-7 on page 77. Data-segment overrides will be ignored when accessing data in the following cases:

- When a stack reference is made that pushes data onto or pops data off of the stack. In those cases, the SS segment is always used.
- When the destination of a string is memory it is always referenced using the ES segment.

Instruction fetches from the CS segment cannot be overridden. However, the CS segment-override prefix can be used to access instructions as data objects and to access data stored in the code segment.

For further details on these prefixes, see “Segment-Override Prefixes” in Volume 3.

3.5.1.3 Lock Prefix

The LOCK prefix causes certain read-modify-write instructions that access memory to occur atomically. The mechanism for doing so is implementation-dependent (for example, the mechanism may involve locking of data-cache lines that contain copies of the referenced memory operands, and/or bus signaling or packet-messaging on the bus). The prefix is intended to give the processor exclusive use of shared memory operands in a multiprocessor system.

The prefix can only be used with forms of the following instructions that write a memory operand: ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, and XOR. An invalid-opcode exception occurs if LOCK is used with any other instruction.

For further details on these prefixes, see “Lock Prefix” in Volume 3.

3.5.1.4 Repeat Prefixes

There are two repeat prefixes byte codes, F3h and F2h. Byte code F3h is the more general and is usually treated as two distinct instructions by assemblers. Byte code F2h is only used with CMPS*x* and SCAS*x* instructions:

- REP (F3h)—This more generalized repeat prefix repeats its associated string instruction the number of times specified in the counter register (rCX). Repetition stops when the value in rCX reaches 0. This prefix is used with the INS, LODS, MOVS, OUTS, and STOS instructions.
- REPE or REPZ (F3h)—This version of REP prefix repeats its associated string instruction the number of times specified in the counter register (rCX). Repetition stops when the value in rCX

reaches 0 or when the zero flag (ZF) is cleared to 0. The prefix can only be used with the CMPS*x* and SCAS*x* instructions.

- REPNE or REPNZ (F2h)—The REPNE or REPNZ prefix repeats its associated string instruction the number of times specified in the counter register (rCX). Repetition stops when the value in rCX reaches 0 or when the zero flag (ZF) is set to 1. The prefix can only be used with the CMPS*x* and SCAS*x* instructions.

The size of the rCX counter is determined by the effective *address size*. For further details about these prefixes, including optimization of their use, see "Repeat Prefixes" in Volume 3.

3.5.2 REX Prefixes

REX prefixes can be used only in 64-bit mode. They enable the 64-bit register extensions. REX prefixes specify the following features:

- Use of an extended GPR register, shown in Figure 3-3 on page 27.
- Use of an extended YMM/XMM register, shown in Figure 4-1 on page 114.
- Use of a 64-bit (quadword) operand size, as described in "Operands" on page 36.
- Use of extended control and debug registers, as described in Volume 2.

REX prefix bytes have a value in the range 40h to 4Fh, depending on the particular combination of register extensions desired. With few exceptions, a REX prefix is required in order to access a 64-bit GPR or one of the extended GPR or XMM registers. A few instructions (described in "General-Purpose Instructions in 64-Bit Mode" in Volume 3) default to 64-bit operand size and do not need the REX prefix to access an extended 64-bit GPR.

Only one REX prefix is needed to express the full selection of 64-bit-mode register extension features. When used, the REX prefix must immediately precede the opcode byte of an instruction, or opcode map escape prefix if present. Any other placement of a REX prefix is ignored.

For further details on the REX prefixes, see "REX Prefixes" in Volume 3.

3.5.3 VEX and XOP Prefixes

The VEX and XOP prefixes extend instruction encoding and operand specification capabilities beyond those of the REX prefixes. They allow the encoding of new instructions and the specification of three, four, or five operands. The VEX prefixes are C4h and C5h and the XOP prefix is 8Eh.

For further details on the VEX and XOP prefixes, see "Encoding Using the VEX and XOP Prefixes" in Volume 3.

3.6 Feature Detection

The CPUID instruction provides information about the processor implementation and its capabilities. Software operating at any privilege level can execute the CPUID instruction to collect this

information. After the information is collected, software can select procedures that optimize performance for a particular hardware implementation.

Support for the CPUID instruction is implementation-dependent, as determined by software's ability to write the RFLAGS.ID bit. After software has determined that the processor implementation supports the CPUID instruction, software can test for support for a specific feature by loading the appropriate function number into the EAX register and executing the CPUID instruction. Processor feature information is returned in the EAX, EBX, ECX, and EDX registers.

See "CPUID" in the *AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions*, order# 24594, for a full description of the CPUID instruction. See Appendix D of Volume 3 for a description of processor feature flags associated with instruction support and Appendix E for an exhaustive list of all processor information accessible via the CPUID instruction.

3.6.1 Feature Detection in a Virtualized Environment

Software writers must assume that their software may be executed as a guest in a virtualized environment. A virtualized guest may be migrated between processors of differing capabilities, so the CPUID indication of a feature's presence must be respected. Operating systems, user programs and libraries must all ensure that the CPUID instruction indicates a feature is present before using that feature. The hypervisor is responsible for ensuring consistent CPUID values across the system.

For example, an OS, program, or library typically detects a feature during initialization and then configures code paths or internal copies of feature indications based on the detection of that feature, with the feature detection occurring once per initialization. In this case, the feature must be detected by use of the CPUID instruction rather than by ignoring CPUID and testing for the presence of that feature.

To ensure guest migration between processors across multiple generations of processors, while allowing for features to be deprecated in future generations of processors, it is imperative that software check the CPUID bit once per program or library initialization before using instructions that are indicated by a CPUID bit; otherwise inconsistent behavior may result.

3.7 Control Transfers

3.7.1 Overview

From the application-program's viewpoint, program-control flow is sequential—that is, instructions are addressed and executed sequentially—except when a branch instruction (a call, return, jump, interrupt, or return from interrupt) is encountered, in which case program flow changes to the branch instruction's target address. Branches are used to iterate through loops and move through conditional program logic. Branches cause a new instruction pointer to be loaded into the rIP register, and sometimes cause the CS register to point to a different code segment. The CS:rIP values can be specified as part of a branch instruction, or they can be read from a register or memory.

Branches can also be used to transfer control to another program or procedure running at a different privilege level. In such cases, the processor automatically checks the source program and target program privileges to ensure that the transfer is allowed before loading CS:rIP with the new values.

3.7.2 Privilege Levels

The processor's *protected* modes include legacy protected mode and long mode (both compatibility mode and 64-bit mode). In all protected modes and virtual x86 mode, privilege levels are used to isolate and protect programs and data from each other. The privilege levels are designated with a numerical value from 0 to 3, with 0 being the most privileged and 3 being the least privileged. Privilege 0 is normally reserved for critical system-software components that require direct access to, and control over, all processor and system resources. Privilege 3 is used by application software. The intermediate privilege levels (1 and 2) are used, for example, by device drivers and library routines that access and control a limited set of processor and system resources.

Figure 3-9 shows the relationship of the four privilege-levels to each other. The protection scheme is implemented using the segmented memory-management mechanism described in "Segmented Virtual Memory" in Volume 2.

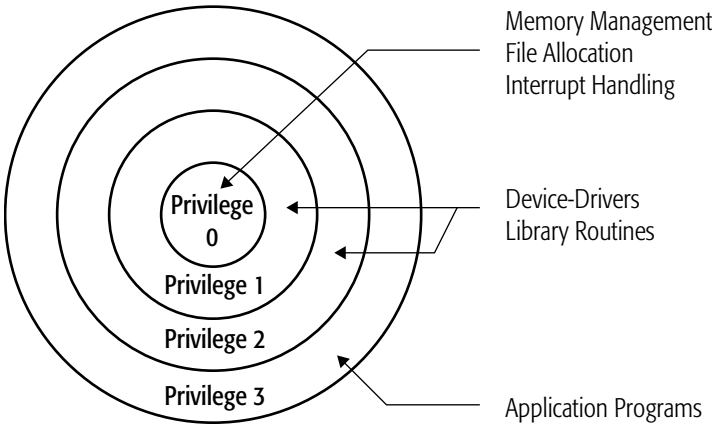


Figure 3-9. Privilege-Level Relationships

3.7.3 Procedure Stack

A procedure stack (also known as 'program stack') is often used by control transfer operations, particularly those that change privilege levels. Information from the calling program is passed to the target program on the procedure stack. CALL instructions, interrupts, and exceptions all push information onto the procedure stack. The pushed information includes a return pointer to the calling program and, for call instructions, optionally includes parameters. When a privilege-level change occurs, the calling program's stack pointer (the pointer to the top of the stack) is pushed onto the stack. Interrupts and exceptions also push a copy of the calling program's rFLAGs register and, in some cases, an error code associated with the interrupt or exception.

The RET or IRET control-transfer instructions reverse the operation of CALLs, interrupts, and exceptions. These return instructions pop the return pointer off the stack and transfer control back to the calling program. If the calling program's stack pointer was pushed, it is restored by popping the saved values off the stack and into the SS and rSP registers.

3.7.3.1 Stack Alignment

Control-transfer performance can degrade significantly when the stack pointer is not aligned properly. Stack pointers should be word aligned in 16-bit segments, doubleword aligned in 32-bit segments, and quadword aligned in 64-bit mode.

3.7.3.2 Stack Operand-Size in 64-Bit Mode

In 64-bit mode, the stack pointer size is always 64 bits. The stack size is not controlled by the default-size (B) bit in the SS descriptor, as it is in compatibility and legacy modes, nor can it be overridden by an instruction prefix. Address-size overrides are ignored for implicit stack references.

Except for far branches, all instructions that implicitly reference the stack pointer default to 64-bit operand size in 64-bit mode. Table 3-8 on page 83 lists these instructions.

The default 64-bit operand size eliminates the need for a REX prefix with these instructions. However, a REX prefix is still required if R8–R15 (the extended set of eight GPRs) are used as operands, because the prefix is required to address the extended registers. Pushes and pops of 32-bit stack values are not possible in 64-bit mode with these instructions, because there is no 32-bit operand-size override prefix for 64-bit mode.

3.7.4 Jumps

Jump instructions provide a simple means for transferring program control from one location to another. Jumps do not affect the procedure stack, and return instructions cannot transfer control back to the instruction following a jump. Two general types of jump instruction are available: unconditional (JMP) and conditional (J*cc*).

There are two types of unconditional jumps (JMP):

- *Near Jumps*—When the target address is within the current code segment.
- *Far Jumps*—When the target address is outside the current code segment.

Although unconditional jumps can be used to change code segments, they cannot be used to change privilege levels.

Conditional jumps (J*cc*) test the state of various bits in the rFLAGS register (or rCX) and jump to a target location based on the results of that test. Only near forms of conditional jumps are available, so J*cc* cannot be used to transfer control to another code segment.

Table 3-8. Instructions that Implicitly Reference RSP in 64-Bit Mode

Mnemonic	Opcode (hex)	Description	Operand Size (bits)	
			Default	Possible Overrides[1]
CALL	E8, FF /2	Call Procedure Near	64	16
ENTER	C8	Create Procedure Stack Frame		
LEAVE	C9	Delete Procedure Stack Frame		
POP reg/mem	8F /0	Pop Stack (register or memory)		
POP reg	58 to 5F	Pop Stack (register)		
POP FS	0F A1	Pop Stack into FS Segment Register		
POP GS	0F A9	Pop Stack into GS Segment Register		
POPF POPFQ	9D	Pop to EFLAGS Word or Quadword		
PUSH imm32	68	Push onto Stack (sign-extended doubleword)		
PUSH imm8	6A	Push onto Stack (sign-extended byte)		
PUSH reg/mem	FF /6	Push onto Stack (register or memory)		
PUSH reg	50–57	Push onto Stack (register)		
PUSH FS	0F A0	Push FS Segment Register onto Stack		
PUSH GS	0F A8	Push GS Segment Register onto Stack		
PUSHF PUSHFQ	9C	Push rFLAGS Word or Quadword onto Stack		
RET	C2, C3	Return From Call (near)		

Note:
1. *There is no 32-bit operand-size override prefix in 64-bit mode.*

3.7.5 Procedure Calls

The CALL instruction transfers control unconditionally to a new address, but unlike jump instructions, it saves a return pointer (CS:rIP) on the stack. The called procedure can use the RET instruction to pop the return pointers to the calling procedure from the stack and continue execution with the instruction following the CALL.

There are four types of CALL:

- *Near Call*—When the target address is within the current code segment.
- *Far Call*—When the target address is outside the current code segment.
- *Interprivilege-Level Far Call*—A far call that changes privilege level.
- *Task Switch*—A call to a target address in another task.

3.7.5.1 Near Call

When a near CALL is executed, only the calling procedure's rIP (the return offset) is pushed onto the stack. After the rIP is pushed, control is transferred to the new rIP value specified by the CALL instruction. Parameters can be pushed onto the stack by the calling procedure prior to executing the CALL instruction. Figure 3-10 shows the stack pointer before (old rSP value) and after (new rSP value) the CALL. The stack segment (SS) is not changed.

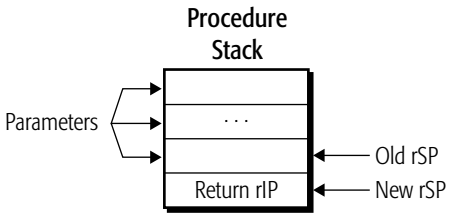


Figure 3-10. Procedure Stack, Near Call

When shadow stacks are enabled at the current privilege level, a near CALL pushes the calling procedure's LIP (CS base + rIP) onto the shadow stack, in addition to pushing the rIP onto the procedure stack.

3.7.5.2 Far Call, Same Privilege

A far CALL changes the code segment, so the full return pointer (CS:rIP) is pushed onto the stack. After the return pointer is pushed, control is transferred to the new CS:rIP value specified by the CALL instruction. Parameters can be pushed onto the stack by the calling procedure prior to executing the CALL instruction. Figure 3-11 shows the stack pointer before (old rSP value) and after (new rSP value) the CALL. The stack segment (SS) is not changed.

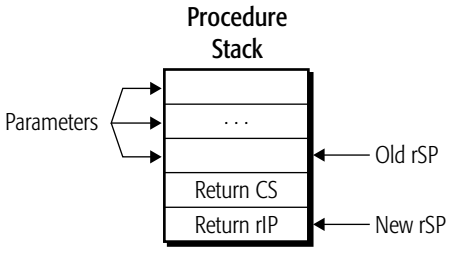


Figure 3-11. Procedure Stack, Far Call to Same Privilege

If the shadow stack feature is enabled at the current privilege level, a far CALL to the same privilege pushes the calling procedure's CS and LIP (CS.base + rIP) onto the shadow stack, in addition to

pushing the CS:rIP onto the procedure stack. For a detailed description of shadow stack operations, see "Shadow Stacks" in Volume 2.

3.7.5.3 Far Call, Greater Privilege

A far CALL to a more-privileged procedure performs a stack switch prior to transferring control to the called procedure. Switching stacks isolates the more-privileged procedure's stack from the less-privileged procedure's stack, and it provides a mechanism for saving the return pointer back to the procedure that initiated the call.

Calls to more-privileged software can only take place through a system descriptor called a *call-gate descriptor*. Call-gate descriptors are created and maintained by system software. In 64-bit mode, only indirect far calls (those whose target memory address is in a register or other memory location) are supported. Absolute far calls (those that reference the base of the code segment) are not supported in 64-bit mode.

When a call to a more-privileged procedure occurs, the processor locates the new procedure's stack pointer from its task-state segment (TSS). The old stack pointer (SS:rSP) is pushed onto the new stack, and (in legacy mode only) any parameters specified by the count field in the call-gate descriptor are copied from the old stack to the new stack (long mode does not support this automatic parameter copying). The return pointer (CS:rIP) is then pushed, and control is transferred to the new procedure. Figure 3-12 shows an example of a stack switch resulting from a call to a more-privileged procedure. "Segmented Virtual Memory" in Volume 2 provides additional information on privilege-changing CALLs.

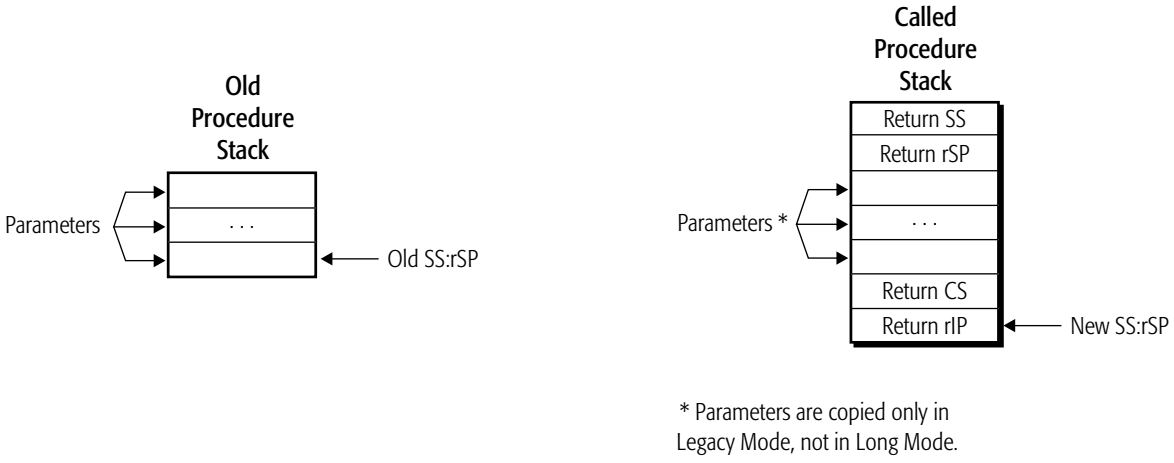


Figure 3-12. Procedure Stack, Far Call to Greater Privilege

If the shadow stack feature is enabled at the target CPL, a far call to a more-privileged level also switches to a new shadow stack. Depending on the target CPL, the old SSP and the CS and LIP may be pushed onto the new shadow stack.

If starting at CPL=3:

- the current SSP is saved to MSR PL3_SSP.
- a new SSP is loaded from PLn_SSP MSR (where n = the target CPL 0, 1 or 2).
- the CS and LIP are not pushed onto the new shadow stack.

If starting at CPL =1 or 2:

- the new SSP is loaded from MSR PLn_SSP, (where n = the target CPL 0 or 1).
- the CS, LIP and old SSP are pushed onto the new shadow stack.

For a detailed description of shadow stack operations, see “Shadow Stacks” in Volume 2.

3.7.5.4 Task Switch

In legacy mode, when a call to a new task occurs, the processor suspends the currently-executing task and stores the processor-state information at the point of suspension in the current task’s task-state segment (TSS). The new task’s state information is loaded from its TSS, and the processor resumes execution within the new task.

In long mode, hardware task switching is disabled. Task switching is fully described in “Segmented Virtual Memory” in Volume 2.

3.7.6 Returning from Procedures

The RET instruction reverses the effect of a CALL instruction. The return address is popped off the procedure stack, transferring control unconditionally back to the calling procedure at the instruction following the CALL. A return that changes privilege levels also switches stacks.

The three types of RET are:

- *Near Return*—Transfers control back to the calling procedure within the current code segment.
- *Far Return*—Transfers control back to the calling procedure outside the current code segment.
- *Interprivilege-Level Far Return*—A far return that changes privilege levels.

All of the RET instruction types can be used with an immediate operand indicating the number of parameter bytes present on the stack. These parameters are *released* from the stack—that is, the stack pointer is adjusted by the value of the immediate operand—but the parameter bytes are not actually popped off of the stack (i.e., read into a register or memory location).

3.7.6.1 Near Return

When a near RET is executed, the calling procedure's return offset is popped off of the stack and into the rIP register. Execution begins from the newly-loaded offset. If an immediate operand is included with the RET instruction, the stack pointer is adjusted by the number of bytes indicated. Figure 3-13 shows the stack pointer before (old rSP value) and after (new rSP value) the RET. The stack segment (SS) is not changed.

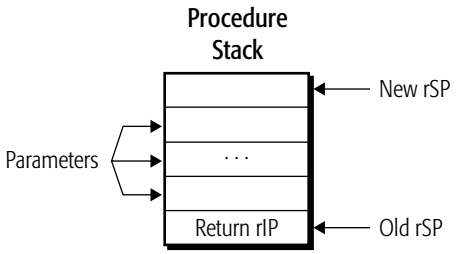


Figure 3-13. Procedure Stack, Near Return

If the shadow stack feature is enabled at the current CPL, a near RET pops the return LIP from the shadow stack and compares it to the return address read from the procedure stack. If the comparison fails, a control-protection (#CP fault) is generated. For a detailed description of shadow stack operations, see "Shadow Stacks" in Volume 2.

3.7.6.2 Far Return, Same Privilege

A far RET changes the code segment, so the full return pointer is popped off the stack and into the CS and rIP registers. Execution begins from the newly-loaded segment and offset. If an immediate operand is included with the RET instruction, the stack pointer is adjusted by the number of bytes indicated. Figure 3-14 on page 87 shows the stack pointer before (old rSP value) and after (new rSP value) the RET. The stack segment (SS) is not changed.

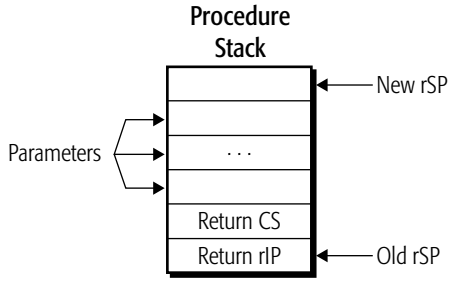


Figure 3-14. Procedure Stack, Far Return from Same Privilege

If the shadow stacks feature is enabled at the current CPL, a far RET to the same CPL pops the old SSP and the return LIP from the shadow stack. The return address is compared with the return address read from the procedure stack. If the comparison fails, a control-protection (#CP fault) is generated. For a detailed description of shadow stack operations, see “Shadow Stacks” in Volume 2.

3.7.6.3 Far Return, Less Privilege

Privilege-changing far RETs can only return to less-privileged code segments, otherwise a general-protection exception occurs. The full return pointer is popped off the stack and into the CS and rIP registers, and execution begins from the newly-loaded segment and offset. A far RET that changes privilege levels also switches stacks. The return procedure’s stack pointer is popped off the stack and into the SS and rSP registers. If an immediate operand is included with the RET instruction, the newly-loaded stack pointer is adjusted by the number of bytes indicated. Figure 3-15 shows the stack pointer before (old SS:rSP value) and after (new SS:rSP value) the RET. “Segmented Virtual Memory” in Volume 2 provides additional information on privilege-changing RETs.

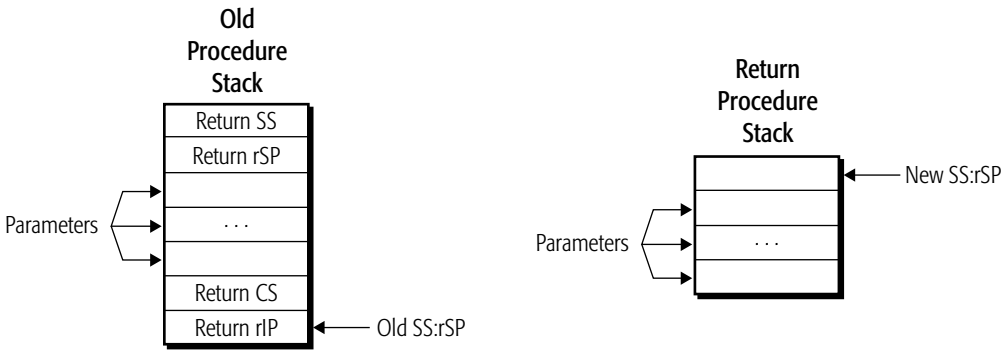


Figure 3-15. Procedure Stack, Far Return from Less Privilege

If the shadow stack feature is enabled, the operation of the shadow stack for a far return from a less-privileged level depends on the target CPL.

If returning to CPL 3:

- the old SSP is restored from PL3_SSP.
- the return address is not checked against the shadow stack.

If returning to CPL 1 or 2:

- the old SSP is popped and restored from the current shadow stack.

- the return CS and LIP are popped from the current shadow stack and compared with the return address read from the procedure stack. If the comparison fails, a control-protection (#CP fault) is generated.

For a detailed description of shadow stack operations, see “Shadow Stacks” in Volume 2.

3.7.7 System Calls

A disadvantage of far CALLs and far RETs is that they use segment-based protection and privilege-checking. This involves significant overhead associated with loading new segment selectors and their corresponding descriptors into the segment registers. The overhead includes not only the time required to load the descriptors from memory but also the time required to perform the privilege, type, and limit checks. Privilege-changing CALLs to the operating system are slowed further by the control transfer through a gate descriptor.

3.7.7.1 SYSCALL and SYSRET

SYSCALL and SYSRET are low-latency system-call and system-return control-transfer instructions. They can be used in protected mode. These instructions eliminate segment-based privilege checking by using pre-determined target and return code segments and stack segments. The operating system sets up and maintains the predetermined segments using special registers within the processor, so the segment descriptors do not need to be fetched from memory when the instructions are used. The simplifications made to privilege checking allow SYSCALL and SYSRET to complete in far fewer processor clock cycles than CALL and RET.

SYSRET can only be used to return from CPL = 0 procedures and is not available to application software. SYSCALL can be used by applications to call operating system service routines running at CPL = 0. The SYSCALL instruction does not take operands. Linkage conventions are initialized and maintained by the operating system. “System-Management Instructions” in Volume 2 contains detailed information on the operation of SYSCALL and SYSRET.

Because SYSCALL and SYSRET do not use the program stack for return address linkage, the shadow stack mechanism is not used to validate their return addresses. However, when the shadow stacks feature is enabled, SYSCALL and SYSRET save and restore the current SSP. See "SYSCALL and SYSRET" section 6.1.1, Volume 2 for more information.

3.7.7.2 SYSENTER and SYSEXIT

The SYSENTER and SYSEXIT instructions provide similar capabilities to SYSCALL and SYSRET. However, these instructions can be used only in legacy mode and are not supported in long mode. SYSCALL and SYSRET are the preferred instructions for calling privileged software. See “System-Management Instructions” in Volume 2 for further information on SYSENTER and SYSEXIT.

3.7.8 General Considerations for Branching

Branching causes delays which are a function of the hardware-implementation's branch-prediction capabilities. Sequential flow avoids the delays caused by branching but is still exposed to delays caused by cache misses, memory bus bandwidth, and other factors.

In general, branching code should be replaced with sequential code whenever practical. This is especially important if the branch body is small (resulting in frequent branching) and when branches depend on random data (resulting in frequent mispredictions of the branch target). In certain hardware implementations, far branches (as opposed to near branches) may not be predictable by the hardware, and recursive functions (those that call themselves) may overflow a return-address stack.

All calls and returns should be paired for optimal performance. Hardware implementations that include a return-address stack can lose stack synchronization if calls and returns are not paired.

3.7.9 Branching in 64-Bit Mode

3.7.9.1 Near Branches in 64-Bit Mode

The long-mode architecture expands the near-branch mechanisms to accommodate branches in the full 64-bit virtual-address space. In 64-bit mode, the operand size for all near branches defaults to 64 bits, so these instructions update the full 64-bit RIP.

Table 3-9 lists the near-branch instructions.

Table 3-9. Near Branches in 64-Bit Mode

Mnemonic	Opcode (hex)	Description	Operand Size (bits)	
			Default	Possible Overrides[1]
CALL	E8, FF /2	Call Procedure Near	64	16
J*cc*	70 to 7F, 0F 80 to 0F 8F	Jump Conditional		
JCXZ JECXZ JRCXZ	E3	Jump on CX/ECX/RCX Zero		
JMP	EB, E9, FF /4	Jump Near		
LOOP	E2	Loop		
LOOP*cc*	E0, E1	Loop if Zero/Equal or Not Zero/Equal		
RET	C2, C3	Return From Call (near)		

Note:
1. *There is no 32-bit operand-size override prefix in 64-bit mode.*

The default 64-bit operand size eliminates the need for a REX prefix with these instructions when registers RAX–RSP (the first set of eight GPRs) are used as operands. A REX prefix is still required if

R8–R15 (the extended set of eight GPRs) are used as operands, because the prefix is required to address the extended registers.

The following aspects of near branches are controlled by the effective operand size:

- Truncation of the instruction pointer.
- Size of a stack pop or push, resulting from a CALL or RET.
- Size of a stack-pointer increment or decrement, resulting from a CALL or RET.
- Indirect-branch operand size.

In 64-bit mode, all of the above actions are forced to 64 bits. However, the size of the displacement field for relative branches is still limited to 32 bits.

The operand size of near branches is fixed at 64 bits without the need for a REX prefix. However, the address size of near branches is not forced in 64-bit mode. Such addresses are 64 bits by default, but they can be overridden to 32 bits by a prefix.

3.7.9.2 Branches to 64-Bit Offsets

Because immediates are generally limited to 32 bits, the only way a full 64-bit absolute RIP can be specified in 64-bit mode is with an indirect branch. For this reason, direct forms of far branches are invalid in 64-bit mode.

3.7.10 Interrupts and Exceptions

Interrupts and exceptions are a form of control transfer operation. They are used to call special system-service routines, called interrupt handlers, which are designed to respond to the interrupt or exception condition. Pointers to the interrupt handlers are stored by the operating system in an *interrupt-descriptor table*, or IDT. In legacy real mode, the IDT contains an array of 4-byte far pointers to interrupt handlers. In legacy protected mode, the IDT contains an array of 8-byte gate descriptors. In long mode, the gate descriptors are 16 bytes. Interrupt gates, task gates, and trap gates can be stored in the IDT, but not call gates.

Interrupt handlers are usually privileged software because they typically require access to restricted system resources. System software is responsible for creating the interrupt gates and storing them in the IDT. “Exceptions and Interrupts” in Volume 2 contains detailed information on the interrupt mechanism and the requirements on system software for managing the mechanism.

The IDT is indexed using the interrupt number, or *vector*. How the vector is specified depends on the source, as described below. The first 32 of the available 256 interrupt vectors are reserved for internal use by the processor—for exceptions (as described below) and other purposes.

Interrupts are caused either by software or hardware. The INT, INT3, and INTO instructions implement a *software interrupt* by calling an interrupt handler directly. These are general-purpose (privilege-level-3) instructions. The operand of the INT instruction is an immediate byte value specifying the interrupt vector used to index the IDT. INT3 and INTO are specific forms of software interrupts used to call interrupt 3 and interrupt 4, respectively. *External interrupts* are produced by

system logic which passes the IDT index to the processor via input signals. External interrupts can be either *maskable* or *non-maskable*.

Exceptions usually occur as a result of software execution errors or other internal-processor errors. Exceptions can also occur in non-error situations, such as debug-program single-stepping or address-breakpoint detection. In the case of exceptions, the processor produces the IDT index based on the detected condition. The handlers for interrupts and exceptions are identical for a given vector.

The processor's response to an exception depends on the type of the exception. For all exceptions except SSE and x87 floating-point exceptions, control automatically transfers to the handler (or service routine) for that exception, as defined by the exceptions vector. For 128-bit-media and x87 floating-point exceptions, there is both a masked and unmasked response. When unmasked, these exceptions invoke their exception handler. When masked, a default masked response is provided instead of invoking the exception handler.

Exceptions and software-initiated interrupts occur synchronously with respect to the processor clock. There are three types of exceptions:

- *Faults*—A fault is a precise exception that is reported on the boundary before the interrupted instruction. Generally, faults are caused by an undesirable error condition involving the interrupted instruction, although some faults (such as page faults) are common and normal occurrences. After the service routine completes, the machine state prior to the faulting instruction is restored, and the instruction is retried.
- *Traps*—A trap is a precise exception that is reported on the boundary following the interrupted instruction. The instruction causing the exception finishes before the service routine is invoked. Software interrupts and certain breakpoint exceptions used in debugging are traps.
- *Aborts*—Aborts are imprecise exceptions. The instruction causing the exception, and possibly an indeterminate additional number of instructions, complete execution before the service routine is invoked. Because they are imprecise, aborts typically do not allow reliable program restart.

Table 3-10 shows the interrupts and exceptions that can occur, together with their vector numbers, mnemonics, source, and causes. For a detailed description of interrupts and exceptions, see "Exceptions and Interrupts" in Volume 2.

Control transfers to interrupt handlers are similar to far calls, except that for the former, the rFLAGS register is pushed onto the stack before the return address. Interrupts and exceptions to several of the first 32 interrupts can also push an error code onto the stack. No parameters are passed by an interrupt. As with CALLs, interrupts that cause a privilege change also perform a stack switch.

Table 3-10. Interrupts and Exceptions

Vector	Interrupt (Exception)	Mnemonic	Source	Cause	Generated By General-Purpose Instructions
0	Divide-By-Zero-Error	#DE	Software	DIV, IDIV instructions	yes
1	Debug	#DB	Internal	Instruction accesses and data accesses	yes
2	Non-Maskable-Interrupt	NMI	External	External $\overline{\text{NMI}}$ signal	no
3	Breakpoint	#BP	Software	INT3 instruction	yes
4	Overflow	#OF	Software	INTO instruction	yes
5	Bound-Range	#BR	Software	BOUND instruction	yes
6	Invalid-Opcode	#UD	Internal	Invalid instructions	yes
7	Device-Not-Available	#NM	Internal	x87 instructions	no
8	Double-Fault	#DF	Internal	Exception during an interrupt/exception transfer	indirectly
9	Coprocessor-Segment-Overrun	—	External	Unsupported (reserved)	
10	Invalid-TSS	#TS	Internal	Task-state segment access and task switch	yes
11	Segment-Not-Present	#NP	Internal	Segment access through a descriptor	yes
12	Stack	#SS	Internal	SS register loads and stack references	yes
13	General-Protection	#GP	Internal	Memory accesses and protection checks	yes
14	Page-Fault	#PF	Internal	Memory accesses when paging enabled	yes
15	Reserved	—			
16	x87 Floating-Point Exception-Pending	#MF	Software	x87 floating-point and 64-bit media floating-point instructions	no
17	Alignment-Check	#AC	Internal	Memory accesses	yes
18	Machine-Check	#MC	Internal External	Model specific	yes
19	SIMD Floating-Point	#XF	Internal	128-bit media floating-point instructions	no
20	Reserved	—			
21	Control-Protection	#CP	Internal	Control transfers	yes
22—31	Reserved (Internal and External)	—			

Table 3-10. Interrupts and Exceptions (continued)

Vector	Interrupt (Exception)	Mnemonic	Source	Cause	Generated By General-Purpose Instructions
30	Security	#SX	External	Security exception	no
31	Reserved	—			
0–255	External Interrupts (Maskable)	—	External	External interrupt signaling	no
0–255	Software Interrupts	—	Software	INT instruction	yes

3.7.10.1 Interrupt to Same Privilege in Legacy Mode

When an interrupt to a handler running at the same privilege occurs, the processor pushes a copy of the rFLAGS register, followed by the return pointer (CS:rIP), onto the stack. If the interrupt generates an error code, it is pushed onto the stack as the last item. Control is then transferred to the interrupt handler. Figure 3-16 on page 94 shows the stack pointer before (old rSP value) and after (new rSP value) the interrupt. The stack segment (SS) is not changed.

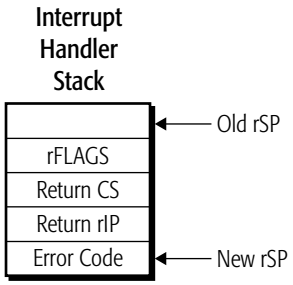


Figure 3-16. Procedure Stack, Interrupt to Same Privilege

If the shadow stack feature is enabled for the current CPL, an interrupt to a handler at the same privilege pushes the interrupted procedure's CS and LIP (linear return IP) onto the shadow stack, in addition to pushing the CS:rIP onto the procedure stack stack. For a detailed description of shadow stack operations, see "Shadow Stacks" in Volume 2.

3.7.10.2 Interrupt to More Privilege or in Long Mode

When an interrupt to a more-privileged handler occurs or the processor is operating in long mode the processor locates the handler's stack pointer from the TSS. The old stack pointer (SS:rSP) is pushed onto the new stack, along with a copy of the rFLAGS register. The return pointer (CS:rIP) to the interrupted program is then copied to the stack. If the interrupt generates an error code, it is pushed onto the stack as the last item. Control is then transferred to the interrupt handler. Figure 3-17 shows an example of a stack switch resulting from an interrupt with a change in privilege.

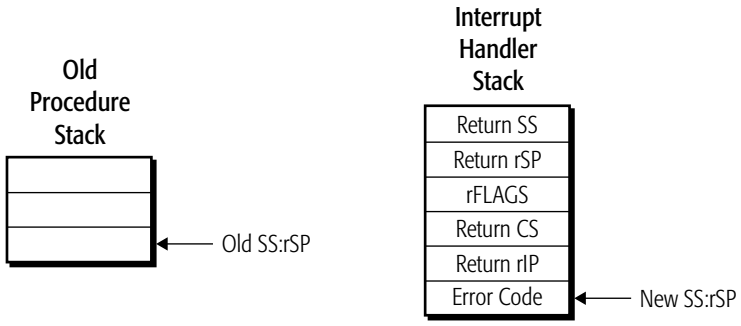


Figure 3-17. Procedure Stack, Interrupt to Higher Privilege

If the shadow stack feature is enabled at the target CPL, an interrupt to a more-privileged level also switches to a new shadow stack. Depending on the CPL of the interrupt procedure, the old SSP and the CS and LIP may be pushed onto the new shadow stack. For a detailed description of shadow stack operations, see "Shadow Stacks" in Volume 2.

3.7.10.3 Interrupt Returns

The IRET, IRETD, and IRETQ instructions are used to return from an interrupt handler. Prior to executing an IRET, the interrupt handler must pop the error code off of the stack if one was pushed by the interrupt or exception. IRET restores the interrupted program's rIP, CS, and rFLAGS by popping their saved values off of the stack and into their respective registers. If a privilege change occurs or IRET is executed in 64-bit mode, the interrupted program's stack pointer (SS:rSP) is also popped off of the stack. Control is then transferred back to the interrupted program.

If the shadow stack feature is enabled at the current CPL, an IRET to the same CPL pops the old SSP, return CS and return LIP from the shadow stack. The CS and return address are compared with the values read from the procedure stack. If the comparison fails, a control-protection (#CP) fault is generated. If IRET is returning to a different CPL, the operation of the shadow stack depends on the CPL of the procedure to which IRET is returning. If the return CPL is 1 or 2, the old SSP, return CS and return LIP are popped from the shadow stack. The CS and return address are compared with the values read from the procedure stack. If the comparison fails, a control-protection (#CP) fault is generated. If the return CPL is 3, the old SSP is restored from PL3_SSP and the return address is not checked against the shadow stack. For a detailed description of shadow stack operations, see "Shadow Stacks" in Volume 2.

3.8 Input/Output

I/O devices allow the processor to communicate with the outside world, usually to a human or to another system. In fact, a system without I/O has little utility. Typical I/O devices include a keyboard, mouse, LAN connection, printer, storage devices, and monitor. The speeds these devices must operate

at vary greatly, and usually depend on whether the communication is to a human (slow) or to another machine (fast). There are exceptions. For example, humans can consume graphics data at very high rates.

There are two methods for communicating with I/O devices in AMD64 processor implementations. One method involves accessing I/O through ports located in I/O-address space (“I/O Addressing” on page 96), and the other method involves accessing I/O devices located in the memory-address space (“Memory Organization” on page 9). The address spaces are separate and independent of each other.

I/O-address space was originally introduced as an optimized means for accessing I/O-device control ports. Then, systems usually had few I/O devices, devices tended to be relatively low-speed, device accesses needed to be strongly ordered to guarantee proper operation, and device protection requirements were minimal or non-existent. Memory-mapped I/O has largely supplanted I/O-address space access as the preferred means for modern operating systems to interface with I/O devices. Memory-mapped I/O offers greater flexibility in protection, vastly more I/O ports, higher speeds, and strong or weak ordering to suit the device requirements.

3.8.1 I/O Addressing

Access to I/O-address space is provided by the IN and OUT instructions, and the string variants of these instructions, INS and OUTS. The operation of these instructions are described in “Input/Output” on page 68. Although not required, processor implementations generally transmit I/O-port addresses and I/O data over the same external signals used for memory addressing and memory data. Different bus-cycles generated by the processor differentiate I/O-address space accesses from memory-address space accesses.

3.8.1.1 I/O-Address Space

Figure 3-18 on page 96 shows the 64 Kbyte I/O-address space. I/O ports can be addressed as bytes, words, or doublewords. As with memory addressing, word-I/O and doubleword-I/O ports are simply two or four consecutively-addressed byte-I/O ports. Word and doubleword I/O ports can be aligned on any byte boundary, but there is typically a performance penalty for unaligned accesses. Performance is optimized by aligning word-I/O ports on word boundaries, and doubleword-I/O ports on doubleword boundaries.

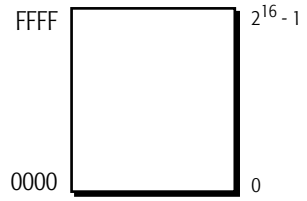


Figure 3-18. I/O Address Space

3.8.1.2 Memory-Mapped I/O

Memory-mapped I/O devices are attached to the system memory bus and respond to memory transactions as if they were memory devices, such as DRAM. Access to memory-mapped I/O devices can be performed using any instruction that accesses memory, but typically MOV instructions are used to transfer data between the processor and the device. Some I/O devices may have restrictions on read-modify-write accesses.

Any location in memory can be used as a memory-mapped I/O address. System software can use the paging facilities to virtualize memory devices and protect them from unauthorized access. See "System-Management Instructions" in Volume 2 for a discussion of memory virtualization and paging.

3.8.2 I/O Ordering

The order of read and write accesses between the processor and an I/O device is usually important for properly controlling device operation. Accesses to I/O-address space and memory-address space differ in the default ordering enforced by the processor and the ability of software to control ordering.

3.8.2.1 I/O-Address Space

The processor always orders I/O-address space operations strongly, with respect to other I/O and memory operations. Software cannot modify the I/O ordering enforced by the processor. IN instructions are not executed until all previous writes to I/O space and memory have completed. OUT instructions delay execution of the *following* instruction until all writes—including the write performed by the OUT—have completed. Unlike memory writes, writes to I/O addresses are never buffered by the processor.

The processor can use more than one bus transaction to access an unaligned, multi-byte I/O port. Unaligned accesses to I/O-address space do not have a defined bus transaction ordering, and that ordering can change from one implementation to another. If the use of an unaligned I/O port is required, and the order of bus transactions to that port is important, software should decompose the access into multiple, smaller aligned accesses.

3.8.2.2 Memory-Mapped I/O

To maximize software performance, processor implementations can execute instructions out of program order. This can cause the sequence of memory accesses to also be out of program order, called *weakly ordered*. As described in "Accessing Memory" on page 99, the processor can perform memory reads in any order, it can perform reads without knowing whether it requires the result (speculation), and it can reorder reads ahead of writes. In the case of writes, multiple writes to memory locations in close proximity to each other can be combined into a single write or a burst of multiple writes. Writes can also be delayed, or buffered, by the processor.

Application software that needs to force memory ordering to memory-mapped I/O devices can do so using the read/write barrier instructions: LFENCE, SFENCE, and MFENCE. These instructions are described in "Forcing Memory Order" on page 100. Serializing instructions, I/O instructions, and

locked instructions can also be used as read/write barriers, but they modify program state and are an inferior method for enforcing strong-memory ordering.

Typically, the operating system controls access to memory-mapped I/O devices. The AMD64 architecture provides facilities for system software to specify the types of accesses and their ordering for entire regions of memory. These facilities are also used to manage the cacheability of memory regions. See “System-Management Instructions” in Volume 2 for further information.

3.8.3 Protected-Mode I/O

In protected mode, access to the I/O-address space is governed by the I/O privilege level (IOPL) field in the rFLAGS register, and the I/O-permission bitmap in the current task-state segment (TSS).

3.8.3.1 I/O-Privilege Level

RFLAGS.IOPL governs access to *IOPL-sensitive* instructions. All of the I/O instructions (IN, INS, OUT, and OUTS) are IOPL-sensitive. IOPL-sensitive instructions cannot be executed by a program unless the program’s current-privilege level (CPL) is numerically less (more privileged) than or equal to the RFLAGS.IOPL field, otherwise a general-protection exception (#GP) occurs.

Only software running at CPL = 0 can change the RFLAGS.IOPL field. Two instructions, POPF and IRET, can be used to change the field. If application software (or any software running at CPL>0) attempts to change RFLAGS.IOPL, the attempt is ignored.

System software uses RFLAGS.IOPL to control the privilege level required to access I/O-address space devices. Access can be granted on a program-by-program basis using different copies of RFLAGS for every program, each with a different IOPL. RFLAGS.IOPL acts as a global control over a program’s access to I/O-address space devices. System software can grant less-privileged programs access to individual I/O devices (overriding RFLAGS.IOPL) by using the I/O-permission bitmap stored in a program’s TSS. For details about the I/O-permission bitmap, see “I/O-Permission Bitmap” in Volume 2.

3.9 Memory Optimization

Generally, application software is unaware of the memory hierarchy implemented within a particular system design. The application simply sees a homogenous address space within a single level of memory. In reality, both system and processor implementations can use any number of techniques to speed up accesses into memory, doing so in a manner that is transparent to applications. Application software can be written to maximize this speed-up even though the methods used by the hardware are not visible to the application. This section gives an overview of the memory hierarchy and access techniques that can be implemented within a system design, and how applications can optimize their use.

3.9.1 Accessing Memory

Implementations of the AMD64 architecture *commit* the results of each instruction—that is, store the result of the executed instruction in software-visible resources, such as a register (including flags), the data cache, an internal write buffer, or memory—in program order, which is the order specified by the instruction sequence in a program. Transparent to the application, implementations can execute instructions in any order and temporarily hold out-of-order results until the instructions are committed. Implementations can also *speculatively* execute instructions—executing instructions before knowing their results will be used (for example, executing both sides of a branch). By executing instructions out-of-order and speculatively, a processor can boost application performance by executing instructions that are ready, rather than delaying them behind instructions that are waiting for data. However, the processor commits results in program order (the order expected by software).

When executing instructions out-of-order and speculatively, processor implementations often find it useful to also allow out-of-order and speculative memory accesses. However, such memory accesses are potentially visible to software and system devices. The following sections describe the architectural rules for memory accesses. See “Memory System” in Volume 2 for information on how system software can further specify the flexibility of memory accesses.

3.9.1.1 Read Ordering

The ordering of memory reads does not usually affect program execution because the ordering does not usually affect the state of software-visible resources. The rules governing read ordering are:

- *Out-of-order reads are allowed.* Out-of-order reads can occur as a result of out-of-order instruction execution. The processor can read memory out-of-order to prevent stalling instructions that are executed out-of-order.
- *Speculative reads are allowed.* A speculative read occurs when the processor begins executing a memory-read instruction before it knows whether the instruction’s result will actually be needed. For example, the processor can predict a branch to occur and begin executing instructions following the predicted branch, before it knows whether the prediction is valid. When one of the speculative instructions reads data from memory, the read itself is speculative.
- *Reads can usually be reordered ahead of writes.* Reads are generally given a higher priority by the processor than writes because instruction execution stalls if the read data required by an instruction is not immediately available. Allowing reads ahead of writes usually maximizes software performance.

  Reads can be reordered ahead of writes, except that a read *cannot* be reordered ahead of a prior write if the read is from the same location as the prior write. In this case, the read instruction stalls until the write instruction is committed. This is because the result of the write instruction is required by the read instruction for software to operate correctly.

Some system devices might be sensitive to reads. Normally, applications do not have direct access to system devices, but instead call an operating-system service routine to perform the access on the application’s behalf. In this case, it is system software’s responsibility to enforce strong read-ordering.

3.9.1.2 Write Ordering

Writes affect program order because they affect the state of software-visible resources. The rules governing write ordering are restrictive:

- *Generally, out-of-order writes are not allowed.* Write instructions executed out-of-order cannot *commit* (write) their result to memory until all previous instructions have completed in program order. The processor can, however, hold the result of an out-of-order write instruction in a private buffer (not visible to software) until that result can be committed to memory.

  System software can create non-cacheable *write-combining* regions in memory when the order of writes is known to not affect system devices. When writes are performed to write-combining memory, they can appear to complete out of order relative to other writes. See “Memory System” in Volume 2 for additional information.
- *Speculative writes are not allowed.* As with out-of-order writes, speculative write instructions cannot commit their result to memory until all previous instructions have completed in program order. Processors can hold the result in a private buffer (not visible to software) until the result can be committed.

3.9.1.3 Atomicity of accesses.

Single load or store operations (from instructions that do just a single load or store) are naturally atomic on any AMD64 processor as long as they do not cross an aligned 8-byte boundary. Accesses up to eight bytes in size which do cross such a boundary may be performed atomically using certain instructions with a lock prefix, such as XCHG, CMPXCHG or CMPXCHG8B, as long as all such accesses are done using the same technique. (Note that misaligned locked accesses may be subject to heavy performance penalties.) CMPXCHG16B can be used to perform 16-byte atomic accesses in 64-bit mode (with certain alignment restrictions).

3.9.2 Forcing Memory Order

Special instructions are provided for application software to force memory ordering in situations where such ordering is important. These instructions are:

- *Load Fence*—The LFENCE instruction forces ordering of memory loads (reads). All memory loads preceding the LFENCE (in program order) are completed prior to completing memory loads following the LFENCE. Memory loads cannot be reordered around an LFENCE instruction, but other non-serializing instructions (such as memory writes) can be reordered around the LFENCE.
- *Store Fence*—The SFENCE instruction forces ordering of memory stores (writes). All memory stores preceding the SFENCE (in program order) are completed prior to completing memory stores following the SFENCE. Memory stores cannot be reordered around an SFENCE instruction, but other non-serializing instructions (such as memory loads) can be reordered around the SFENCE.
- *Memory Fence*—The MFENCE instruction forces ordering of all memory accesses (reads and writes). All memory accesses preceding the MFENCE (in program order) are completed prior to completing any memory access following the MFENCE. Memory accesses cannot be reordered

around an MFENCE instruction. Additionally in AMD64 processors, MFENCE is a serializing instruction (see below).

Although they serve different purposes, other instructions can be used as read/write barriers when the order of memory accesses must be strictly enforced. These read/write barrier instructions force all prior reads and writes to complete before subsequent reads or writes are executed. Unlike the fence instructions listed above, these other instructions alter the software-visible state. This makes these instructions less general and more difficult to use as read/write barriers than the fence instructions, although their use may reduce the total number of instructions executed. The following instructions are usable as read/write barriers:

- *Serializing instructions*—Serializing instructions force the processor to commit the serializing instruction and all previous instructions, then restart instruction fetching at the next instruction. This flushes any speculatively fetched instructions that may be in execution behind the serializing instruction. The serializing instructions available to applications (aside from MFENCE; see above) are CPUID and IRET. A serializing instruction is committed when the following operations are complete:
  - The instruction has executed.
  - All registers modified by the instruction are updated.
  - All memory updates performed by the instruction are complete.
  - All data held in the write buffers have been written to memory. (Write buffers are described in “Write Buffering” on page 103).
- *I/O instructions*—Reads from and writes to I/O-address space use the IN and OUT instructions, respectively. When the processor executes an I/O instruction, it orders it with respect to other loads and stores, depending on the instruction:
  - IN instructions (IN, INS, and REP INS) are not executed until all previous stores to memory and I/O-address space are complete.
  - Instructions *following* an OUT instruction (OUT, OUTS, or REP OUTS) are not executed until all previous stores to memory and I/O-address space are complete, including the store performed by the OUT.
- *Locked instructions*—A locked instruction is one that contains the LOCK instruction prefix. A locked instruction is used to perform an atomic read-modify-write operation on a memory operand, so it needs exclusive access to the memory location for the duration of the operation. Locked instructions order memory accesses in the following way:
  - All previous loads and stores (in program order) are completed prior to executing the locked instruction.
  - The locked instruction is completed before allowing loads and stores for subsequent instructions (in program order) to occur.

Only certain instructions can be locked. See “Lock Prefix” in Volume 3 for a list of instructions that can use the LOCK prefix.

3.9.3 Caches

Depending on the instruction, operands can be encoded in the instruction opcode or located in registers, I/O ports, or memory locations. An operand that is located in memory can actually be physically present in one or more locations within a system's *memory hierarchy*.

3.9.3.1 Memory Hierarchy

A system's memory hierarchy may have some or all of the following levels:

- *Main Memory*—Main memory is external to the processor chip and is the memory-hierarchy level farthest from the processor's execution units. All physical-memory addresses are present in main memory, which is implemented using relatively slow, but high-density memory devices.
- *External Caches*—External caches are external to the processor chip, but are implemented using lower-capacity, higher-performance memory devices than system memory. The system uses external caches to hold copies of frequently-used instructions and data found in main memory. A subset of the physical-memory addresses can be present in the external caches at any time. A system can contain any number of external caches, or none at all.
- *Internal Caches*—Internal caches are present on the processor chip itself, and are the closest memory-hierarchy level to the processor's execution units. Because of their presence on the processor chip, access to internal caches is very fast. Internal caches contain copies of the most frequently-used instructions and data found in main memory *and* external caches, and their capacities are relatively small in comparison to external caches. A processor implementation can contain any number of internal caches, or none at all. Implementations often contain a first-level instruction cache and first-level data (operand) cache, and they may also contain a higher-capacity (and slower) second- and even third-level internal cache for storing both instructions and data.

Figure 3-19 on page 103 shows an example of a four-level memory hierarchy that combines main memory, external third-level (L3) cache, and internal second-level (L2) and two first-level (L1) caches. As the figure shows, the first-level and second-level caches are implemented on the processor chip, and the third-level cache is external to the processor. The first-level cache is a split cache, with separate caches used for instructions and data. The second-level and third-level caches are unified (they contain both instructions and data). Memory at the highest levels of the hierarchy have greater capacity (larger size), but have slower access, than memory at the lowest levels.

Using caches to store frequently used instructions and data can result in significantly improved software performance by avoiding accesses to the slower main memory. Applications function identically on systems without caches and on systems with caches, although cacheless systems typically execute applications more slowly. Application software can, however, be optimized to make efficient use of caches when they are present, as described later in this section.

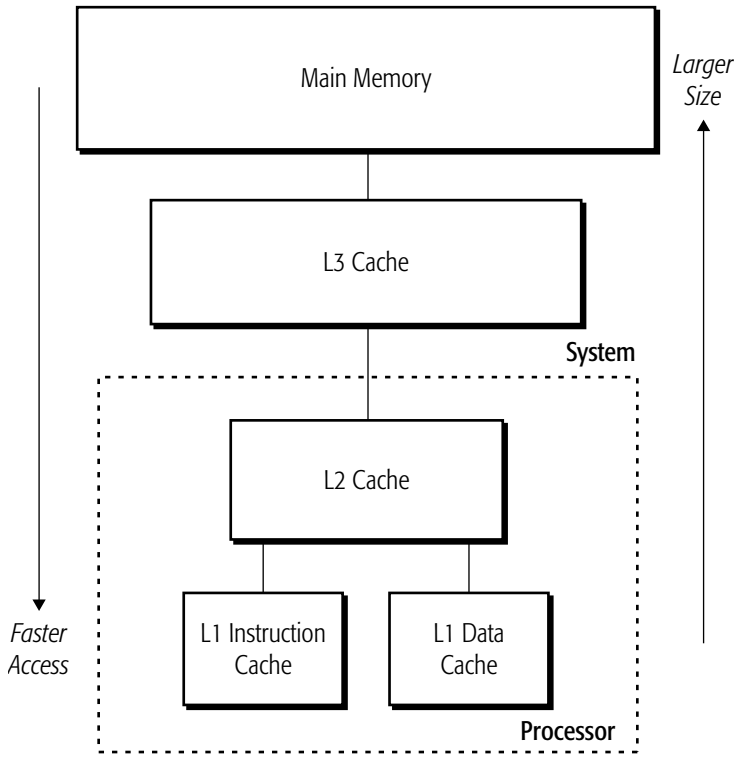


Figure 3-19. Memory Hierarchy Example

3.9.3.2 Write Buffering

Processor implementations can contain write-buffers attached to the internal caches. Write buffers can also be present on the interface used to communicate with the external portions of the memory hierarchy. Write buffers temporarily hold data writes when main memory or the caches are busy responding to other memory-system accesses. The existence of write buffers is transparent to software. However, some of the instructions used to optimize memory-hierarchy performance can affect the write buffers, as described in “Forcing Memory Order” on page 100.

3.9.4 Cache Operation

Although the existence of caches is transparent to application software, a simple understanding how caches are accessed can assist application developers in optimizing their code to run efficiently when caches are present.

Caches are divided into fixed-size blocks, called *cache lines*. Typically, implementations have either 32-byte or 64-byte cache lines. The processor *allocates* a cache line to correspond to an identically-sized region in main memory. After a cache line is allocated, the addresses in the corresponding region of main memory are used as addresses into the cache line. It is the processor’s responsibility to keep the contents of the allocated cache line *coherent* with main memory. Should another system device

access a memory address that is cached, the processor maintains coherency by providing the correct data back to the device and main memory.

When a memory-read occurs as a result of an instruction fetch or operand access, the processor first checks the cache to see if the requested information is available. A *read hit* occurs if the information is available in the cache, and a *read miss* occurs if the information is not available. Likewise, a *write hit* occurs if a memory write can be stored in the cache, and a *write miss* occurs if it cannot be stored in the cache.

A read miss or write miss can result in the allocation of a cache line, followed by a *cache-line fill*. Even if only a single byte is needed, all bytes in a cache line are loaded from memory by a cache-line fill. Typically, a cache-line fill must write over an existing cache line in a process called a *cache-line replacement*. In this case, if the existing cache line is modified, the processor performs a cache-line *writeback* to main memory prior to performing the cache-line fill.

Cache-line writebacks help maintain coherency between the caches and main memory. Internally, the processor can also maintain cache coherency by *internally probing* (checking) the other caches and write buffers for a more recent version of the requested data. External devices can also check a processor's caches and write buffers for more recent versions of data by *externally probing* the processor. All coherency operations are performed in hardware and are completely transparent to applications.

3.9.4.1 Cache Coherency and MOESI

Implementations of the AMD64 architecture maintain coherency between memory and caches using a five-state protocol known as MOESI. The five MOESI states are *modified*, *owned*, *exclusive*, *shared*, and *invalid*. See "Memory System" in Volume 2 for additional information on MOESI and cache coherency.

3.9.4.2 Instruction Cache Coherency

Instruction caches in AMD64 processors do not support in-cache updates. Any stores that hit a line in an instruction cache will cause that line to be invalidated by hardware to maintain coherency of the cache contents. The line may then be re-fetched and loaded into the cache as needed by the instruction fetch logic, reflecting the update. Special considerations for self-modifying code (code which writes into its own pending instruction stream) and cross-modifying code (code which writes into the active instruction stream of anotherthread) may be found in Volume 2, Section 7.6.1.

3.9.5 Cache Pollution

Because cache sizes are limited, caches should be filled only with data that is frequently used by an application. Data that is used infrequently, or not at all, is said to *pollute* the cache because it occupies otherwise useful cache lines. Ideally, the best data to cache is data that adheres to the *principle of locality*. This principle has two components: *temporal locality* and *spatial locality*.

- *Temporal locality* refers to data that is likely to be used more than once in a short period of time. It is useful to cache temporal data because subsequent accesses can retrieve the data quickly. Non-

temporal data is assumed to be used once, and then not used again for a long period of time, or ever. Caching of non-temporal data pollutes the cache and should be avoided.

Cache-control instructions (“Cache-Control Instructions” on page 105) are available to applications to minimize cache pollution caused by non-temporal data.

- *Spatial locality* refers to data that resides at addresses adjacent to or very close to the data being referenced. Typically, when data is accessed, it is likely the data at nearby addresses will be accessed in a short period of time. Caches perform cache-line fills in order to take advantage of spatial locality. During a cache-line fill, the referenced data and nearest neighbors are loaded into the cache. If the characteristics of spacial locality do not fit the data used by an application, then the cache becomes polluted with a large amount of unreferenced data.

  Applications can avoid problems with this type of cache pollution by using data structures with good spatial-locality characteristics.

Another form of cache pollution is *stale data*. Data that adheres to the principle of locality can become stale when it is no longer used by the program, or won’t be used again for a long time. Applications can use the CLFLUSH instruction to remove stale data from the cache.

3.9.6 Cache-Control Instructions

General control and management of the caches is performed by system software and not application software. System software uses special registers to assign *memory types* to physical-address ranges, and page-attribute tables are used to assign memory types to virtual address ranges. Memory types define the cacheability characteristics of memory regions and how coherency is maintained with main memory. See “Memory System” in Volume 2 for additional information on memory typing.

Instructions are available that allow application software to control the cacheability of data it uses on a more limited basis. These instructions can be used to boost an application’s performance by prefetching data into the cache, and by avoiding cache pollution. Run-time analysis tools and compilers may be able to suggest the use of cache-control instructions for critical sections of application code.

3.9.6.1 Cache Prefetching

Applications can prefetch entire cache lines into the caching hierarchy using one of the prefetch instructions. The prefetch should be performed in advance, so that the data is available in the cache when needed. Although load instructions can mimic the prefetch function, they do not offer the same performance advantage, because a load instruction may cause a subsequent instruction to stall until the load completes, but a prefetch instruction will never cause such a stall. Load instructions also unnecessarily require the use of a register, but prefetch instructions do not.

The instructions available in the AMD64 architecture for cache-line prefetching include one SSE instruction and two 3DNow! instructions:

- *PREFETCHlevel*—(an SSE instruction) Prefetches read/write data into a specific level of the cache hierarchy. If the requested data is already in the desired cache level or closer to the processor (lower cache-hierarchy level), the data is not prefetched. If the operand specifies an invalid

memory address, no exception occurs, and the instruction has no effect. Attempts to prefetch data from non-cacheable memory, such as video frame buffers, or data from write-combining memory, are also ignored. The exact actions performed by the PREFETCH*level* instructions depend on the processor implementation. Current AMD processor families map all PREFETCH*level* instructions to a PREFETCH. Refer to the *Optimization Guide for AMD Athlon™ 64 and AMD Opteron™ Processors*, order# 25112, for details relating to a particular processor family, brand or model.

- PREFETCHT0—Prefetches temporal data into the entire cache hierarchy.
- PREFETCHT1—Prefetches temporal data into the second-level (L2) and higher-level caches, but not into the L1 cache.
- PREFETCHT2—Prefetches temporal data into the third-level (L3) and higher-level caches, but not into the L1 or L2 cache.
- PREFETCHNTA—Prefetches non-temporal data into the processor, minimizing cache pollution. The specific technique for minimizing cache pollution is implementation-dependent and can include such techniques as allocating space in a software-invisible buffer, allocating a cache line in a single cache or a specific way of a cache, etc.

- *PREFETCH*—(a 3DNow! instruction) Prefetches read data into the L1 data cache. Data can be written to such a cache line, but doing so can result in additional delay because the processor must signal externally to negotiate the right to change the cache line's cache-coherency state for the purpose of writing to it.
- *PREFETCHW*—(a 3DNow! instruction) Prefetches write data into the L1 data cache. Data can be written to the cache line without additional delay, because the data is already prefetched in the *modified* cache-coherency state. Data can also be read from the cache line without additional delay. However, prefetching write data takes longer than prefetching read data if the processor must wait for another caching master to first write-back its modified copy of the requested data to memory before the prefetch request is satisfied.

The PREFETCHW instruction provides a hint to the processor that the cache line is to be modified, and is intended for use when the cache line will be written to shortly after the prefetch is performed. The processor can place the cache line in the modified state when it is prefetched, but before it is actually written. Doing so can save time compared to a PREFETCH instruction, followed by a subsequent cache-state change due to a write.

To prevent a false-store dependency from stalling a prefetch instruction, prefetched data should be located at least one cache-line away from the address of any surrounding data write. For example, if the cache-line size is 32 bytes, avoid prefetching from data addresses within 32 bytes of the data address in a preceding write instruction.

3.9.6.2 Non-Temporal Stores

Non-temporal store instructions are provided to prevent memory writes from being stored in the cache, thereby reducing cache pollution. These non-temporal store instructions are specific to the type of register they write:

- *GPR Non-temporal Stores*—MOVNTI.

- *YMM/XMM Non-temporal Stores*—(V)MASKMOVDQU, (V)MOVNTDQ, (V)MOVNTPD, and (V)MOVNTPS.
- *MMX Non-temporal Stores*—MASKMOVQ and MOVNTQ.

3.9.6.3 Removing Stale Cache Lines

When cache data becomes stale, it occupies space in the cache that could be used to store frequently-accessed data. Applications can use the CLFLUSH instruction to free a stale cache-line for use by other data. CLFLUSH writes the contents of a cache line to memory and then invalidates the line in the cache and in all other caches in the cache hierarchy that contain the line. Once invalidated, the line is available for use by the processor and can be filled with other data.

3.10 Performance Considerations

In addition to typical code optimization techniques, such as those affecting loops and the inlining of function calls, the following considerations may help improve the performance of application programs written with general-purpose instructions.

These are implementation-independent performance considerations. Other considerations depend on the hardware implementation. For information about such implementation-dependent considerations and for more information about application performance in general, see the data sheets and the software-optimization guides relating to particular hardware implementations.

3.10.1 Use Large Operand Sizes

Loading, storing, and moving data with the largest relevant operand size maximizes the memory bandwidth of these instructions.

3.10.2 Use Short Instructions

Use the shortest possible form of an instruction (the form with fewest opcode bytes). This increases the number of instructions that can be decoded at any one time, and it reduces overall code size.

3.10.3 Align Data

Data alignment directly affects memory-access performance. Data alignment is particularly important when accessing *streaming* (also called *non-temporal*) data—data that will not be reused and therefore should not be cached. Data alignment is also important in cases where data that is written by one instruction is subsequently read by a subsequent instruction soon after the write.

3.10.4 Avoid Branches

Branching can be very time-consuming. If the body of a branch is small, the branch may be replaceable with conditional move (CMOV*cc*) instructions, or with 128-bit or 64-bit media instructions that simulate predicated parallel execution or parallel conditional moves.

3.10.5 Prefetch Data

Memory latency can be substantially reduced—especially for data that will be used multiple times—by prefetching such data into various levels of the cache hierarchy. Software can use the PREFETCH*x* instructions very effectively in such cases. One PREFETCH*x* per cache line should be used.

Some of the best places to use prefetch instructions are inside loops that process large amounts of data. If the loop goes through less than one cache line of data per iteration, partially unroll the loop. Try to use virtually all of the prefetched data. This usually requires unit-stride memory accesses—those in which all accesses are to contiguous memory locations.

For data that will be used only once in a procedure, consider using non-temporal accesses. Such accesses are not burdened by the overhead of cache protocols.

3.10.6 Keep Common Operands in Registers

Keep frequently used values in registers rather than in memory. This avoids the comparatively long latencies for accessing memory.

3.10.7 Avoid True Dependencies

Spread out true dependencies (write-read or flow dependencies) to increase the opportunities for parallel execution. This spreading out is not necessary for anti-dependencies and output dependencies.

3.10.8 Avoid Store-to-Load Dependencies

Store-to-load dependencies occur when data is stored to memory, only to be read back shortly thereafter. Hardware implementations of the architecture may contain means of accelerating such store-to-load dependencies, allowing the load to obtain the store data before it has been written to memory. However, this acceleration might be available only when the addresses and operand sizes of the store and the dependent load are matched, and when both memory accesses are aligned. Performance is typically optimized by avoiding such dependencies altogether and keeping the data, including temporary variables, in registers.

3.10.9 Optimize Stack Allocation

When allocating space on the stack for local variables and/or outgoing parameters within a procedure, adjust the stack pointer and use moves rather than pushes. This method of allocation allows random access to the outgoing parameters, so that they can be set up when they are calculated instead of being held in a register or memory until the procedure call. This method also reduces stack-pointer dependencies.

3.10.10 Consider Repeat-Prefix Setup Time

The repeat instruction prefixes have a setup overhead. If the repeated count is variable, the overhead can sometimes be avoided by substituting a simple loop to move or store the data. Repeated string instructions can be expanded into equivalent sequences of inline loads and stores. For details, see “Repeat Prefixes” in Volume 3.

3.10.11 Replace GPR with Media Instructions

Some integer-based programs can be made to run faster by using 128-bit media or 64-bit media instructions. These instructions have their own register sets. Because of this, they relieve register pressure on the GPR registers. For loads, stores, adds, shifts, etc., media instructions may be good substitutes for general-purpose integer instructions. GPR registers are freed up, and the media instructions increase opportunities for parallel operations.

3.10.12 Organize Data in Memory Blocks

Organize frequently accessed constants and coefficients into cache-line-size blocks and prefetch them. Procedures that access data arranged in memory-bus-sized blocks, or memory-burst-sized blocks, can make optimum use of the available memory bandwidth.

4 Streaming SIMD Extensions Media and Scientific Programming

This chapter describes the programming model and instructions that make up the Streaming SIMD Extensions (SSE). SSE instructions perform integer and floating-point operations primarily on vector operands (a subset of the instructions take scalar operands) held in the YMM/XMM registers or loaded from memory. They can speed up certain types of procedures—typically high-performance media and scientific procedures—by substantial factors, depending on data element size and the regularity and locality of data accessed from memory.

4.1 Overview

Most of the SSE arithmetic instructions perform parallel operations on pairs of vectors. *Vector* operations are also called *packed* or *SIMD* (single-instruction, multiple-data) operations. They take vector operands consisting of multiple elements and all elements are operated on in parallel. Some SSE instructions operate on scalars instead of vectors.

4.1.1 Capabilities

The SSE instructions are designed to support media and scientific applications. Many physical and mathematical objects can be modeled as a set of numbers (elements) that quantify a fixed number of attributes related to that object. These elements are then aggregated together into what is called a vector. The SSE instructions allow applications to perform mathematical operations on vectors. In a vector instruction, each element of the one or more vector operands is operated upon in parallel using the same mathematical function. The elements can be integers (from bytes to octwords) or floating-point values (either single-precision or double-precision). Arithmetic operations produce signed, unsigned, and/or saturating results.

The availability of several types of vector move instructions and (in 64-bit mode) twice the number of YMM/XMM registers (a total of 16) can drastically reduce memory-access overhead, making a substantial difference in performance.

Types of Applications

Applications well-suited to the SSE programming model include a broad range of audio, video, and graphics programs. For example, music synthesis, speech synthesis, speech recognition, audio and video compression (encoding) and decompression (decoding), 2D and 3D graphics, streaming video (up to high-definition TV), and digital signal processing (DSP) kernels are all likely to experience higher performance using SSE instructions than using other types of instructions in AMD64 architecture.

Such applications commonly use small-sized integer or single-precision floating-point data elements in repetitive loops, in which the typical operations are inherently parallel. For example, 8-bit and 16-bit data elements are commonly used for pixel information in graphics applications, in which each of

the RGB pixel components (red, green, blue, and alpha) are represented by an 8-bit or 16-bit integer. 16-bit data elements are also commonly used for audio sampling.

The SSE instructions allow multiple data elements like these to be packed into 256-bit or 128-bit vector operands located in YMM/XMM registers or memory. The instructions operate in parallel on each of the elements in these vectors. For example, 32 elements of 8-bit data can be packed into a 256-bit vector operand, so that all 32 byte elements are operated on simultaneously, and in pairs of source operands, by a single instruction.

The SSE instructions also support a broad spectrum of scientific applications. For example, their ability to operate in parallel on double-precision floating-point vector elements makes them well-suited to computations like dense systems of linear equations, including matrix and vector-space operations with real and complex numbers. In professional CAD applications, for example, high-performance physical-modeling algorithms can be implemented to simulate processes such as heat transfer or fluid dynamics.

4.1.2 Origins

The SSE instruction set includes instructions originally introduced as the Streaming SIMD Extensions (Herein referred to as SSE1), and instructions added in subsequent extensions (SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, AES, AVX, AVX2, CLMUL, FMA4, FMA, and XOP).

Collectively the SSE1, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, and SSE4A subsets are referred to as the *legacy* SSE instructions. All legacy SSE instructions support 128-bit vector operands. The *extended SSE* instructions include the AES, AVX, AVX2, CLMUL, FMA4, FMA, and XOP subsets. All extended SSE instructions provide support for 128-bit vector operands and most also support 256-bit operands.

Legacy SSE instructions support the specification of two vector operands, the AVX and AVX2 subsets support three, and AMD's FMA4 and XOP instruction sets support the specification of four 128-bit or 256-bit vector operands.

Each AVX instruction mirrors one of the legacy SSE instructions but presents different exception behavior. Most AVX instructions that operate on vector floating-point data types provide support for 256-bit vector widths. AVX2 adds support for 256-bit widths to most vector integer AVX instructions.

AVX, AVX2, FMA4, FMA, and XOP support the specification of a distinct destination register. This is called a non-destructive operation because none of the source operands is overwritten as a result of the execution of the instruction.

The assembler mnemonic for each AVX and AVX2 instruction is distinguished from the corresponding legacy form by prepending the letter *V* . In the discussion below, mnemonics for instructions which have both and a legacy SSE and an AVX form will be written (V)*mnemonic* (for example, (V)ADDPD). The mnemonics for the other extended SSE instructions also begin with the letter *V*.

4.1.3 Compatibility

The SSE instructions can be executed in any of the architecture's operating modes. Existing SSE binary programs run in legacy and compatibility modes without modification. The support provided by the AMD64 architecture for such binaries is identical to that provided by legacy x86 architectures.

To run in 64-bit mode, legacy SSE programs must be recompiled. The recompilation has no side effects on such programs, other than to provide access to the following additional resources:

- Eight additional YMM/XMM registers (for a total of 16).
- Eight additional general-purpose registers (for a total of 16 GPRs).
- Extended 64-bit width of all GPRs.
- 64-bit virtual address space.
- RIP-relative addressing mode.

The SSE instructions use data registers, a control and status register (MXCSR), rounding control, and an exception reporting and response mechanism that are distinct from and functionally independent of those used by the x87 floating-point instructions. Because of this, SSE programming support usually requires exception handlers that are distinct from those used for x87 exceptions. This support is provided by virtually all legacy operating systems for the x86 architecture.

4.2 Registers

The SSE programming model introduced the 128-bit XMM registers. In the extended SSE programming model these registers double in width to 256 bits and are designated YMM0–15. Rather than defining a separate array of registers, the extended SSE model overlays the YMM registers on the XMM registers, with each XMM register occupying the lower 128 bits of the corresponding YMM register. When referring to these registers in general, they are designated YMM/XMM0–15.

4.2.1 SSE Registers

The YMM/XMM registers are diagrammed in Figure 4-1 below. Most SSE instructions read operands from these registers or memory and store results in these registers. Operation of the SSE instructions is supported by the Media eXtension Control and Status Register (MXCSR) described below. A few SSE instructions—those that perform data conversion or move operations—can have operands located in MMX registers or general-purpose registers (GPRs).

Sixteen 256-bit YMM data registers, YMM0–YMM15, support the 256-bit media instructions. Sixteen 128-bit XMM data registers, XMM0–XMM15, support the 128-bit media instructions. They can hold operands for both vector and scalar operations utilizing the 128-bit and 256-bit integer and floating-point data types. The high eight YMM/XMM registers, YMM/XMM8–15, are available to software running in 64-bit mode for instructions that use a REX, VEX, or XOP prefix. For a discussion of the REX prefix, see "REX Prefixes" on page 79. For a discussion of VEX and XOP, see "VEX and XOP Prefixes" on page 79).

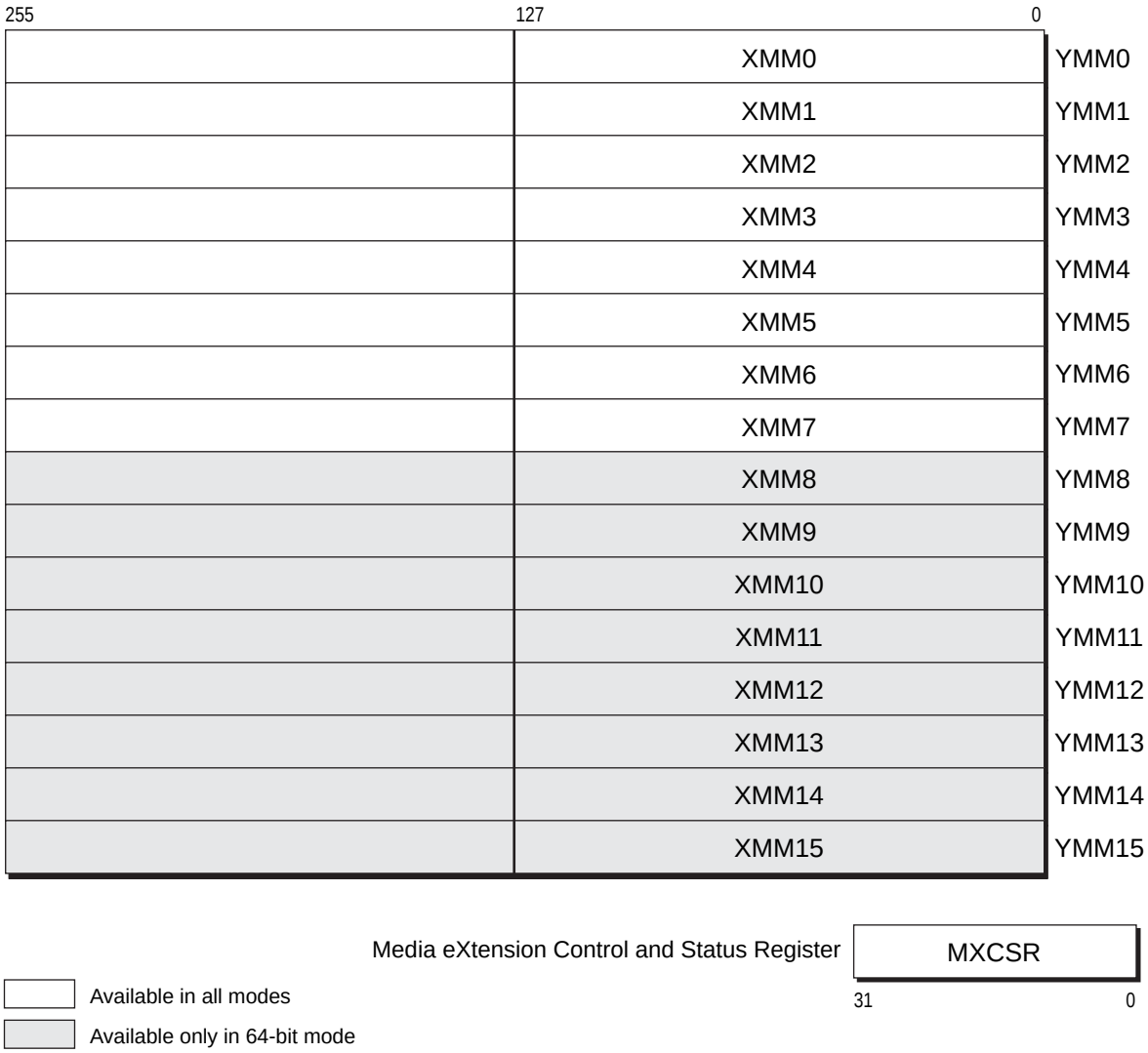


Figure 4-1. SSE Registers

Upon power-on reset, all 16 YMM/XMM registers are cleared to +0.0. However, initialization by means of the #INIT external input signal does not change the state of the YMM/XMM registers.

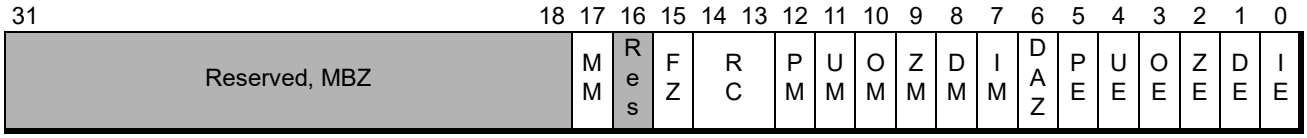
Handling of Upper Octword

128-bit media instructions read source operands from and write results to XMM registers, while the 256-bit media instructions use the 256-bit YMM registers. This raises the question—what is the disposition of the upper octword of a YMM register when the result of a 128-bit media instruction is written into the lower octword (the XMM register)? The answer differs depending on whether the instruction is a legacy SSE instruction or an extended SSE instruction. When a legacy SSE instruction writes a 128-bit result to an XMM register, the upper octword of the corresponding YMM register

remains unchanged. However, when the 128-bit form of an extended SSE instruction writes its result, the upper octword of the YMM register is cleared.

4.2.2 MXCSR Register

Figure 4-2 below shows a detailed view of the Media eXtension Control and Status Register (MXCSR). All defined fields in this register are read/write. The fields within the MXCSR apply only to operations performed by 256-bit and 128-bit media instructions. Software can load the register from memory using the XRSTOR, XRSTORS, FXRSTOR or LDMXCSR instructions, and it can store the register to memory using the XSAVE, XSAVEOPT, XSAVEC, XSAVES, FXSAVE or STMXCSR instructions.



Bits	Mnemonic	Description	Reset Bit-Value
31:18	–	Reserved, MBZ	
17	MM	Misaligned Exception Mask	0
16	–	Reserved, MBZ	
15	FZ	Flush-to-Zero for Masked Underflow	0
14:13	RC	Floating-Point Rounding Control	00
		Exception Masks	
12	PM	Precision Exception Mask	1
11	UM	Underflow Exception Mask	1
10	OM	Overflow Exception Mask	1
9	ZM	Zero-Divide Exception Mask	1
8	DM	Denormalized-Operand Exception Mask	1
7	IM	Invalid-Operation Exception Mask	1
6	DAZ	Denormals Are Zeros	0
		Exception Flags	
5	PE	Precision Exception	0
4	UE	Underflow Exception	0
3	OE	Overflow Exception	0
2	ZE	Zero-Divide Exception	0
1	DE	Denormalized-Operand Exception	0
0	IE	Invalid-Operation Exception	0

Figure 4-2. Media eXtension Control and Status Register (MXCSR)

On power-on reset, all bits are initialized to the values indicated above. However, initialization by means of the #INIT external input signal does not change the state of the MXCSR.

The six exception flags (IE, DE, ZE, OE, UE, PE) are sticky bits. (Once set by the processor, such a bit remains set until software clears it.) For details about the causes of SIMD floating-point exceptions

indicated by bits 5:0, see "SIMD Floating-Point Exception Causes" on page 220. For details about the masking of these exceptions, see "SIMD Floating-Point Exception Masking" on page 226.

Invalid-Operation Exception (IE). Bit 0. The processor sets this bit to 1 when an invalid-operation exception occurs. These exceptions are caused by many types of errors, such as an invalid operand.

Denormalized-Operand Exception (DE). Bit 1. The processor sets this bit to 1 when one of the source operands of an instruction is in denormalized form, except that if software has set the denormals are zeros (DAZ) bit, the processor does not set the DE bit. (See "Denormalized (Tiny) Numbers" on page 125.)

Zero-Divide Exception (ZE). Bit 2. The processor sets this bit to 1 when a non-zero number is divided by zero.

Overflow Exception (OE). Bit 3. The processor sets this bit to 1 when the absolute value of a rounded result is larger than the largest representable normalized floating-point number for the destination format. (See "Normalized Numbers" on page 125.)

Underflow Exception (UE). Bit 4. The processor sets this bit to 1 when the absolute value of a rounded non-zero result is too small to be represented as a normalized floating-point number for the destination format. (See "Normalized Numbers" on page 125.)

When masked by the UM bit, the processor reports a UE exception only if the UE occurs *together with* a precision exception (PE). Also, see bit 15, the flush-to-zero (FZ) bit.

Precision Exception (PE). Bit 5. The processor sets this bit to 1 when a floating-point result, after rounding, differs from the infinitely precise result and thus cannot be represented exactly in the specified destination format. The PE exception is also called the *inexact-result* exception.

Denormals Are Zeros (DAZ). Bit 6. Software can set this bit to 1 to enable the DAZ mode, if the hardware implementation supports this mode. In the DAZ mode, when the processor encounters source operands in the denormalized format it converts them to signed zero values, with the sign of the denormalized source operand, before operating on them, and the processor does not set the denormalized-operand exception (DE) bit, regardless of whether such exceptions are masked or unmasked. DAZ mode does not comply with the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754).

Support for the DAZ bit is indicated by the MXCSR_MASK field in the FXSAVE memory save area or the low 512 bytes of the XSAVE extended save area. See "Saving Media and x87 State" in Volume 2.

Exception Masks (PM, UM, OM, ZM, DM, IM). Bits 12:7. Software can set these bits to mask, or clear this bits to unmask, the corresponding six types of SIMD floating-point exceptions (PE, UE, OE, ZE, DE, IE). A bit masks its exception type when set to 1, and unmasks it when cleared to 0.

In general, masking a type of exception causes the processor to handle all subsequent instances of the exception type in a default way (the UE exception has an unusual behavior). Unmasking the exception

type causes the processor to branch to the SIMD floating-point exception service routine when an exception occurs. For details about the processor's responses to masked and unmasked exceptions, see "SIMD Floating-Point Exception Masking" on page 226.

Floating-Point Rounding Control (RC). Bits 14:13. Software uses these bits to specify the rounding method for SSE floating-point operations. The choices are:

- 00 = round to nearest (default)
- 01 = round down
- 10 = round up
- 11 = round toward zero

For details, see "Floating-Point Rounding" on page 129.

Flush-to-Zero (FZ). Bit 15. If the rounded result is tiny and the underflow mask is set, the FTZ bit causes the result to be flushed to zero. This naturally causes the result to be inexact, which causes both PE and UE to be set. The sign returned with the zero is the sign of the true result. The FTZ bit does not have any effect if the underflow mask is 0.

This response does not comply with the IEEE 754 standard, but it may offer higher performance than can be achieved by responding to an underflow in this circumstance. The FZ bit is only effective if the UM bit is set to 1. If the UM bit is cleared to 0, the FZ bit is ignored. For details, see Table 4-16 on page 227.

Misaligned Exception Mask (MM). Bit 17. This bit is applicable to processors that support Misaligned SSE Mode. For these processors, MM controls the exception behavior triggered by an attempt to access a misaligned vector memory operand. If the misaligned exception mask (MM) is set to 1, an attempt to access a non-aligned vector memory operand does not cause a #GP exception, but is instead subject to alignment checking. When MM is set and alignment checking is enabled, a #AC exception is generated, if the memory operand is not aligned. When MM is set and alignment checking is not enabled, no exception is triggered by accessing a non-aligned vector operand.

Support for Misaligned SSE Mode is indicated by CPUID Fn8000_0001_ECX[MisAlignSse] = 1. For details on alignment requirements, see "Data Alignment" on page 120.

The corresponding MXCSR_MASK bit (17) is 1, regardless of whether MM is set or not. For details on MXCSR and MXCSR_MASK, see "SSE, MMX, and x87 Programming" in Volume 2 of this manual.

4.2.3 Other Data Registers

Some SSE instructions that perform data transfer, data conversion or data reordering operations ("Data Transfer" on page 150, "Data Conversion" on page 155, and "Data Reordering" on page 157) can access operands in the MMX or general-purpose registers (GPRs). When addressing GPRs registers in 64-bit mode, the REX instruction prefix can be used to access the extended GPRs, as described in "REX Prefixes" on page 79.

For a description of the GPR registers, see “Registers” on page 23. For a description of the MMX registers, see “MMX™ Registers” on page 246.

4.2.4 Effect on rFLAGS Register

The execution of most SSE instructions have no effect on the rFLAGS register. However, some SSE instructions, such as COMISS and PTEST, do write flag bits based on the results of a comparison. For a description of the rFLAGS register, see “Flags Register” on page 34.

4.3 Operands

Operands for most SSE instructions are held in the YMM/XMM registers, sourced from memory, or encoded in the instruction as an immediate value. Instructions operate on two distinct operand widths — either 256 bits or 128 bits. 256-bit operands may be held in one or more of the YMM registers. 128-bit operands may be held in one or more of the XMM registers. As shown in Figure 4-1 on page 114, the 128-bit XMM registers overlay the lower octword of the 256-bit YMM registers. The data types of these operands include scalar integers, integer vectors, and scalar and floating-point vectors.

4.3.1 Operand Addressing

Depending on the instruction, referenced operands may be in registers or memory.

4.3.1.1 Register Operands

Most SSE instructions can access source and destination operands in YMM or XMM registers. A few of these instructions access the MMX registers, GPR registers, rFLAGS register, or MXCSR register. The type of register addressed is specified in the instruction syntax. When addressing a GPR or YMM/XMM register, the REX instruction prefix can be used to access the eight additional GPR or YMM/XMM registers, as described in “Instruction Prefixes” on page 217. Instructions encoded with the VEX/XOP prefix can utilize an immediate byte to provide the specification of additional operands.

4.3.1.2 Memory Operands

Most SSE instructions can read memory for source operands, and some of the instructions can write results to memory. Figure 4-3 below illustrates how a vector operand is stored in memory relative to its arrangement in an SSE register.

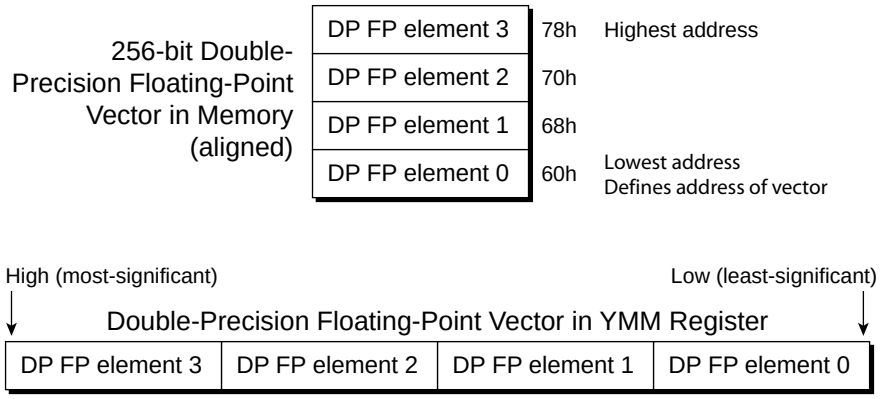


Figure 4-3. Vector (Packed) Data in Memory

This specific example shows a 256-bit double-precision floating-point vector.

This particular data type is composed of four 64-bit double-precision floating-point values (abbreviated “DP FP”, in the figure) packed into 256 bits. The four values comprise the four elements of the vector. Elements are numbered right to left. Element 0 is defined to occupy the least significant (right-most) position. Element 1 is in the next most significant position and 2 the next. Element 3 occupies the most significant (left-most) position. When held in a YMM register the elements are laid out as shown in the figure.

When stored in memory, element 0 is stored at the lowest address (60h in this example). Element 1 is stored at that address incremented by the element size in bytes (each double-precision floating-point value is 8 bytes long). Element 2 is located at the initial address plus 16 (10h) and element 3 is stored at the initial address plus 24 (18h). Each element is stored based on the rules for the fundament data type of the element (double-precision floating-point in this example). See Section 4.3.3.3 “Floating-Point Data Types” on page 123 for details on how double-precision floating-point values are represented in registers and memory.

The address of a vector is the same as the address of element 0 (60h in this example). This vector is said to be *naturally aligned* (or, simply, *aligned*) because it is located at an address that is an integer multiple of its size in bytes (32, in this case). Alignment of vector operands is not required. See “Data Alignment” below.

Other vector data types are stored in memory in an analogous fashion with the lowest indexed element placed at the lowest address.

4.3.1.3 Immediate Operands

Immediate operands are used in certain data-conversion, vector-shift, and vector-compare instructions. Such instructions take 8-bit immediates, which provide control for the operation.

4.3.1.4 I/O Ports

I/O ports in the I/O address space cannot be directly addressed by SSE instructions, and although memory-mapped I/O ports can be addressed by such instructions, doing so may produce unpredictable results, depending on the hardware implementation of the architecture.

4.3.2 Data Alignment

Generally, legacy SSE instructions that attempt to access a vector operand in memory that is not naturally aligned trigger a general-protection exception (#GP).

AMD processors that support Misaligned SSE Mode may be programmed to disable this exception behavior for legacy load/execute SSE instructions. For these processors, exception behavior on misaligned memory access for vector operands of load/execute instructions is controlled by the MXCSR.MM bit. If MM is not set, the default behavior occurs (a #GP results).

If MXCSR.MM is set, the #GP is inhibited and the exception behavior depends on the alignment checking mechanism. If alignment checking is enabled (CR0.AM = 1 and rFLAGS.AC = 1), a misaligned memory access to a vector operand will trigger an #AC exception. On the other hand, if alignment checking is disabled, no exception will be triggered.

Support for Misaligned SSE Mode is indicated by CPUID Fn8000_0001_ECX[MisAlignSse] = 1. For information on using the CPUID instruction to determine support for Misaligned SSE Mode, see the description of the CPUID instruction in Volume 3 and the definition of the MisAlignSse feature flag in Appendix E of Volume 3.

The FXSAVE, FXRSTOR, (V)MOVAPD, (V)MOVAPS, and (V)MOVDQA, (V)MOVNTDQ, (V)MOVNTPD and (V)MOVNTPS instructions do *not* support misaligned accesses. These instructions always generate an exception when attempting to access misaligned data. See individual instruction listings for specific alignment requirements.

Legacy SSE instructions that manipulate scalar operands never trigger a #GP due to data misalignment, nor do any of the following instructions:

- LDDQU—Load Unaligned Double Quadword
- MASKMOVDQU—Masked Move Double Quadword Unaligned.
- MOVDQU—Move Unaligned Double Quadword.
- MOVUPD—Move Unaligned Packed Double-Precision Floating-Point.
- MOVUPS—Move Unaligned Packed Single-Precision Floating-Point.
- PCMPESTRI—Packed Compare Explicit Length Strings Return Index
- PCMPESTRM—Packed Compare Explicit Length Strings Return Mask
- PCMPISTRI—Packed Compare Implicit Length Strings Return Index
- PCMPISTRM—Packed Compare Implicit Length Strings Return Mask

For extended SSE instructions, the MXCSR.MM bit does not control exception behavior. Only those extended SSE instructions that explicitly require aligned memory operands (VMOVAPS/PD, VMOVDQA, VMOVNTPS/PD, and VMOVNTDQ) will result in a general protection exception (#GP) when attempting to access unaligned memory operands.

For all other extended SSE instructions, unaligned memory accesses do not result in a #GP. However, software can enable alignment checking, where misaligned memory accesses cause an #AC exception, by the means specified above.

While the architecture does not impose data-alignment requirements for SSE instructions (except for those that explicitly demand it), the consequence of storing operands at unaligned locations is that accesses to those operands may require more processor and bus cycles than for aligned accesses. See “Data Alignment” on page 43 for details.

4.3.3 SSE Instruction Data Types

Most SSE instructions operate on packed (also called vector) data. These data types are aggregations of the fundamental data types—signed and unsigned integers and single- and double-precision floating-point numbers. The following sections describe the encoding and characteristics of these data types.

4.3.3.1 Integer Data Types

The architecture defines signed and unsigned integers in sizes from 8 to 128 bits. The characteristics of these data types are described below.

Sign. The sign bit is the most-significant bit—bit 7 for a byte, bit 15 for a word, bit 31 for a doubleword, bit 63 for a quadword, or bit 127 for a double quadword. Arithmetic instructions that are not specifically named as unsigned perform signed two’s-complement arithmetic.

Range of Representable Values. Table 4-1 below shows the range of representable values for the integer data types.

Table 4-1. Range of Values of Integer Data Types

Data-Type Interpretation		Byte	Word	Doubleword	Quadword	Double Quadword
Unsigned integers	Base-2 (exact)	0 to $+2^8-1$	0 to $+2^{16}-1$	0 to $+2^{32}-1$	0 to $+2^{64}-1$	0 to $+2^{128}-1$
	Base-10 (approx.)	0 to 255	0 to 65,535	0 to $4.29 * 10^9$	0 to $1.84 * 10^{19}$	0 to $3.40 * 10^{38}$
Signed integers[1]	Base-2 (exact)	-2^7 to $+(2^7-1)$	-2^{15} to $+(2^{15}-1)$	-2^{31} to $+(2^{31}-1)$	-2^{63} to $+(2^{63}-1)$	-2^{127} to $+(2^{127}-1)$
	Base-10 (approx.)	-128 to +127	-32,768 to +32,767	$-2.14 * 10^9$ to $+2.14 * 10^9$	$-9.22 * 10^{18}$ to $+9.22 * 10^{18}$	$-1.70 * 10^{38}$ to $+1.70 * 10^{38}$

Note:
1. *The sign bit is the most-significant bit (bit 7 for a byte, bit 15 for a word, bit 31 for doubleword, bit 63 for quadword, bit 127 for double quadword.).*

Saturation. Saturating (also called limiting or clamping) instructions limit the value of a result to the maximum or minimum value representable by the applicable data type. Saturating versions of integer vector-arithmetic instructions operate on byte-sized and word-sized elements. These instructions—for example, (V)PACKx, (V)PADDSx, (V)PADDUSx, (V)PSUBSx, and (V)PSUBUSx—saturate signed or unsigned data at the vector-element level when the element reaches its maximum or minimum representable value. Saturation avoids overflow or underflow errors. Many of the integer multiply and accumulate instructions saturate the cumulative results of the multiplication and addition (accumulation) operations before writing the final results to the destination (accumulator) register.

Note, however, that not all multiply and accumulate instructions saturate results.

The examples in Table 4-2 below illustrate saturating and non-saturating results with word operands. Saturation for other data-type sizes follows similar rules. Once saturated, the saturated value is treated like any other value of its type. For example, if 0001h is subtracted from the saturated value, 7FFFh, the result is 7FFEh.

Table 4-2. Saturation Examples

Operation	Non-Saturated Infinitely Precise Result	Saturated Signed Result	Saturated Unsigned Result
7000h + 2000h	9000h	7FFFh	9000h
7000h + 7000h	E000h	7FFFh	E000h
F000h + F000h	1E000h	E000h	FFFFh
9000h + 9000h	12000h	8000h	FFFFh
7FFFh + 0100h	80FFh	7FFFh	80FFh
7FFFh + FF00h	17EFFh	7EFFh	FFFFh

Arithmetic instructions not specifically designated as saturating perform non-saturating, twos-complement arithmetic.

4.3.3.2 Other Fixed-Point Operands

The architecture provides specific support only for integer fixed-point operands—those in which an implied binary point is always located to the right of bit 0. Nevertheless, software may use fixed-point operands in which the implied binary point is located in any position. In such cases, software is responsible for managing the interpretation of such implied binary points, as well as any redundant sign bits that may occur during multiplication.

4.3.3.3 Floating-Point Data Types

The floating-point data types, shown in Figure 4-4 below, include 32-bit single precision and 64-bit double precision. Both formats are fully compatible with the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754). The SSE instructions operate internally on floating-point data types in the precision specified by each instruction.

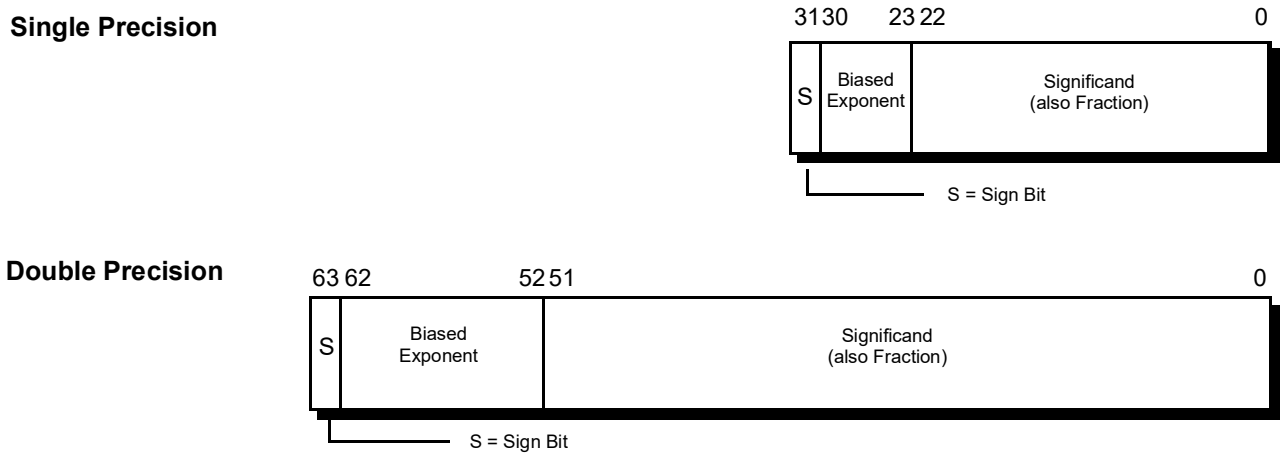


Figure 4-4. Floating-Point Data Types

Both of the floating-point data types consist of a sign (0 = positive, 1 = negative), a biased exponent (base-2), and a significand, which represents the integer and fractional parts of the number. The integer bit (also called the *J bit)* is implied (called a *hidden integer bit*). The value of an implied integer bit can be inferred from number encodings, as described in Section "Floating-Point Number Encodings" on page 127. The bias of the exponent is a constant that makes the exponent always positive and allows reciprocation, without overflow, of the smallest normalized number representable by that data type.

Specifically, the data types are formatted as follows:

- *Single-Precision Format*—This format includes a 1-bit sign, an 8-bit biased exponent whose value is 127, and a 23-bit significand. The integer bit is implied, making a total of 24 bits in the significand.
- *Double-Precision Format*—This format includes a 1-bit sign, an 11-bit biased exponent whose value is 1023, and a 52-bit significand. The integer bit is implied, making a total of 53 bits in the significand.

Table 4-3 shows the range of finite values representable by the two floating-point data types.

Table 4-3. Range of Values in Normalized Floating-Point Data Types

Data Type	Range of Normalized[1] Values	
	Base 2 (exact)	Base 10 (approximate)
Single Precision	2^{-126} to $2^{127} * (2 - 2^{-23})$	$1.17 * 10^{-38}$ to $+3.40 * 10^{38}$
Double Precision	2^{-1022} to $2^{1023} * (2 - 2^{-52})$	$2.23 * 10^{-308}$ to $+1.79 * 10^{308}$

Note:
1. See "Normalized Numbers" on page 125 for a definition of "normalized".

For example, in the single-precision format, the largest normal number representable has an exponent of FEh and a significand of 7FFFFFh, with a numerical value of $2^{127} * (2 - 2^{-23})$. Results that overflow above the maximum representable value return either the maximum representable normalized number (see "Normalized Numbers" on page 125) or infinity, with the sign of the true result, depending on the rounding mode specified in the rounding control (RC) field of the MXCSR register. Results that underflow below the minimum representable value return either the minimum representable normalized number or a denormalized number (see "Denormalized (Tiny) Numbers" on page 125), with the sign of the true result, or a result determined by the SIMD floating-point exception handler, depending on the rounding mode and the underflow-exception mask (UM) in the MXCSR register (see "Unmasked Responses" on page 229).

Compatibility with x87 Floating-Point Data Types

The results produced by SSE floating-point instructions comply fully with the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754), because these instructions represent data in the single-precision or double-precision data types throughout their operations. The x87 floating-point instructions, however, by default perform operations in the double-extended-precision format. Because of this, x87 instructions operating on the same source operands as SSE floating-point instructions may return results that are slightly different in their least-significant bits.

Floating-Point Number Types

A SSE floating-point value can be one of five types, as follows:

- Normal
- Denormal (Tiny)
- Zero

- Infinity
- Not a Number (NaN)

In common engineering and scientific usage, floating-point numbers—also called *real numbers*—are represented in base (radix) 10. A non-zero number consists of a *sign*, a normalized *significand*, and a signed *exponent*, as in:

```
+2.71828 e0
```

Both large and small numbers are representable in this notation, subject to the limits of data-type precision. For example, a million in base-10 notation appears as +1.00000 e6 and -0.0000383 is represented as -3.83000 e-5. A non-zero number can always be written in *normalized form*—that is, with a leading non-zero digit immediately before the decimal point. Thus, a normalized significand in base-10 notation is a number in the range [1,10). The signed exponent specifies the number of positions that the decimal point is shifted.

Unlike the common engineering and scientific usage described above, SSE floating-point numbers are represented in base (radix) 2. Like its base-10 counterpart, a normalized base-2 significand is written with its leading non-zero digit immediately to the left of the radix point. In base-2 arithmetic, a non-zero digit is always a one, so the range of a binary significand is [1,2):

```
+1.fraction ±exponent
```

The leading non-zero digit is called the *integer bit*. As shown in Figure 4-4 on page 123, the integer bit is omitted (and called the *hidden integer bit)* in the single-precision and the double-precision floating-point formats, because its implied value is always 1 in a normalized significand (0 in a denormalized significand), and the omission allows an extra bit of precision.

Floating-Point Representations

The following sections describe the number representations.

Normalized Numbers. Normalized floating-point numbers are the most frequent operands for SSE instructions. These are finite, non-zero, positive or negative numbers in which the integer bit is 1, the biased exponent is non-zero and non-maximum, and the fraction is any representable value. Thus, the significand is within the range of [1, 2). Whenever possible, the processor represents a floating-point result as a normalized number.

Denormalized (Tiny) Numbers. Denormalized numbers (also called *tiny* numbers) are smaller than the smallest representable normalized numbers. They arise through an underflow condition, when the exponent of a result lies below the representable minimum exponent. These are finite, non-zero, positive or negative numbers in which the integer bit is 0, the biased exponent is 0, and the fraction is non-zero.

The processor generates a denormalized-operand exception (DE) when an instruction uses a denormalized *source operand*. The processor may generate an underflow exception (UE) when an instruction produces a rounded, non-zero *result* that is too small to be represented as a normalized floating-point number in the destination format, and thus is represented as a denormalized number. If a

result, after rounding, is too small to be represented as the minimum denormalized number, it is represented as zero. (See “Exceptions” on page 218 for specific details.)

Denormalization may correct the exponent by placing leading zeros in the significand. This may cause a loss of precision, because the number of significant bits in the fraction is reduced by the leading zeros. In the single-precision floating-point format, for example, normalized numbers have biased exponents ranging from 1 to 254 (the unbiased exponent range is from –126 to +127). A true result with an exponent of, say, –130, undergoes denormalization by right-shifting the significand by the difference between the normalized exponent and the minimum exponent, as shown in Table 4-4 below.

Table 4-4. Example of Denormalization

Significand (base 2)	Exponent	Result Type
1.0011010000000000	–130	True result
0.0001001101000000	–126	Denormalized result

Zero. The floating-point zero is a finite, positive or negative number in which the integer bit is 0, the biased exponent is 0, and the fraction is 0. The sign of a zero result depends on the operation being performed and the selected rounding mode. It may indicate the direction from which an underflow occurred, or it may reflect the result of a division by +∞ or –∞.

Infinity. Infinity is a positive or negative number, +∞ and –∞, in which the integer bit is 1, the biased exponent is maximum, and the fraction is 0. The infinities are the maximum numbers that can be represented in floating-point format. Negative infinity is less than any finite number and positive infinity is greater than any finite number (i.e., the affine sense).

An infinite result is produced when a non-zero, non-infinite number is divided by 0 or multiplied by infinity, or when infinity is added to infinity or to 0. Arithmetic on infinities is exact. For example, adding any floating-point number to +∞ gives a result of +∞. Arithmetic comparisons work correctly on infinities. Exceptions occur only when the use of an infinity as a source operand constitutes an invalid operation.

Not a Number (NaN). NaNs are non-numbers, lying outside the range of representable floating-point values. The integer bit is 1, the biased exponent is maximum, and the fraction is non-zero. NaNs are of two types:

- *Signaling NaN (SNaN)*
- *Quiet NaN (QNaN)*

A QNaN is a NaN with the most-significant fraction bit set to 1, and an SNaN is a NaN with the most-significant fraction bit cleared to 0. When the processor encounters an SNaN as a source operand for an instruction, an invalid-operation exception (IE) occurs and a QNaN is produced as the result, if the exception is masked. In general, when the processor encounters a QNaN as a source operand for an instruction, the processor does not generate an exception but generates a QNaN as the result.

The processor never generates an SNaN as a result of a floating-point operation. When an invalid-operation exception (IE) occurs due to an SNaN operand, the invalid-operation exception mask (IM) bit determines the processor's response, as described in "SIMD Floating-Point Exception Masking" on page 226.

When a floating-point operation or exception produces a QNaN result, its value is determined by the rules in Table 4-5 below.

Table 4-5. NaN Results

Source Operands (in either order)		NaN Result[1]
QNaN	Any non-NaN floating-point value, or single-operand instructions	Value of QNaN
SNaN	Any non-NaN floating-point value, or single-operand instructions	Value of SNaN converted to a QNaN[2]
QNaN	QNaN	Value of operand 1
QNaN	SNaN	
SNaN	QNaN	Value of operand 1 converted to a QNaN, if necessary[2]
SNaN	SNaN	
Invalid-Operation Exception (IE) occurs without QNaN or SNaN source operands		Floating-point indefinite value[3] (a special form of QNaN)

Note:
1. *The NaN result is produced when the floating-point invalid-operation exception is masked.*
2. *The conversion is done by changing the most-significant fraction bit to 1.*
3. *See "Indefinite Values" on page 128.*

Floating-Point Number Encodings

Supported Encodings. Table 4-6 below shows the floating-point encodings of supported numbers and non-numbers. The number categories are ordered from large to small. In this affine ordering, positive infinity is larger than any positive normalized number, which in turn is larger than any positive denormalized number, which is larger than positive zero, and so forth. Thus, the ordinary rules of comparison apply between categories as well as within categories, so that comparison of any two numbers is well-defined.

The actual exponent field length is 8 or 11 bits, and the fraction field length is 23 or 52 bits, depending on operand precision. The single-precision and double-precision formats do not include the integer bit in the significand (the value of the integer bit can be inferred from number encodings). Exponents of both types are encoded in biased format, with respective biasing constants of 127 and 1023.

Table 4-6. Supported Floating-Point Encodings

Classification		Sign	Biased Exponent[1]	Significand[2]
Positive Non-Numbers	SNaN	0	111 ... 111	1.011 ... 111 to 1.000 ... 001
	QNaN	0	111 ... 111	1.111 ... 111 to 1.100 ... 000
Positive Floating-Point Numbers	Positive Infinity (+∞)	0	111 ... 111	1.000 ... 000
	Positive Normal	0	111 ... 110 to 000 ... 001	1.111 ... 111 to 1.000 ... 000
	Positive Denormal	0	000 ... 000	0.111 ... 111 to 0.000 ... 001
	Positive Zero	0	000 ... 000	0.000 ... 000
Negative Floating-Point Numbers	Negative Zero	1	000 ... 000	0.000 ... 000
	Negative Denormal	1	000 ... 000	0.000 ... 001 to 0.111 ... 111
	Negative Normal	1	000 ... 001 to 111 ... 110	1.000 ... 000 to 1.111 ... 111
	Negative Infinity (–∞)	1	111 ... 111	1.000 ... 000
Negative Non-Numbers	SNaN	1	111 ... 111	1.000 ... 001 to 1.011 ... 111
	QNaN[3]	1	111 ... 111	1.100 ... 000 to 1.111 ... 111

Note:

1. *The actual exponent field length is 8 or 11 bits, depending on operand precision.*
2. *The "1." and "0." prefixes represent the implicit integer bit. The actual fraction field length is 23 or 52 bits, depending on operand precision.*
3. *The floating-point indefinite value is a QNaN with a negative sign and a significand whose value is 1.100 ... 000.*

Indefinite Values. Floating-point and integer data type each have a unique encoding that represents an *indefinite value*. The processor returns an indefinite value when a masked invalid-operation exception (IE) occurs.

For example, if a floating-point division operation is attempted using source operands that are both zero, and IE exceptions are masked, the floating-point indefinite value is returned as the result. Or, if a

floating-point-to-integer data conversion overflows its destination integer data type, and IE exceptions are masked, the integer indefinite value is returned as the result.

Table 4-7 shows the encodings of the indefinite values for each data type. For floating-point numbers, the indefinite value is a special form of QNaN. For integers, the indefinite value is the largest representable negative twos-complement number, 80...00h. (This value is the largest representable negative number, except when a masked IE exception occurs, in which case it is generated as the indefinite value.)

Table 4-7. Indefinite-Value Encodings

Data Type	Indefinite Encoding
Single-Precision Floating-Point	FFC0_0000h
Double-Precision Floating-Point	FFF8_0000_0000_0000h
16-Bit Integer	8000h
32-Bit Integer	8000_0000h
64-Bit Integer	8000_0000_0000_0000h

Floating-Point Rounding

The floating-point rounding control (RC) field comprises bits [14:13] of the MXCSR. This field which specifies how the results of floating-point computations are rounded. Rounding modes apply to most arithmetic operations. When rounding occurs, the processor generates a precision exception (PE). Rounding is not applied to operations that produce NaN results.

The IEEE 754 standard defines the four rounding modes as shown in Table 4-8 below.

Table 4-8. Types of Rounding

RC Value	Mode	Type of Rounding
00 (default)	Round to nearest	The rounded result is the representable value closest to the infinitely precise result. If equally close, the even value (with least-significant bit 0) is taken.
01	Round down	The rounded result is closest to, but no greater than, the infinitely precise result.
10	Round up	The rounded result is closest to, but no less than, the infinitely precise result.
11	Round toward zero	The rounded result is closest to, but no greater in absolute value than, the infinitely precise result.

Round to nearest is the default rounding mode. It provides a statistically unbiased estimate of the true result, and is suitable for most applications. The other rounding modes are directed roundings: round up (toward +∞), round down (toward –∞), and round toward zero. Round up and round down are used in interval arithmetic, in which upper and lower bounds bracket the true result of a computation. Round toward zero takes the smaller in magnitude, that is, always truncates.

The processor produces a floating-point result defined by the IEEE standard to be infinitely precise. This result may not be representable exactly in the destination format, because only a subset of the continuum of real numbers finds exact representation in any particular floating-point format. Rounding modifies such a result to conform to the destination format, thereby making the result inexact and also generating a precision exception (PE), as described in “SIMD Floating-Point Exception Causes” on page 220.

Suppose, for example, the following 24-bit result is to be represented in single-precision format, where “E_2 1010” represents the biased exponent:

$$1.0011\ 0101\ 0000\ 0001\ 0010\ 0111\ E_2\ 1010$$

This result has no exact representation, because the least-significant 1 does not fit into the single-precision format, which allows for only 23 bits of fraction. The rounding control field determines the direction of rounding. Rounding introduces an error in a result that is less than one *unit in the last place (ulp)*, that is, the least-significant bit position of the floating-point representation.

Half-Precision Floating-Point Data Type

The architecture supports a half-precision floating-point data type. This representation requires only 16 bits and is used primarily to save space when floating-point values are stored in memory. One instruction converts packed half-precision floating-point numbers loaded from memory to packed single-precision floating-point numbers and another converts packed single-precision numbers in a YMM/XMM register to packed half-precision numbers in preparation for storage. See Section 4.7.2.5 “Half-Precision Floating-Point Conversion” on page 193 for more information on these instructions.

The 16-bit floating-point data type, shown in Figure 4-5, includes a 1-bit sign, a 5-bit exponent with a bias of 15 and a 10-bit significand. The integer bit is implied, making a total of 11 bits in the significand. The value of the integer bit can be inferred from the number encoding. Table 4-9 on page 131 shows the floating-point encodings of supported numbers and non-numbers.

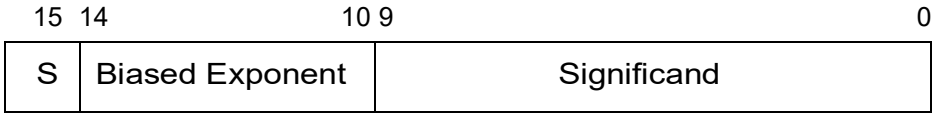


Figure 4-5. 16-Bit Floating-Point Data Type

Table 4-9. Supported 16-Bit Floating-Point Encodings

Sign	Bias Exponent	Significand[a]	Classification	
0	1 1111	1.00 0000 0000	Positive Floating-Point Numbers	Positive Infinity
0	1 1110 to 0 0001	1.11 1111 1111 to 1.00 0000 0000		Positive Normal
0	0 0000	0.11 1111 1111 to 0.00 0000 0001		Positive Denormal
0	0 0000	0.00 0000 0000		Positive Zero
1	0 0000	0.00 0000 0000	Negative Floating-Point Numbers	Negative Zero
1	0 0000	0.00 0000 0001 to 0.11 1111 1111		Negative Denormal
1	0 0001 to 1 1110	1.00 0000 0000 to 1.11 1111 1111		Negative Normal
1	1 1111	1.00 0000 0000		Negative Infinity
X	1 1111	1.00 0000 0001 to 1.01 1111 1111	Non-Number	SNaN
X	1 1111	1.10 0000 0000 to 1.11 1111 1111		QNaN

a. *The "1." and "0." prefixes represent the implicit integer bit.*

4.3.3.4 Vector and Scalar Data Types

Most SSE instructions accept vector or scalar operands. These data types are composites of the fundamental data types discussed above. The following data types are supported:

- Vector (packed) single-precision (32-bit) floating-point numbers
- Vector (packed) double-precision (64-bit) floating-point numbers
- Vector (packed) signed (two's-complement) integers
- Vector (packed) unsigned integers
- Scalar single- and double-precision floating-point numbers
- Scalar signed (two's-complement) integers
- Scalar unsigned integers

Hardware does not check or enforce the data types for instructions. Software is responsible for ensuring that each operand for an instruction is of the correct data type. If data produced by a previous instruction is of a type different from that used by the current instruction, and the current instruction

sources such data, the current instruction may incur a latency penalty, depending on the hardware implementation.

For the sake of identifying a specific element within a vector (packed) data type, the elements are numbered from right to left starting with 0 and ending with *(vector_size/element_size) −1*. Some instructions operate on even and odd pairs of elements. The even elements are (0, 2, 4 ...) and the odd elements are (1,3,5 ...).

Software can interpret data in ways other than those listed —such fixed-point or fractional numbers— but the SSE instructions do not directly support such interpretations and software must handle them entirely on its own.

128-bit Vector Data Types

Figure 4-6 below illustrates the 128-bit vector data types.

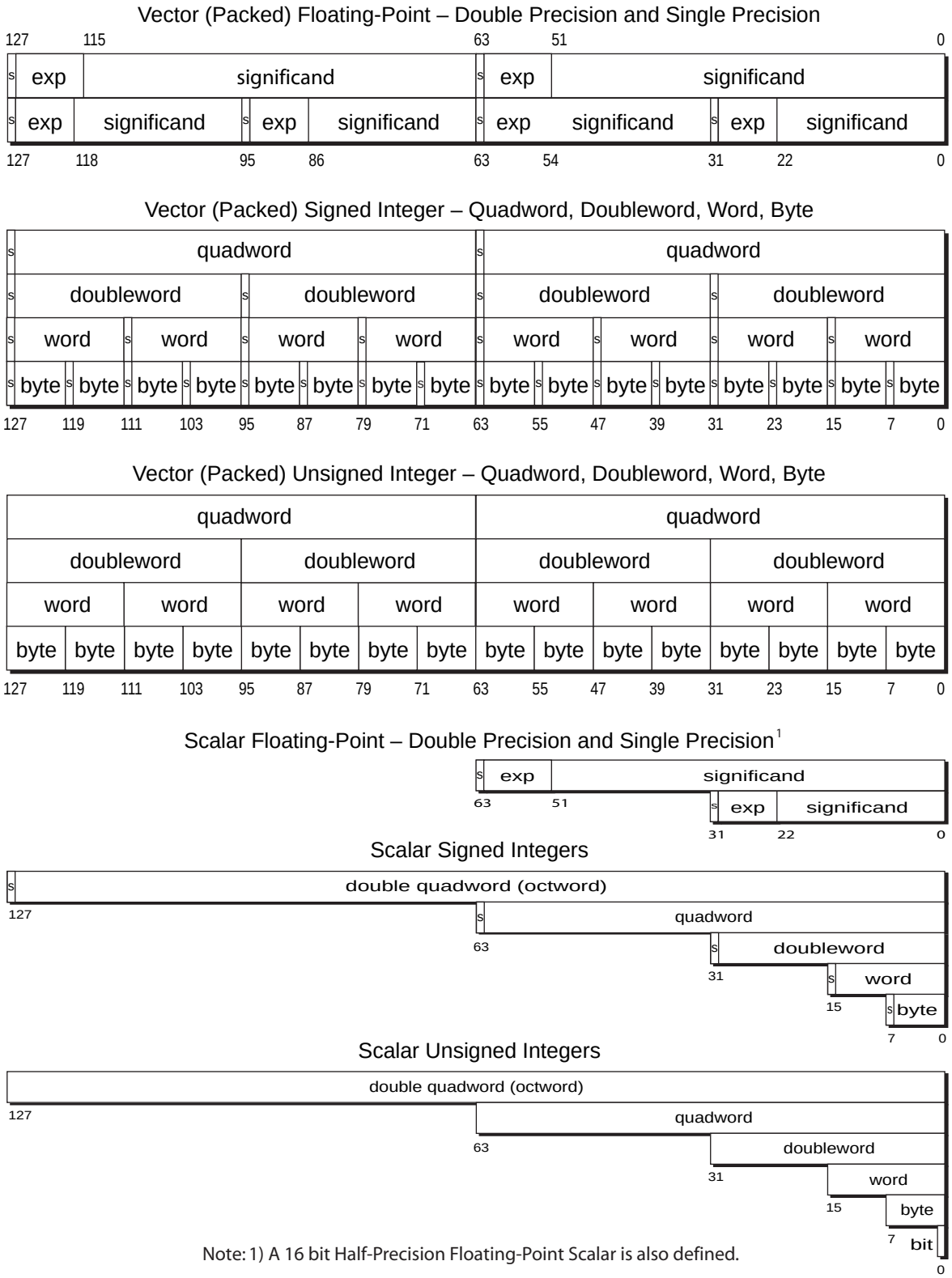


Figure 4-6. 128-Bit Media Data Types

256-bit Vector Data Types

Figure 4-7 and Figure 4-8 below illustrate the 256-bit vector data types.

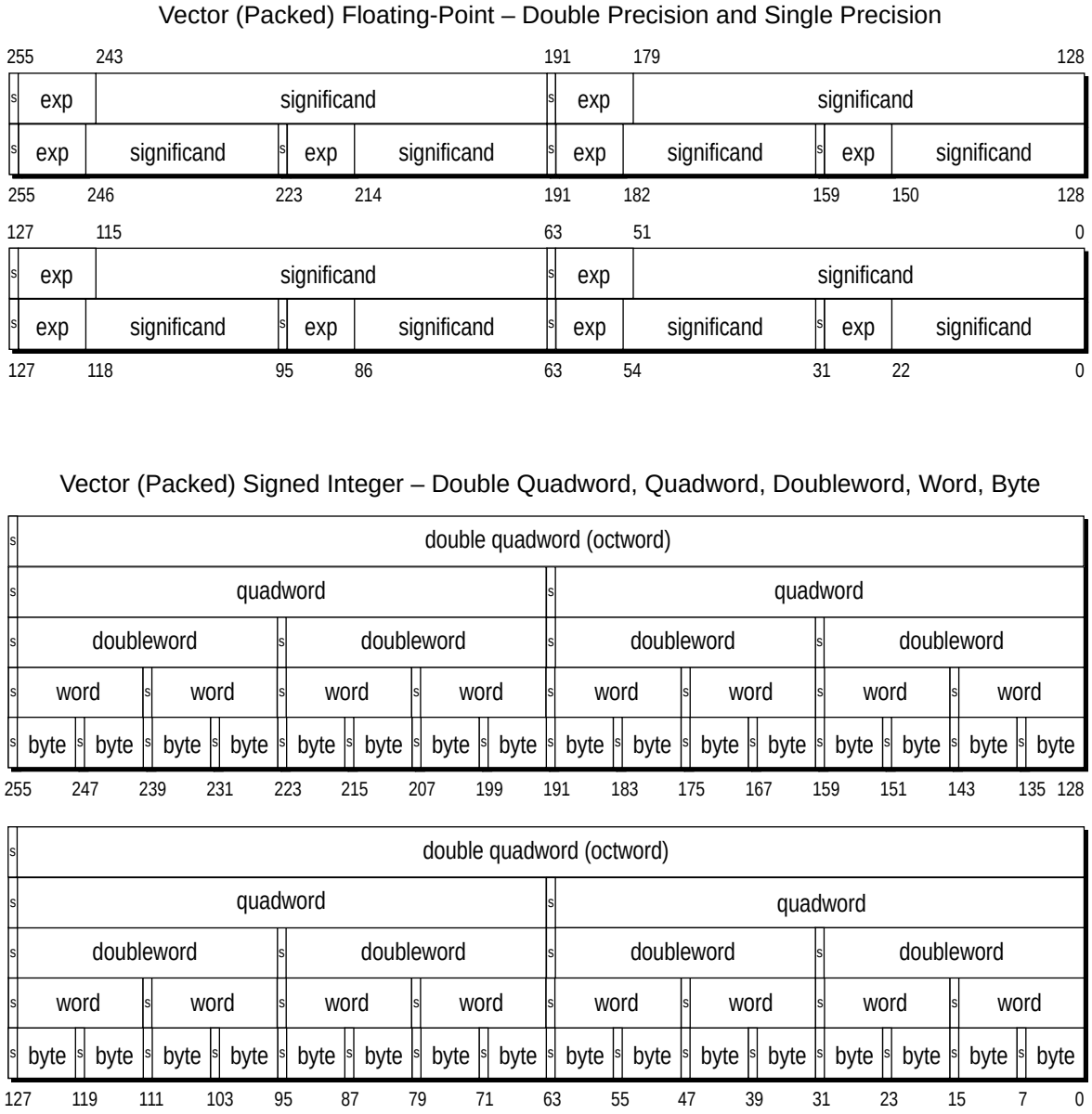
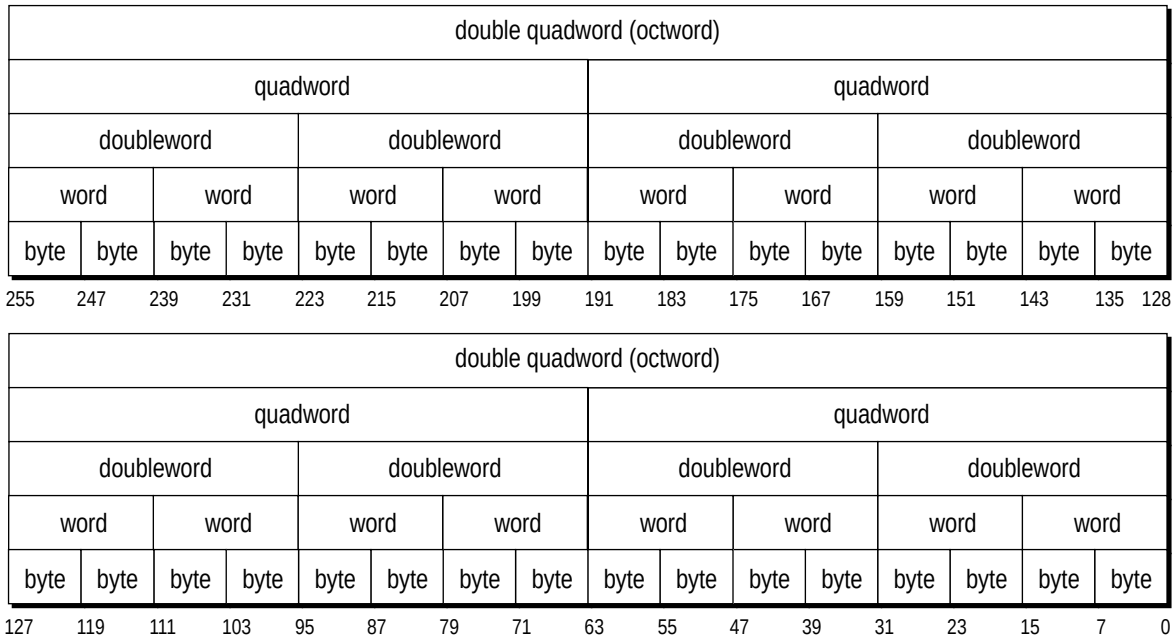
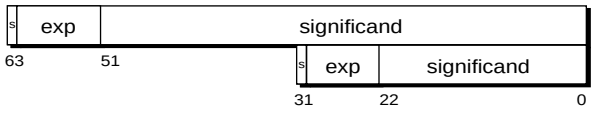


Figure 4-7. 256-Bit Media Data Types

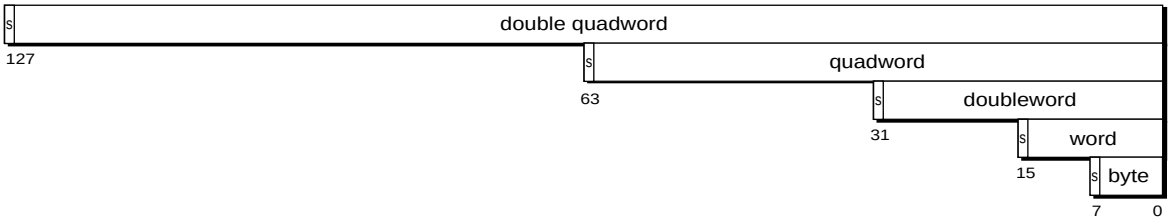
Vector (Packed) Unsigned Integer – Double Quadword, Quadword, Doubleword, Word, Byte



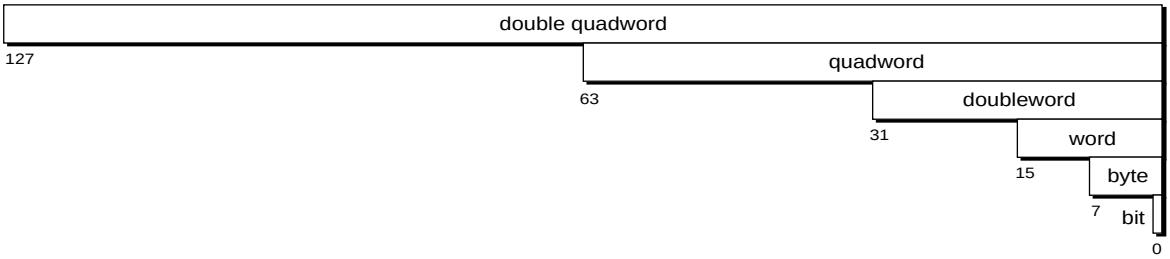
Scalar Floating-Point – Double Precision and Single Precision[1]



Scalar Signed Integers



Scalar Unsigned Integers



Note: 1) A 16 bit Half-Precision Floating-Point Scalar is also defined.

Figure 4-8. 256-Bit Media Data Types (Continued)

Software can interpret the data types in ways other than those shown—such as bit fields or fractional numbers—but the instructions do not directly support such interpretations and software must handle them entirely on its own.

4.3.4 Operand Sizes and Overrides

Operand sizes for SSE instructions are determined by instruction opcodes. Some of these opcodes include an operand-size override prefix, but this prefix acts in a special way to modify the opcode and is considered an integral part of the opcode. The general use of the 66h operand-size override prefix described in “Instruction Prefixes” on page 76 does not apply to SSE instructions.

For details on the use of operand-size override prefixes in SSE instructions, see “*Volume 4: 128-Bit and 256-Bit Media Instructions*”.

4.4 Vector Operations

4.4.1 Integer Vector Operations

Figure 4-9 below shows an example of a typical two operand integer vector operation. In this example, each n-bit wide vector contains 16 integer elements. Note that the same mathematical operation is performed on all 16 elements in parallel. The computation of one element of the result vector does not affect the computation of any of the other result elements. For example, a carry out that could occur as a result of computing a sum is not added into the sum of the next most significant element of the vector.

In general, the result of a vector operation is a vector of the same width as the operands with the same number of elements. (Although there are instructions which increase or decrease the width and number of elements in the result.) There are instructions that operate on vectors of words, doublewords, quadwords and octwords. Both 128-bit and 256-bit wide vectors are supported. See Section 4.6 “Instruction Summary—Integer Instructions” on page 149 for more information on the supported 128-bit and 256-bit integer data types.

Most legacy SSE instructions support the specification of two operands. For these instructions the result overwrites the first operand as shown. The extended SSE set includes instructions that support two, three, or four vector operands. In these instructions, the result is generally written to a destination register specified by the instruction encoding.

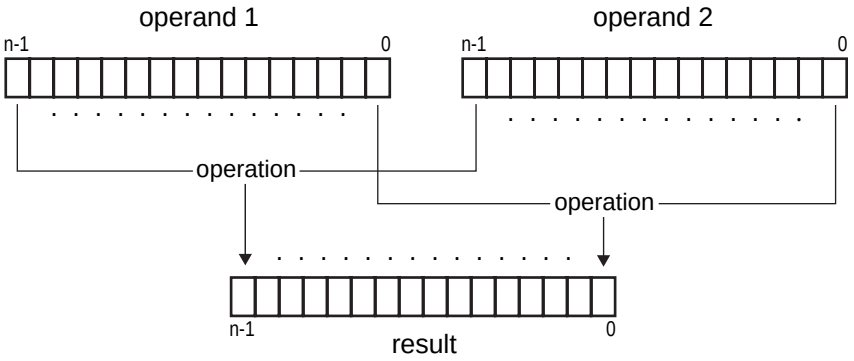


Figure 4-9. Mathematical Operations on Integer Vectors

The SSE instruction set also supports a vector form of the unary arithmetic operation *absolute value*. In these instructions the absolute value operation is applied independently to all the elements of the source operand to produce the result.

4.4.2 Floating-Point Vector Operations

The SSE instruction set supports vectors of both single-precision and double-precision floating-point values in both 128-bit and 256-bit vector widths. See Section 4.6 “Instruction Summary—Integer Instructions” on page 149 for more information on the supported 128-bit and 256-bit data types.

Figure 4-10 shows an example of a parallel operation on two 256-bit vectors, each containing four 64-bit double-precision floating-point values. As in the integer vector operation, each element of the vector result is the product of the mathematical operation applied to corresponding elements of the source operands. The number of elements and parallel operations is 2, 4, or 8 depending on vector and element size.

Some SSE floating-point instructions support the specification of only two operands. For most of these instructions the result overwrites the first operand. The extended SSE instructions include instructions that support three or four operands. In most three and four operand instructions, the result is written to a separate destination register specified by the instruction encoding.

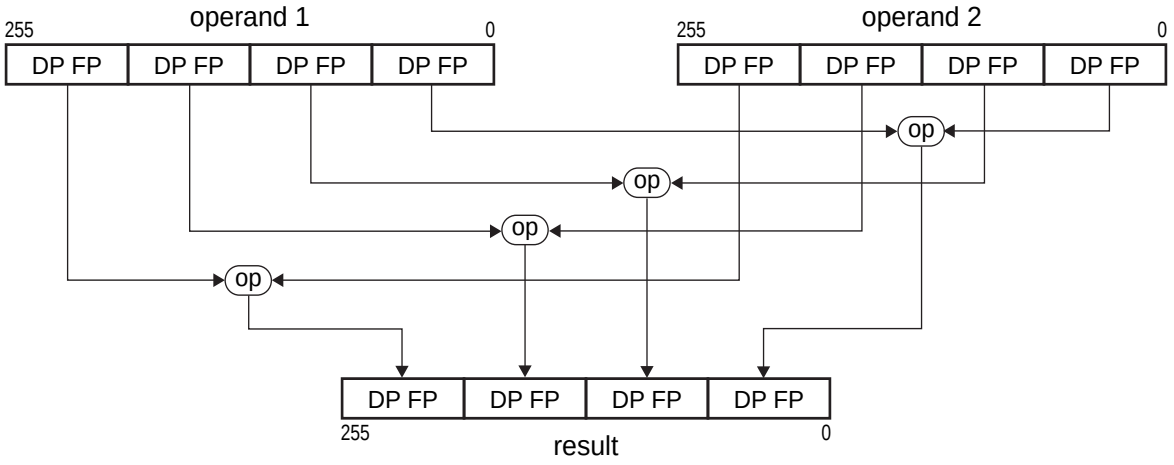


Figure 4-10. Mathematical Operations on Floating-Point Vectors

Integer and floating-point instructions can be freely intermixed in the same procedure. The floating-point instructions allow media applications such as 3D graphics to accelerate geometry, clipping, and lighting calculations. Pixel data are typically integer-based, although both integer and floating-point instructions are often required to operate completely on the data. For example, software can change the viewing perspective of a 3D scene through transformation matrices by using floating-point instructions in the same procedure that contains integer operations on other aspects of the graphics data.

For media and scientific programs that demand floating-point operations, it is often easier and more powerful to use SSE instructions. Such programs perform better than x87 floating-point programs, because the YMM/XMM register file is flat rather than stack-oriented, there are twice as many registers (in 64-bit mode), and SSE instructions can operate on four or eight times the number of floating-point operands as can x87 instructions. This ability to operate in parallel on multiple pairs of floating-point elements often makes it possible to remove local temporary variables that would otherwise be needed in x87 floating-point code.

4.5 Instruction Overview

4.5.1 Instruction Syntax

Each instruction has a *mnemonic syntax* used by assemblers to specify the operation and the operands to be used for source and destination (result) data.

Legacy SSE Instructions

The legacy SSE instructions accept two operands and generally have the following syntax:

```
MNEMONIC xmm1, xmm2/mem128
```

Figure 4-11 below shows an example of the mnemonic syntax for a packed add bytes (PADDB) instruction.

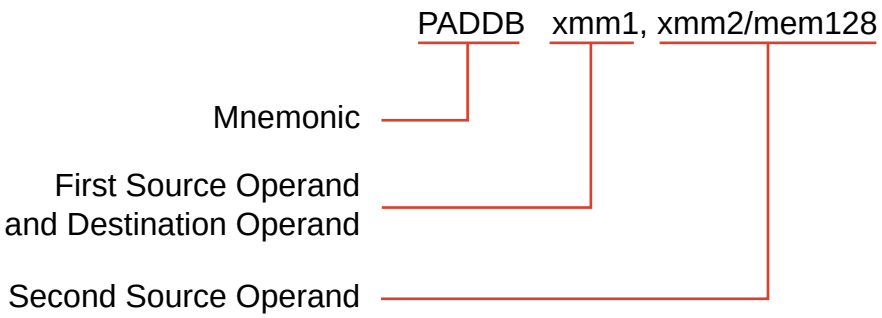


Figure 4-11. Mnemonic Syntax for Typical Legacy SSE Instruction

This example shows the PADDB mnemonic followed by two operands, a 128-bit XMM register operand and another 128-bit XMM register or 128-bit memory operand. In most instructions that take two operands, the first (left-most) operand is both a source operand and the destination operand. The second (right-most) operand serves only as a source. Some instructions can have one or more prefixes that modify default properties, as described in “Instruction Prefixes” on page 217.

Extended SSE Instructions

The extended SSE instructions support operands of either 128 bits or 256 bits. They also support the specification of two, three, four, or five operands sourced from YMM/XMM registers, memory or immediate bytes. A three-operand 128-bit extended SSE instruction has the following syntax:

```
MNEMONIC xmm1, xmm2, xmm3/mem128
```

A three-operand 256-bit extended SSE instruction has the following syntax:

```
MNEMONIC ymm1, ymm2, ymm3/mem256
```

Figure 4-12 below shows an example of the mnemonic syntax for the packed add bytes (VPADDB) instruction.

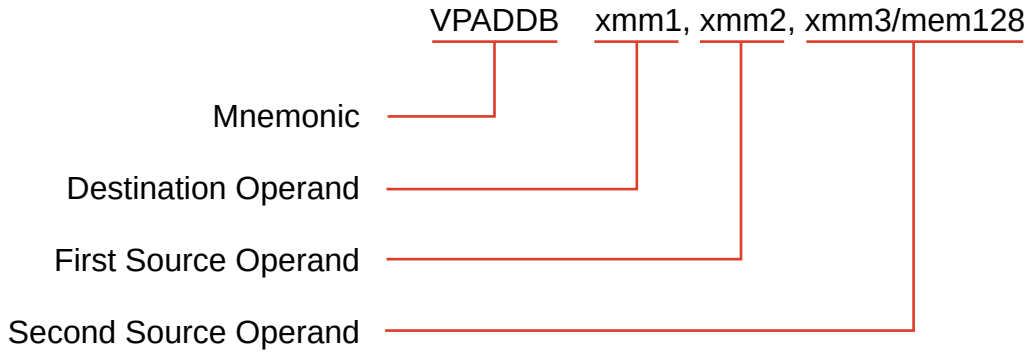


Figure 4-12. Mnemonic Syntax for Typical Extended SSE Instruction

This example shows the VPADDB mnemonic followed by three operands—a destination XMM register and two source operands. Instruction operand number 2 located in an XMM register is actually the first source operand and operand 3 is the second source operand. The second source operand may be located in an XMM register or in memory. The result of the vector add operation is placed in the specified destination register. Some instructions can have one or more prefixes that modify default properties, as described in “Instruction Prefixes” on page 217.

4.5.2 Mnemonics

Most mnemonics follow some general conventions:

As noted above, a **V** prepended to a mnemonic means that it is an extended SEE instruction.

The initial character string of the mnemonic (immediately after the possibly prepended **V)** represents the operation the instruction performs. An initial **P** in the string representing the operation stands for “Packed.” Subsequent character strings in various combinations either refer to operand types or indicate a variant of the basic operation. The following lists most of these conventions:

- **A**—Aligned
- **B**—Byte
- **D**—Doubleword
- **DQ**—Double quadword
- **HL**—High to low
- **LH**—Low to high
- **L**—Left
- **PD**—Packed double-precision floating-point
- **PI**—Packed integer
- **PS**—Packed single-precision floating-point
- **Q**—Quadword
- **R**—Right
- **S**—Signed, or Saturation, or Shift
- **SD**—Scalar double-precision floating-point
- **SI**—Signed integer
- **SS**—Scalar single-precision floating-point, or Signed saturation
- **U**—Unsigned, or Unordered, or Unaligned
- **US**—Unsigned saturation
- **V**—Intial letter designates an extended SSE instruction
- **W**—Word
- **2**—to. Used in data type conversion instruction mnemonics.

Consider the example VPMULHUW. The initial **V** indicates that the instruction is an extended SSE instruction (in this case, an AVX instruction). It is a packed (that is, vector) multiply (**P** for packed and **MUL** for multiply) of unsigned words (**U** for unsigned and **W** for Word). Finally, the **H** refers to the fact that the high word of each intermediate double word result is written to the destination vector element.

4.5.3 Move Operations

Move instructions—along with unpack instructions—are among the most frequently used instructions in media procedures.

When moving between XMM registers, or between an XMM register and memory, each integer move instruction can copy up to 16 bytes of data. When moving between an XMM register and an MMX or GPR register, an integer move instruction can move up to 8 bytes of data. The packed floating-point move instructions can copy vectors of four single-precision or two double-precision floating-point operands in parallel.

Figure 4-13 below provides an overview of the basic move operations involving the XMM registers. Crosshatching in the figure represents bits in the destination register which may either be zero-extended or left unchanged in the move operation based on the instruction or (for one instruction) the source of the data. Data written to memory is never zero-extended. The AVX subset provides a number of 3-operand variants of the basic move instructions that merge additional data from a XMM register into the destination register. These are not shown in this figure nor are those instructions that extend fields in the destination register by duplicating bits from the source register.

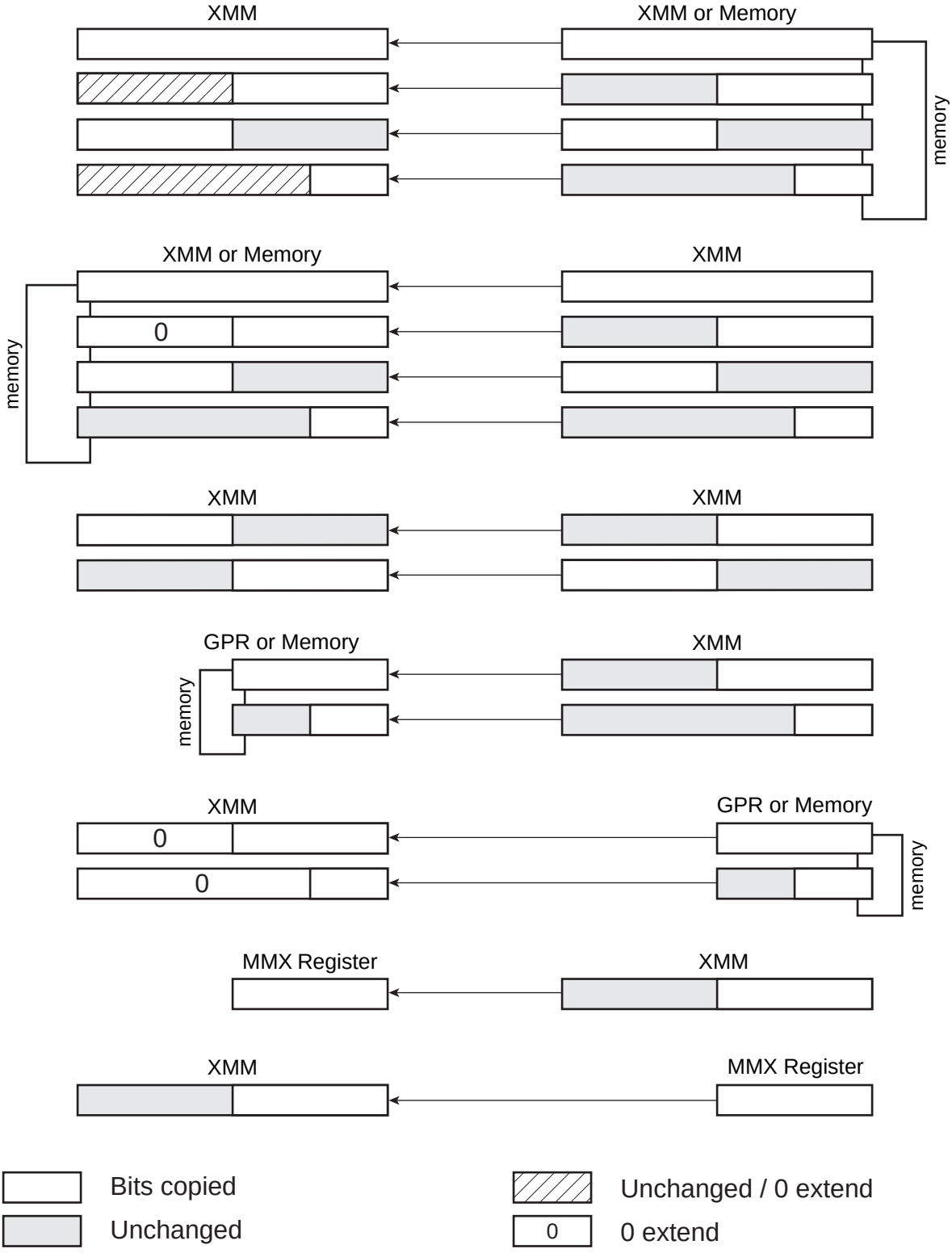


Figure 4-13. XMM Move Operations

The extended SSE instruction set provides instructions that load a YMM register from memory, store the contents of a YMM register to memory, or move 256 bits from one register to another. Figure 4-14 below provides a schematic representation of these operations.

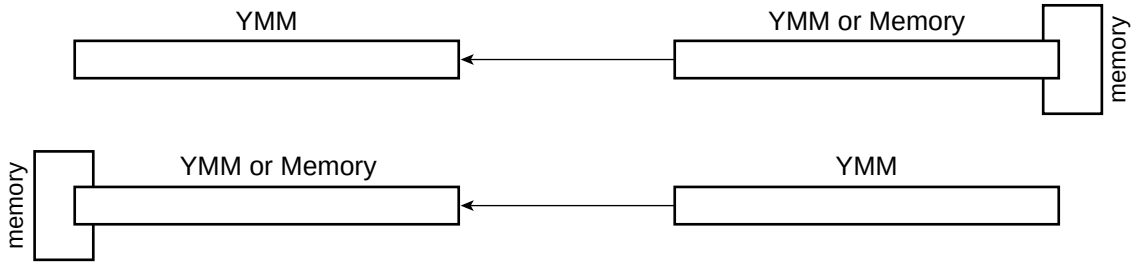


Figure 4-14. YMM Move Operations

Streaming-store versions of the move instructions (also known as non-temporal moves) bypass the cache when storing data that is accessed only once. This maximizes memory-bus utilization and minimizes cache pollution.

The move-mask instruction stores specific bytes from one vector, as selected by mask values in a second vector. Figure 4-15 below shows the (V)MASKMOVDQU operation. It can be used, for example, to handle end cases in block copies and block fills based on streaming stores.

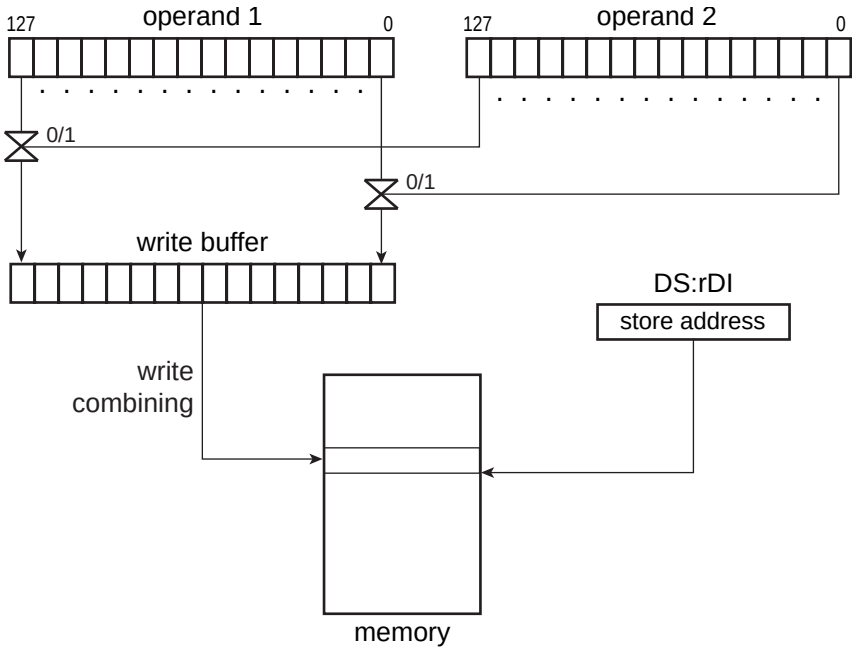


Figure 4-15. Move Mask Operation

4.5.4 Data Conversion and Reordering

SSE instructions support data conversion of vector elements, including conversions between integer and floating-point data types—located in YMM/XMM registers, MMX™ registers, GPR registers, or memory—and conversions of element-ordering or precision.

For example, the unpack instructions take two vector operands and interleave their low or high elements. Figure 4-16 shows an unpack and interleave operation on word-sized elements. This this case, (V)PUNPCKLWD. If operand 1 is a vector of unsigned integers and the left-hand source operand has elements whose value is zero, the operation converts each element in the low half of operand 1 to an integer data type of twice its original width. This would be a useful step prior to multiplying two integer vectors together to ensure that no overflow can occur during a vector multiply operation.

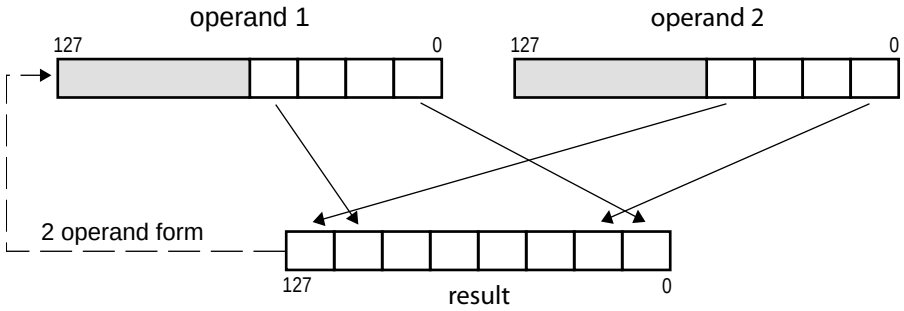


Figure 4-16. Unpack and Interleave Operation

There are also pack instructions, such as (V)PACKSSDW shown below in Figure 4-17, that convert each element in a pair of integer vectors to lower precision and pack them into the result vector.

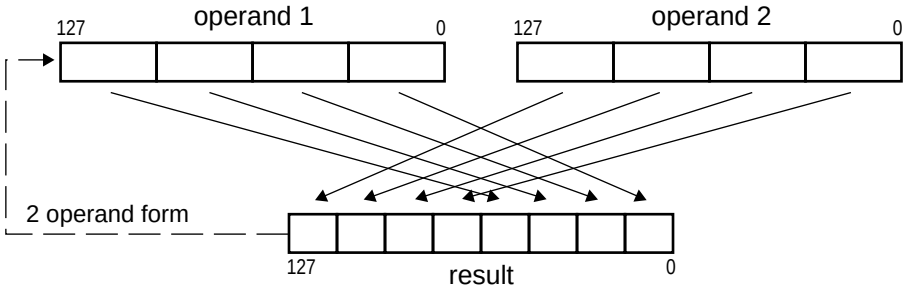


Figure 4-17. Pack Operation

Vector-shift instructions are also provided. These instructions may be used to scale each element in an integer vector up or down.

Figure 4-18 shows one of many types of shuffle operations; in this case, PSHUFD. Here the second operand is a vector containing doubleword elements, and an immediate byte provides shuffle control for up to 256 permutations of the elements. Shuffles are useful, for example, in color imaging when

computing alpha saturation of RGB values. In this case, a shuffle instruction can replicate an alpha value in a register so that parallel comparisons with three RGB values can be performed.

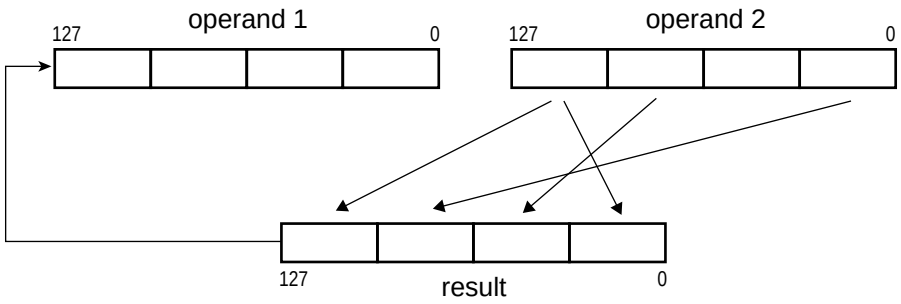


Figure 4-18. Shuffle Operation

The (V)PINSRB, (V)PINSRW, (V)PINSRD, and (V)PINSRQ instructions insert a byte, word, doubleword or quadword from a general-purpose register or memory into an XMM register, at a specified location. The legacy instructions leave the other elements in the XMM register unmodified. The extended instructions fill in the other elements from a second XMM source operand.

4.5.5 Matrix and Special Arithmetic Operations

The instruction set provides a broad assortment of vector add, subtract, multiply, divide, and square-root operations for use on matrices and other data structures common to media and scientific applications. It also provides special arithmetic operations including multiply-add, average, sum-of-absolute differences, reciprocal square-root, and reciprocal estimation.

SSE integer and floating-point instructions can perform several types of matrix-vector or matrix-matrix operations, such as addition, subtraction, multiplication, and accumulation. Efficient matrix multiplication is further supported with instructions that can first transpose the elements of matrix rows and columns. These transpositions can make subsequent accesses to memory or cache more efficient when performing arithmetic matrix operations.

Figure 4-19 on page 146 shows a Packed Multiply and Add instruction ((V)PMADDWD) which multiplies vectors of 16-bit integer elements to yield intermediate results of 32-bit elements, which are then summed pair-wise to yield four 32-bit elements. This operation can be used with one source operand (for example, a coefficient) taken from memory and the other source operand (for example, the data to be multiplied by that coefficient) taken from an XMM register. It can also be used together with a vector-add operation to accumulate *dot product* results (also called *inner* or *scalar products*), which are used in many media algorithms such as those required for finite impulse response (FIR) filters, one of the commonly used DSP algorithms.

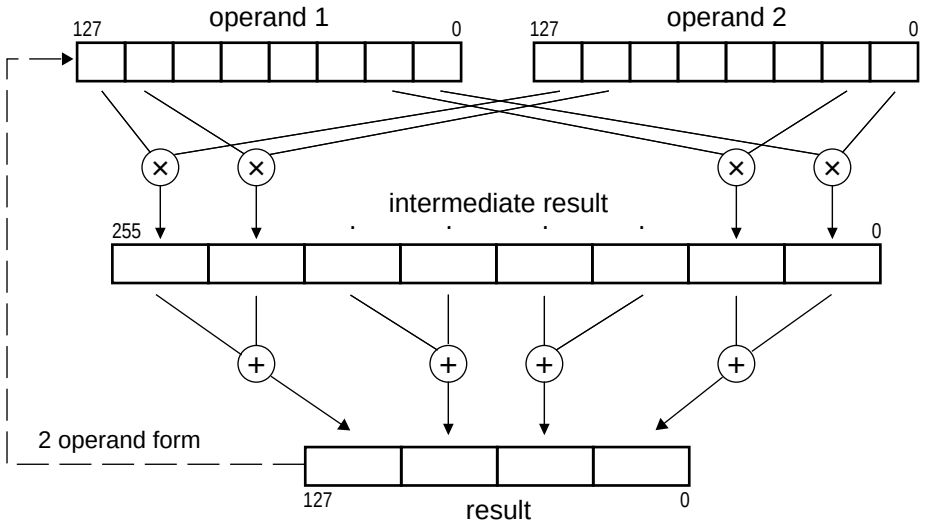


Figure 4-19. Multiply-Add Operation

See Section 4.7.5 “Fused Multiply-Add Instructions” on page 206 for a discussion of floating-point fused multiply-add instructions.

The sum-of-absolute-differences instruction ((V)PSADBW), shown in Figure 4-20 is useful, for example, in computing motion-estimation algorithms for video compression.

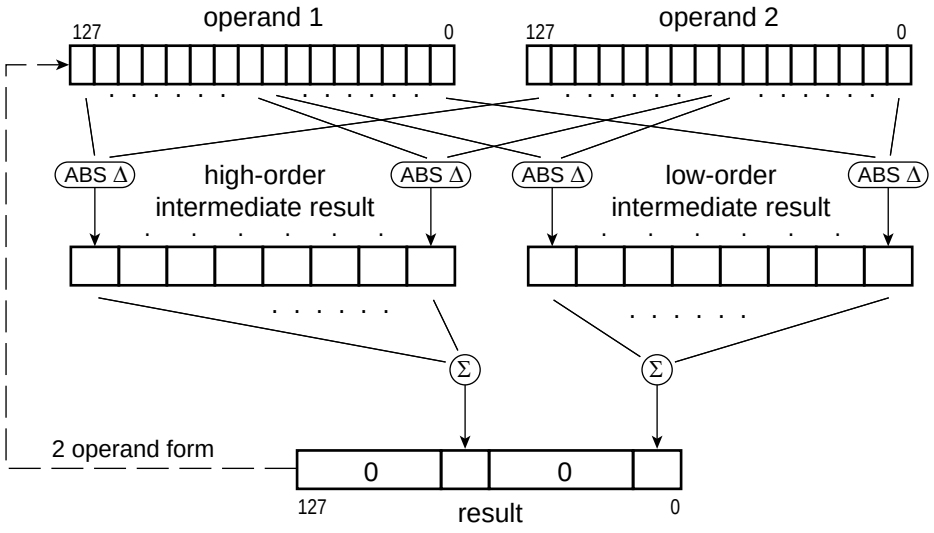


Figure 4-20. Sum-of-Absolute-Differences Operation

There is an instruction for computing the average of unsigned bytes or words. The instruction is useful for MPEG decoding, in which motion compensation involves many byte-averaging operations

between and within macroblocks. In addition to speeding up these operations, the instruction also frees up registers and makes it possible to unroll the averaging loops.

Some of the arithmetic and pack instructions produce vector results in which each element *saturates* independently of the other elements in the result vector. Such results are clamped (limited) to the maximum or minimum value representable by the destination data type when the true result exceeds that maximum or minimum representable value. Saturating data is useful for representing physical-world data, such as sound and color. It is used, for example, when combining values for pixel coloring.

4.5.6 Branch Removal

Branching is a time-consuming operation that, unlike most SSE vector operations, does not exhibit parallel behavior (there is only one branch target, not multiple targets, per branch instruction). In many media applications, a branch involves selecting between only a few (often only two) cases. Such branches can be replaced with SSE vector compare and vector logical instructions that simulate predicated execution or conditional moves.

Figure 4-21 shows an example of a non-branching sequence that implements a two-way multiplexer—one that is equivalent to the ternary operator "?:" in C and C++. The comparable code sequence is explained in "Compare and Write Mask" on page 177.

The sequence begins with a vector compare instruction that compares the elements of two source operands in parallel and produces a mask vector containing elements of all 1s or 0s. This mask vector is ANDed with one source operand and ANDed-Not with the other source operand to isolate the desired elements of both operands. These results are then ORed to select the relevant elements from each operand. A similar branch-removal operation can be done using floating-point source operands.

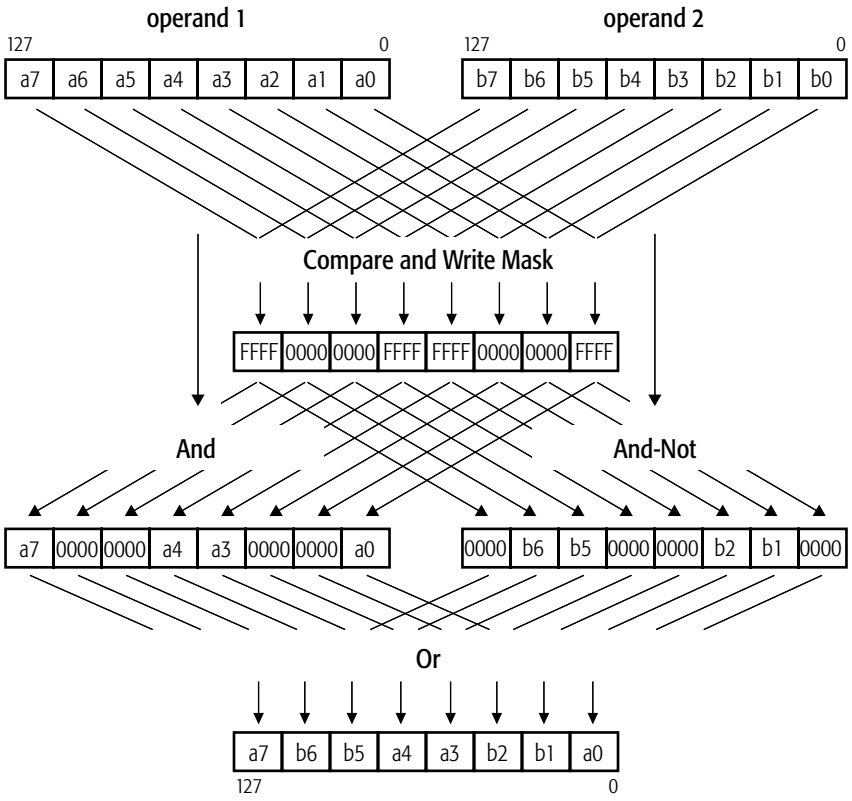


Figure 4-21. Branch-Removal Sequence

The min/max compare instructions, for example, are useful for clamping, such as color clamping in 3D graphics, without the need for branching. Figure 4-22 on page 148 illustrates a move-mask instruction ((V)PMOVMSKB) that copies sign bits to a general-purpose register (GPR). The instruction can extract bits from mask patterns, or zero values from quantized data, or sign bits—resulting in a byte that can be used for data-dependent branching.

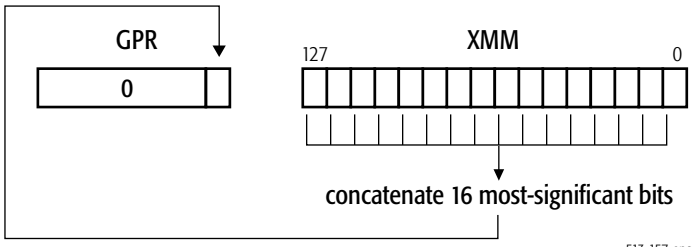


Figure 4-22. Move Mask Operation

4.6 Instruction Summary—Integer Instructions

This section summarizes the SSE instructions that operate on scalar and packed integers. Software running at any privilege level can use any of the instructions discussed below given that hardware and system software support is provided and the appropriate instruction subset is enabled. Detection and enablement of instruction subsets is normally handled by operating system software. Hardware support for each instruction subset is indicated by processor feature bits. These are accessed via the CPUID instruction. See Volume 3 for details on the CPUID instruction and the feature bits associated with the SSE instruction set.

The SSE instructions discussed below include those that use the YMM/XMM registers as well as instructions that convert data from integer to floating-point formats. For more detail on each instruction, see individual instruction reference pages in the Instruction Reference chapter of Volume 4, "*128-Bit and 256-Bit Media Instructions.*"

For a summary of the floating-point instructions including instructions that convert from floating-point to integer formats, see "Instruction Summary—Floating-Point Instructions" on page 184.

The following subsections are organized by functional groups. These are:

- Data Transfer
- Data Conversion
- Data Reordering
- Arithmetic
- Enhanced Media
- Shift and Rotate
- Compare
- Logical
- Save and Restore

Most of the instructions described below have both a legacy and an AVX form. Generally the AVX form is functionally equivalent to the legacy form except for the affect of the instruction on the upper octword of the destination YMM register. The legacy form of an instruction leaves the upper octword of the YMM register that overlays the destination XMM register unchanged, while the AVX form always clears the upper octword.

The descriptions that follow apply equally to the legacy instruction and its 128-bit AVX form. Many of the AVX instructions also support a 256-bit version of the instruction that operates on the 256-bit data types. For the instructions which accept vector operands, the only difference in functionality between the 128-bit form and the 256-bit form is the number of elements operated upon in parallel (that is, the number of elements doubles). Other differences will be noted at the end of the discussion.

4.6.1 Data Transfer

The data-transfer instructions copy data between a memory location, a YMM/XMM register, an MMX register, or a GPR. The MOV mnemonic, which stands for *move*, is a misnomer. A copy function is actually performed instead of a move. A new copy of the source value is created at the destination address, and the original copy remains unchanged at its source location.

4.6.1.1 Move

- (V)MOVD—Move Doubleword or Quadword
- (V)MOVQ—Move Quadword
- (V)MOVDQA—Move Aligned Double Quadword
- (V)MOVDQU—Move Unaligned Double Quadword
- MOVDQ2Q—Move Quadword to Quadword
- MOVQ2DQ—Move Quadword to Quadword
- (V)LDDQU—Load Double Quadword Unaligned

When copying between YMM registers, or between a YMM register and memory, a move instruction can copy up to 32 bytes of data. When copying between XMM registers, or between an XMM register and memory, a move instruction can copy up to 16 bytes of data. When copying between an XMM register and an MMX or GPR register, a move instruction can copy up to 8 bytes of data.

The (V)MOVD instruction copies a 32-bit or 64-bit value from a GPR register or memory location to the low-order 32 or 64 bits of an XMM register, or from the low-order 32 or 64 bits of an XMM register to a 32-bit or 64-bit GPR or memory location. If the source operand is a GPR or memory location, the source is zero-extended to 128 bits in the XMM register. If the source is an XMM register, only the low-order 32 or 64 bits of the source are copied to the destination.The 64-bit (long) form of (V)MOVD is aliased as (V)MOVQ.

The (V)MOVQ instruction copies a 64-bit value from memory to the low quadword of an XMM register, or from the low quadword of an XMM register to memory, or between the low quadwords of two XMM registers. If the source is in memory and the destination is an XMM register, the source is zero-extended to 128 bits in the XMM register.

The (V)MOVDQA instruction copies a 128-bit value from memory to an XMM register, or from an XMM register to memory, or between two XMM registers. If either the source or destination is a memory location, the memory address must be aligned. The (V)MOVDQU instruction does the same, except for unaligned operands. The (V)LDDQU instruction is virtually identical in operation to the (V)MOVDQU instruction. The (V)LDDQU instruction moves a double quadword of data from a 128-bit memory operand into a destination XMM register.

The VMOVDQA and VMOVDQU instructions have 256-bit forms which copy a 256-bit value from memory to a YMM register, or from a YMM register to memory, or between two YMM registers. VLDDQU has a 256-bit form that loads a 256-bit value from memory.

The MOVDQ2Q instruction copies the low-order 64-bit value in an XMM register to an MMX register. The MOVQ2DQ instruction copies a 64-bit value from an MMX register to the low-order 64 bits of an XMM register, with zero-extension to 128 bits.

Figure 4-23 below diagrams the capabilities of these instructions. (V)LDDQU and the 128-bit forms of the extended instructions are not shown.

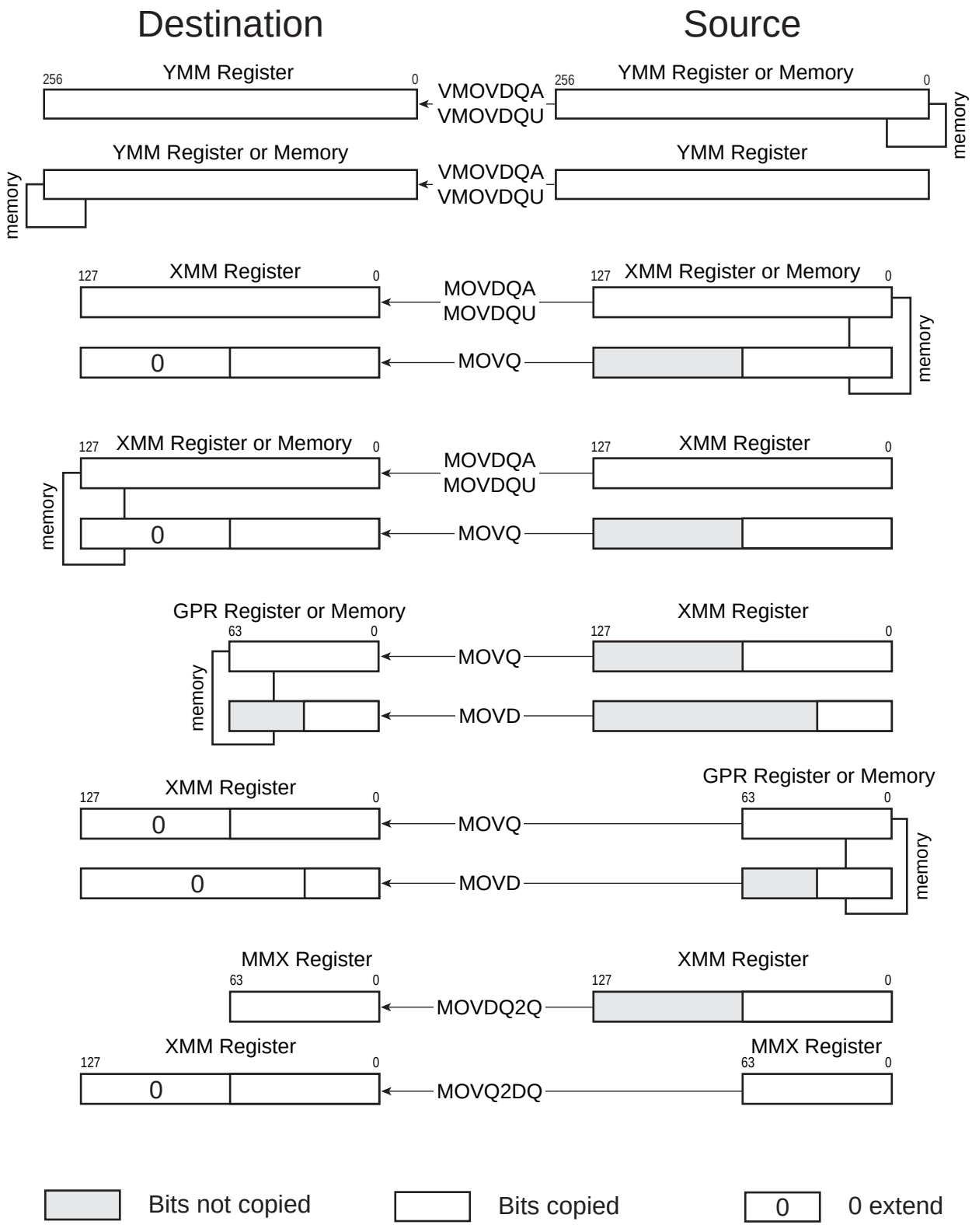


Figure 4-23. Integer Move Operations

The move instructions are in many respects similar to the assignment operator in high-level languages. The simplest example of their use is for initializing variables. To initialize a register to 0, however, rather than using a MOV*x* instruction it may be more efficient to use the (V)PXOR instruction with identical destination and source operands.

4.6.1.2 Move Non-Temporal

The move non-temporal instructions are *streaming-store* instructions. They minimize pollution of the cache.

- (V)MOVNTDQ—Move Non-temporal Double Quadword
- (V)MOVNTDQA—Move Non-temporal Double Quadword Aligned
- (V)MASKMOVDQU—Masked Move Double Quadword Unaligned

The (V)MOVNTDQ instruction stores its source operand (a 128-bit XMM register value or a 256-bit YMM register value) to a 128-bit or 256-bit memory location. (V)MOVNTDQ indicates to the processor that its data is *non-temporal*, which assumes that the referenced data will be used only once and is therefore not subject to cache-related overhead (as opposed to *temporal* data, which assumes that the data will be accessed again soon and should be cached). The non-temporal instructions use weakly-ordered, write-combining buffering of write data, and they minimize cache pollution. The exact method by which cache pollution is minimized depends on the hardware implementation of the instruction. For further information, see “Memory Optimization” on page 98 and “Use Streaming Loads and Stores” on page 233.

The MOVNTDQA instruction loads an XMM register from an aligned 128-bit memory location. VMOVNTDQA loads either an XMM or a YMM register from an aligned 128-bit or 256-bit memory location. An attempt by MOVNTDQA to read from an unaligned memory address causes a #GP or invokes the alignment checking mechanism depending on the setting of the MXCSR[MM] bit. An attempt by the VMOVNTDQA to read from an unaligned memory address causes a #GP.

(V)MASKMOVDQU is also a non-temporal instruction. It stores bytes from the first operand, as selected by the mask value in the second operand. Bytes are written to a memory location specified in the rDI and DS registers. The first and second operands are both XMM registers. The address may be unaligned. Figure 4-24 shows the (V)MASKMOVDQU operation. It is useful for the handling of end cases in block copies and block fills based on streaming stores.

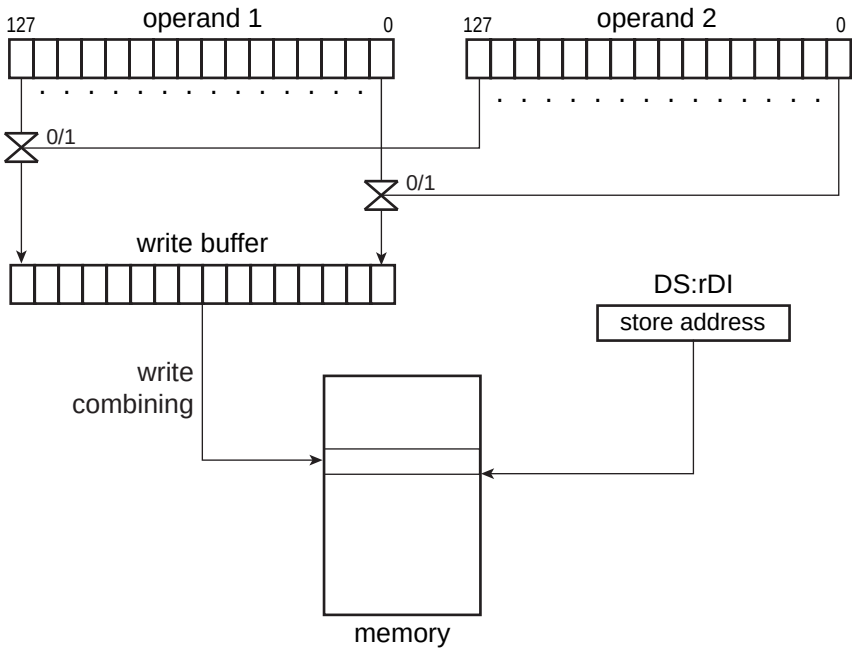


Figure 4-24. (V)MASKMOVDQU Move Mask Operation

4.6.1.3 Move Mask

- (V)PMOVMSKB—Packed Move Mask Byte

The (V)PMOVMSKB instruction moves the most-significant bit of each byte in an XMM register to the low-order word of a 32-bit or 64-bit general-purpose register, with zero-extension. The instruction is useful for extracting bits from mask patterns, or zero values from quantized data, or sign bits—resulting in a byte that can be used for data-dependent branching. Figure 4-25 below shows the (V)PMOVMSKB operation using the example of a 128-bit source operand held in an XMM register.

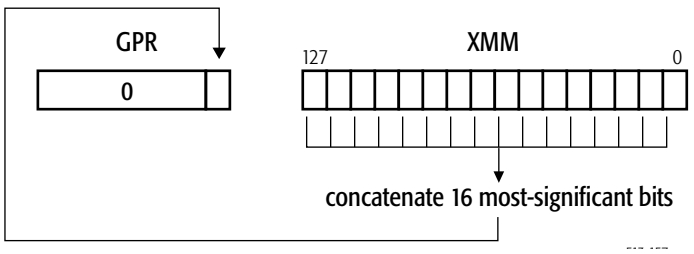


Figure 4-25. (V)PMOVMSKB Move Mask Operation

AVX2 adds support for a 256-bit source operand held in a YMM register. The 32-bit results is zero-extended to 64 bits.

4.6.1.4 Vector Conditional Moves

XOP instruction set includes the vector conditional move instructions:

- VPCMOV—Vector Conditional Moves
- VPPERM—Packed Permute Bytes

The VPCMOV instruction implements the C/C++ language ternary '?' operator a bit level. Each bit of the destination YMM/XMM register is copied from the corresponding bit of either the first or second source operand based on the value of the corresponding bit of a third source operand. This instruction has both 128-bit and 256-bit forms. The VPCMOV instruction allows either the second or the third operand to be source from memory, based on the XOP.W bit.

The VPPERM instruction performs vector permutation on a packed array of 32 bytes composed of two 16-byte input operands. The VPPERM instruction replaces each destination byte with 00h, FFh, or one of the 32 bytes of the packed array. A byte selected from the array may have an additional operation such as NOT or bit reversal applied to it, before it is written to the destination. The action for each destination byte is determined by a corresponding control byte. The VPPERM instruction allows either the second 16-byte input array or the control array to be memory based, per the XOP.W bit.

4.6.2 Data Conversion

The integer data-conversion instructions convert integer operands to floating-point operands. These instructions take integer source operands. For data-conversion instructions that take floating-point source operands, see "Data Conversion" on page 190. For data-conversion instructions that take 64-bit source operands, see Section 5.6.4 "Data Conversion" on page 257 and Section 5.7.2 "Data Conversion" on page 271.

4.6.2.1 Convert Integer to Floating-Point

These instructions convert integer data types in a YMM/XMM register or memory into floating-point data types in a YMM/XMM register.

- (V)CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point
- (V)CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point

The (V)CVTDQ2PS instruction converts four (eight, for 256-bit form) 32-bit signed integer values in the second operand to four (eight) single-precision floating-point values and writes the converted values to the specified XMM (YMM) register. If the result of the conversion is an inexact value, the value is rounded. The (V)CVTDQ2PD instruction is analogous to (V)CVTDQ2PS except that it converts two (four) 64-bit signed integer values to two (four) double-precision floating-point values.

4.6.2.2 Convert MMX Integer to Floating-Point

These instructions convert integer data types in MMX registers or memory into floating-point data types in XMM registers.

- CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point
- CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point

The CVTPI2PS instruction converts two 32-bit signed integer values in an MMX register or a 64-bit memory location to two single-precision floating-point values and writes the converted values in the low-order 64 bits of an XMM register. The high-order 64 bits of the XMM register are not modified.

The CVTPI2PD instruction is analogous to CVTPI2PS except that it converts two 32-bit signed integer values to two double-precision floating-point values and writes the converted values in the full 128 bits of an XMM register.

Before executing a CVTPI2*x* instruction, software should ensure that the MMX registers are properly initialized so as to prevent conflict with their aliased use by x87 floating-point instructions. This may require clearing the MMX state, as described in "Accessing Operands in MMX™ Registers" on page 232.

For a description of SSE instructions that convert in the opposite direction—floating-point to integer in MMX registers—see "Convert Floating-Point to MMX™ Integer" on page 192. For a summary of instructions that operate on MMX registers, see Chapter 5, "64-Bit Media Programming."

4.6.2.3 Convert GPR Integer to Floating-Point

These instructions convert integer data types in GPR registers or memory into floating-point data types in XMM registers.

- (V)CVTSI2SS—Convert Signed Doubleword or Quadword Integer to Scalar Single-Precision Floating-Point
- (V)CVTSI2SD—Convert Signed Doubleword or Quadword Integer to Scalar Double-Precision Floating-Point

The (V)CVTSI2SS instruction converts a 32-bit or 64-bit signed integer value in a general-purpose register or memory location to a single-precision floating-point value and writes the converted value to the low-order 32 bits of an XMM register. The legacy version of the instruction leaves the three high-order doublewords of the destination XMM register unmodified. The extended version of the instruction copies the three high-order doublewords of another XMM register (specified in the first source operand) to the destination.

The (V)CVTSI2SD instruction converts a 32-bit or 64-bit signed integer value in a general-purpose register or memory location to a double-precision floating-point value and writes the converted value to the low-order 64 bits of an XMM register. The legacy version of the instruction leaves the high-order 64 bits in the destination XMM register unmodified. The extended version of the instruction copies the upper 64 bits of another XMM register (specified in the first source operand) to the destination.

4.6.2.4 Convert Packed Integer Format

A common operation on packed integers is the conversion by zero or sign extension of packed integers into wider data types. These instructions convert from a smaller packed integer type to a larger integer type. Only the number of integers that will fit in the destination XMM or YMM register are converted starting with the least-significant integer in the source operand.

- (V)PMOVSXBW—Sign extend 8-bit integers in the source operand to 16 bits and pack into destination register.
- (V)PMOVZXBW—Zero extend 8-bit integers in the source operand to 16 bits and pack into destination register.
- (V)PMOVSXBD—Sign extend 8-bit integers in the source operand to 32 bits and pack into destination register.
- (V)PMOVZXBD—Zero extend 8-bit integers in the source operand to 32 bits and pack into destination register.
- (V)PMOVSXWD—Sign extend 16-bit integers in the source operand to 32 bits and pack into destination register.
- (V)PMOVZXWD—Zero extend 16-bit integers in the source operand to 32 bits and pack into destination register.
- (V)PMOVSXBQ—Sign extend 8-bit integers in the source operand to 64 bits and pack into destination register.
- (V)PMOVZXBQ—Zero extend 8-bit integers in the source operand to 64 bits and pack into destination register.
- (V)PMOVSXWQ—Sign extend 16-bit integers in the source operand to 64 bits and pack into destination register.
- (V)PMOVZXWQ—Zero extend 16-bit integers in the source operand to 64 bits and pack into destination register.
- (V)PMOVSXDQ—Sign extend 32-bit integers in the source operand to 64 bits and pack into destination register.
- (V)PMOVZXDQ—Zero extend 32-bit integers in the source operand to 64 bits and pack into destination register.

The source operand is an XMM/YMM register or a 128-bit or 256-bit memory location. The destination is an XMM/YMM register. When accessing memory, no alignment is required for any of the instructions unless alignment checking is enabled. In which case, all conversions must be aligned to the width of the memory reference. The legacy form of these instructions support 128-bit operands. AVX2 adds support for 256-bit operands to the extended forms.

4.6.3 Data Reordering

The integer data-reordering instructions pack, unpack, interleave, extract, insert, and shuffle the elements of vector operands.

4.6.3.1 Pack with Saturation

These instructions pack larger data types into smaller data types, thus halving the precision of each element in a vector operand.

- (V)PACKSSDW—Pack with Saturation Signed Doubleword to Word
- (V)PACKUSDW—Pack with Unsigned Saturation Doubleword to Word
- (V)PACKSSWB—Pack with Saturation Signed Word to Byte
- (V)PACKUSWB—Pack with Saturation Signed Word to Unsigned Byte

PACKSSDW and the 128-bit form of VPACKSSDW convert each of the four signed doubleword integers in two source operands (an XMM register, and another XMM register or 128-bit memory location) into signed word integers and packs the converted values into the destination register. The 256-bit form of VPACKSSDW performs this operation separately on the upper and lower 128 bits of its operands. The (V)PACKUSDW instruction does the same operation except that it converts signed doubleword integers into unsigned (rather than signed) word integers.

PACKSSWB and the 128-bit form of VPACKSSWB convert each of the eight signed word integers in two source operands (an XMM register, and another XMM register or 128-bit memory location) into signed 8-bit integers and packs the converted values into the destination register. The 256-bit form of VPACKSSDW performs this operation separately on the upper and lower 128 bits of its operands. The (V)PACKUSWB instruction does the same operation except that it converts signed word integers into unsigned (rather than signed) bytes.

Figure 4-26 shows an example of a (V)PACKSSDW instruction using the example of 128-bit vector operands. The operation merges vector elements of 2x size into vector elements of 1x size, thus reducing the precision of the vector-element data types. Any results that would otherwise overflow or underflow are saturated (clamped) at the maximum or minimum representable value, respectively, as described in “Saturation” on page 122.

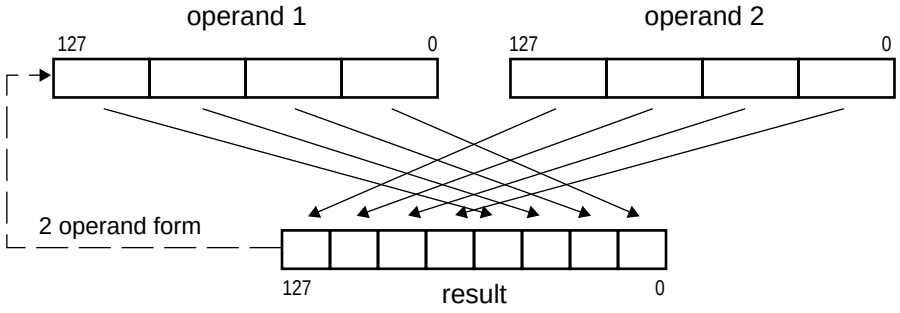


Figure 4-26. (V)PACKSSDW Pack Operation

Conversion from higher-to-lower precision is often needed, for example, by multiplication operations in which the higher-precision format is used for source operands in order to prevent possible overflow, and the lower-precision format is the desired format for the next operation.

4.6.3.2 Packed Blend

These instructions select vector elements from one of two source operands to be copied to the corresponding elements of the destination.

- (V)PBLENDVB—Variable Blend Packed Bytes
- (V)PBLENDW—Blend Packed Words

(V)PBLENDVB and (V)PBLENDW instructions copy bytes or words from either of two sources to the specified destination register based on selector bits in a mask. If the mask bit is a 0 the corresponding element is copied from the first source operand. If the mask bit is a 1, the element is copied from the second source operand.

For (V)PBLENDVB the mask is composed of the most significant bits of the elements of a third source operand. For the legacy instruction PBLENDVB, the mask is contained in the implicit operand register XMM0. For the extended form, the mask is contained in an XMM or YMM register specified via encoding in the instruction. For (V)PBLENDW the mask is specified via an immediate byte.

For the legacy form and the 128-bit extended form of these instructions, the first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. For the 256-bit extended form, the first source operand is an YMM register and the second source operand is either an YMM register or a 256-bit memory location.

For the legacy instructions, the destination is also the first source operand. For the extended forms, the destination is a separately specified YMM/XMM register.

AVX2 adds support for 256-bit operands to the AVX forms of these instructions.

4.6.3.3 Unpack and Interleave

These instructions interleave vector elements from either the high or low halves of two source operands.

- (V)PUNPCKHBW—Unpack and Interleave High Bytes
- (V)PUNPCKHWD—Unpack and Interleave High Words
- (V)PUNPCKHDQ—Unpack and Interleave High Doublewords
- (V)PUNPCKHQDQ—Unpack and Interleave High Quadwords
- (V)PUNPCKLBW—Unpack and Interleave Low Bytes
- (V)PUNPCKLWD—Unpack and Interleave Low Words
- (V)PUNPCKLDQ—Unpack and Interleave Low Doublewords
- (V)PUNPCKLQDQ—Unpack and Interleave Low Quadwords

The (V)PUNPCKHBW instruction copies the eight high-order bytes from its two source operands and interleaves them into the destination register. The bytes in the low-order half of the source operands are ignored. The (V)PUNPCKHWD, (V)PUNPCKHDQ, and (V)PUNPCKHQDQ instructions perform analogous operations for words, doublewords, and quadwords in the source operands,

packing them into interleaved words, interleaved doublewords, and interleaved quadwords in the destination.

The (V)PUNPCKLBW, (V)PUNPCKLWD, (V)PUNPCKLDQ, and (V)PUNPCKLQDQ instructions are analogous to their high-element counterparts except that they take elements from the low quadword of each source vector and ignore elements in the high quadword.

Depending on the hardware implementation, if the second source operand is located in memory, the 64 bits of the operand not required to perform the operation may or may not be read.

Figure 4-27 shows an example of the (V)PUNPCKLWD instruction using the example of 128-bit vector operands. The elements are taken from the low half of the source operands. Elements from the second source operand are placed to the left of elements from first source operand.

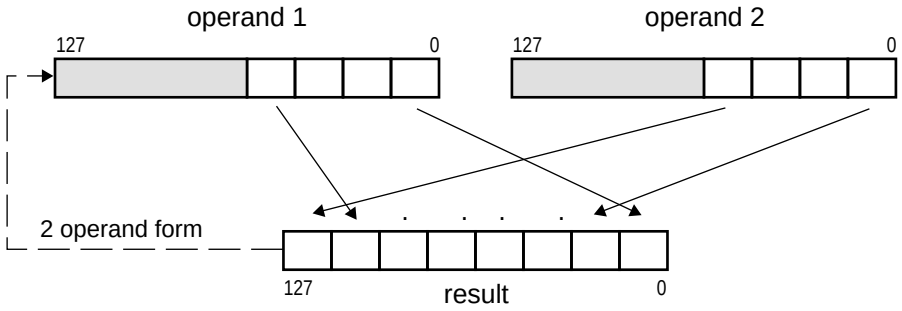


Figure 4-27. (V)PUNPCKLWD Unpack and Interleave Operation

If operand 2 is a vector consisting of all zero-valued elements, the unpack instructions perform the function of expanding vector elements of 1x size into vector elements of 2x size. Conversion from lower-to-higher precision is often needed, for example, prior to multiplication operations in which the higher-precision format is used for source operands in order to prevent possible overflow during multiplication.

If both source operands are of identical value, the unpack instructions can perform the function of duplicating adjacent elements in a vector.

The (V)PUNPCK*x* instructions can be used in a repeating sequence to transpose rows and columns of an array. For example, such a sequence could begin with (V)PUNPCK*x*WD and be followed by (V)PUNPCK*x*QD**.** These instructions can also be used to convert pixel representation from RGB format to color-plane format, or to interleave interpolation elements into a vector.

AVX2 adds support for 256-bit operands. When the operand size is 256 bits, the unpack and interleave operation is performed independently on the upper and lower halves of the source operands and the results written to the respective 128 bits of the destination YMM register.

4.6.3.4 Extract and Insert

These instructions copy a word element from a vector, in a manner specified by an immediate operand.

- EXTRQ—Extract Field from Register
- INSERTQ—Insert Field
- (V)PEXTRB—Extract Packed Byte
- (V)PEXTRW—Extract Packed Word
- (V)PEXTRD—Extract Packed Doubleword
- (V)PEXTRQ—Extract Packed Quadword
- (V)PINSRB—Packed Insert Byte
- (V)PINSRW—Packed Insert Word
- (V)PINSRD—Packed Insert Doubleword
- (V)PINSRQ—Packed Insert Quadword

The EXTRQ instruction extracts specified bits from the lower 64 bits of the destination XMM register. The extracted bits are saved in the least-significant bit positions of the destination and the remaining bits in the lower 64 bits of the destination register are cleared to 0. The upper 64 bits of the destination register are undefined.

The INSERTQ instruction inserts a specified number of bits from the lower 64 bits of the source operand into a specified bit position of the lower 64 bits of the destination operand. No other bits in the lower 64 bits of the destination are modified. The upper 64 bits of the destination are undefined.

The (V)PEXTRB, (V)PEXTRW, (V)PEXTRD, and (V)PEXTRQ instructions extract a single byte, word, doubleword, or quadword from an XMM register, as selected by the immediate-byte operand, and write it to memory or to the low-order bits of a general-purpose register with zero-extension to 32 bit or 64 bits as required. These instructions are useful for loading computed values, such as table-lookup indices, into general-purpose registers where the values can be used for addressing tables in memory.

The (V)PINSRB, (V)PINSRW, (V)PINSRD, and (V)PINSRQ instructions insert a byte, word, or doubleword value from the low-order bits of a general-purpose register or from a memory location into an XMM register. The location in the destination register is selected by the immediate-byte operand. For the legacy form, the other elements of the destination register are not modified. For the extended form, the other elements are filled in from a second source XMM register. As an example of these instructions, Figure 4-28 below provides a schematic the (V)PINSRD instruction.

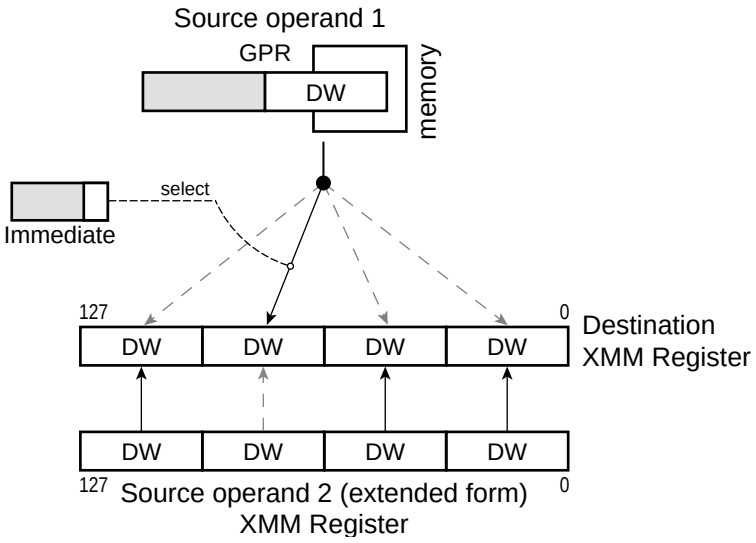


Figure 4-28. (V)PINSRD Operation

4.6.3.5 Shuffle

These instructions reorder the elements of a vector.

- (V)PSHUFB—Packed Shuffle Byte
- (V)PSHUFD—Packed Shuffle Doublewords
- (V)PSHUFHW—Packed Shuffle High Words
- (V)PSHUFLW—Packed Shuffle Low Words

The (V)PSHUFB instruction copies bytes from the first source operand to the destination or clears bytes in the destination as specified by control bytes in the second source operand. Each byte in the second operand controls how the corresponding byte is the destination is selected or cleared.

For PSHUFB and the 128-bit version of VPSHUFB, the first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. For the 256-bit version of the extended form, the first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. For the legacy form, the first source XMM register is also the destination. For the extended form, a separate destination register is specified in the instruction.

The (V)PSHUFD instruction fills each doubleword of the destination register by copying any one of the doublewords in the source operand . An immediate byte operand specifies for each double word of the destination which doubleword to copy. For the 256-bit version of the extended form, the immediate byte is reused to specify the shuffle operation for the upper four doublewords of the destination YMM register.

The ordering of the shuffle can occur in one of 256 possible ways for a 128-bit destination or for each half of a 256-bit destination.

Figure 4-29 below shows one of the 256 possible shuffle operations using the example of a 128-bit source and destination.

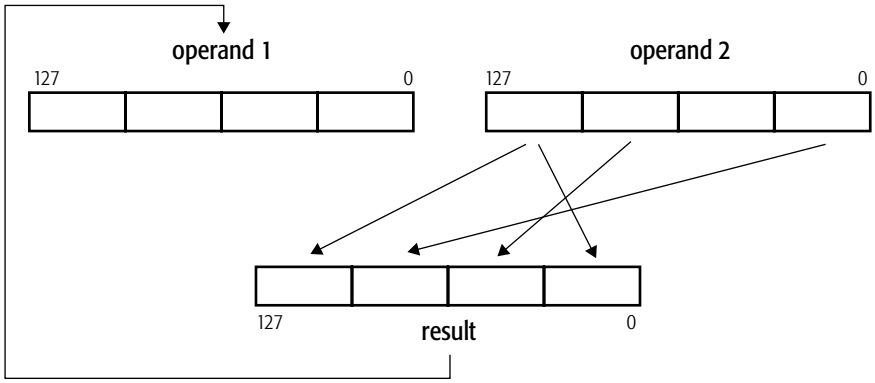


Figure 4-29. (V)PSHUFD Shuffle Operation

For the PSHUFD and the 128-bit version of VPSHUFD, the source operand is an XMM register or a 128-bit memory location and the destination is an XMM register. For the 256-bit version of VPSHUFD, the source operand is a YMM register or a 256-bit memory location and the destination is a YMM register.

The (V)PSHUFHW and (V)PSHUFLW instructions are analogous to (V)PSHUFD, except that they fill each word of the high or low quadword, respectively, of the destination register by copying any one of the four words in the high or low quadword of the source operand. The 256-bit version of the extended form of these instructions repeats the same operation on the high or low quadword, respectively, of the upper half of the destination YMM register using either the high or low quadword of the upper half of the source operand.

Figure 4-30 shows the (V)PSHUFHW operation using the example of a 128-bit source and destination.

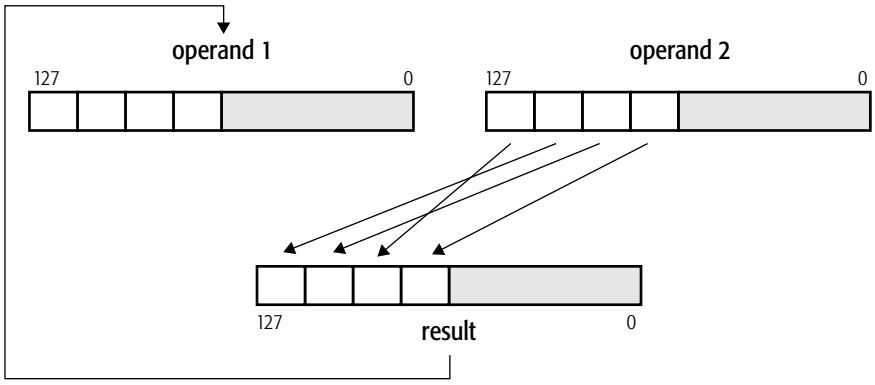


Figure 4-30. (V)PSHUFHW Shuffle Operation

(V)PSHUFHW and (V)PSHUFLW are useful, for example, in color imaging when computing alpha saturation of RGB values. In this case, (V)PSHUF*x*W can replicate an alpha value in a register so that parallel comparisons with three RGB values can be performed.

AVX2 adds support for 256-bit operands to the AVX forms of these instructions.

4.6.4 Arithmetic

Arithmetic operations can be unary or binary. A unary operation has a single operand and produces a single result. A binary operation has two operands that are combined arithmetically to produce a result. A vector arithmetic operation applies the same operation independently to all elements of a vector.

Figure 4-31 shows a typical unary vector operation. Figure 4-32 on page 165 shows a typical binary vector arithmetic operation.

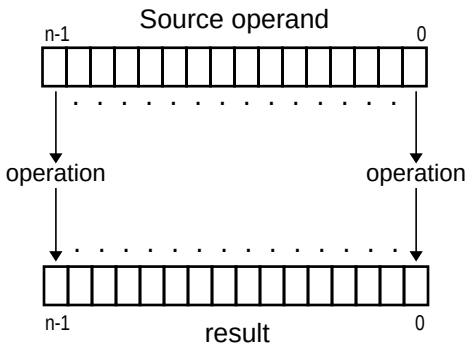


Figure 4-31. Unary Vector Arithmetic Operation

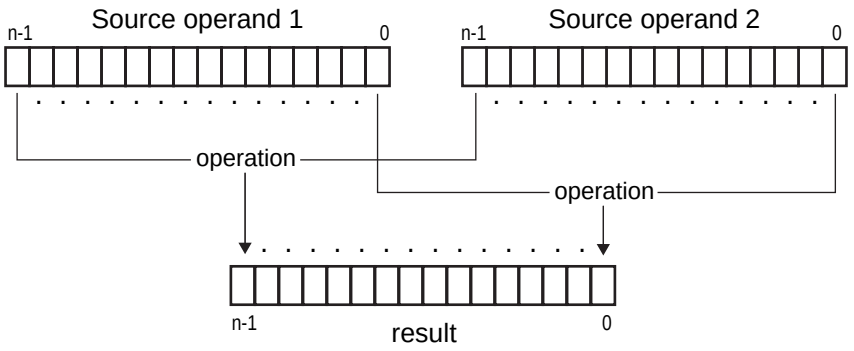


Figure 4-32. Binary Vector Arithmetic Operation

4.6.4.1 Absolute Value

- (V)PABSB—Packed Absolute Value Signed Byte
- (V)PABSW—Packed Absolute Value Signed Word
- (V)PABSD—Packed Absolute Value Signed Doubleword

These instructions operate on a vector of signed 8-bit, 16-bit, or 32-bit integers and produce a vector of unsigned integers of the same data width. Each element of the result is the absolute value of the corresponding element of the source operand. The AVX form of these instructions supports 128-bit operands and the AVX2 form supports 256-bit operands.

4.6.4.2 Addition

- (V)PADDB—Packed Add Bytes
- (V)PADDW—Packed Add Words
- (V)PADDD—Packed Add Doublewords
- (V)PADDQ—Packed Add Quadwords
- (V)PADDSB—Packed Add with Saturation Bytes
- (V)PADDSW—Packed Add with Saturation Words
- (V)PADDUSB—Packed Add Unsigned with Saturation Bytes
- (V)PADDUSW—Packed Add Unsigned with Saturation Words

The (V)PADDB, (V)PADDW, (V)PADDD, and (V)PADDQ instructions add each packed 8-bit ((V)PADDB), 16-bit ((V)PADDW), 32-bit ((V)PADDD), or 64-bit ((V)PADDQ) integer element in the second source operand to the corresponding same-sized integer element in the first source operand and write the integer result to the corresponding, same-sized element of the destination. Figure 4-32 diagrams a (V)PADDB operation (where the operation is addition). These instructions operate on both signed and unsigned integers. However, if the result overflows, the carry is ignored and only the low-order byte, word, doubleword, or quadword of each result is written to the destination. The

(V)PADDD instruction can be used together with (V)PMADDWD (page 169) to implement dot products.

The (V)PADDSB and (V)PADDSW instructions add each 8-bit ((V)PADDSB) or 16-bit ((V)PADDSW) signed integer element in the second source operand to the corresponding, same-sized signed integer element in the first source operand and write the signed integer result to the corresponding same-sized element of the destination. For each result in the destination, if the result is larger than the largest, or smaller than the smallest, representable 8-bit ((V)PADDSB) or 16-bit ((V)PADDSW) signed integer, the result is saturated to the largest or smallest representable value, respectively.

The (V)PADDUSB and (V)PADDUSW instructions perform saturating-add operations analogous to the (V)PADDSB and (V)PADDSW instructions, except on unsigned integer elements.

For the legacy form and 128-bit extended form of these instructions, the first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. For the 256-bit extended form, the first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. For the legacy form, the first source XMM register is also the destination. For the extended form, a separate destination register is specified in the instruction.

AVX2 adds support for 256-bit operands to the AVX forms of these instructions.

4.6.4.3 Subtraction

- (V)PSUBB—Packed Subtract Bytes
- (V)PSUBW—Packed Subtract Words
- (V)PSUBD—Packed Subtract Doublewords
- (V)PSUBQ—Packed Subtract Quadword
- (V)PSUBSB—Packed Subtract with Saturation Bytes
- (V)PSUBSW—Packed Subtract with Saturation Words
- (V)PSUBUSB—Packed Subtract Unsigned and Saturate Bytes
- (V)PSUBUSW—Packed Subtract Unsigned and Saturate Words

The subtraction instructions perform operations analogous to the addition instructions.

The (V)PSUBB, (V)PSUBW, (V)PSUBD, and (V)PSUBQ instructions subtract each 8-bit ((V)PSUBB), 16-bit ((V)PSUBW), 32-bit ((V)PSUBD), or 64-bit ((V)PSUBQ) integer element in the second operand from the corresponding, same-sized integer element in the first operand and write the integer result to the corresponding, same-sized element of the destination. For vectors of *n* number of elements, the operation is:

```
result[i] = operand1[i] - operand2[i]
```

where: i = 0 to n – 1

These instructions operate on both signed and unsigned integers. However, if the result underflows, the borrow is ignored and only the low-order byte, word, doubleword, or quadword of each result is written to the destination.

The (V)PSUBSB and (V)PSUBSW instructions subtract each 8-bit ((V)PSUBSB) or 16-bit ((V)PSUBSW) signed integer element in the second operand from the corresponding, same-sized signed integer element in the first operand and write the signed integer result to the corresponding, same-sized element of the destination. For each result in the destination, if the result is larger than the largest, or smaller than the smallest, representable 8-bit ((V)PSUBSB) or 16-bit ((V)PSUBSW) signed integer, the result is saturated to the largest or smallest representable value, respectively.

The (V)PSUBUSB and (V)PSUBUSW instructions perform saturating-add operations analogous to the (V)PSUBSB and (V)PSUBSW instructions, except on unsigned integer elements.

For the legacy form and 128-bit extended form of these instructions, the first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. For the 256-bit extended form, the first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. For the legacy form, the first source XMM register is also the destination. For the extended form, a separate destination register is specified in the instruction.

AVX2 adds support for 256-bit operands to the AVX forms of these instructions.

4.6.4.4 Multiplication

- (V)PMULHW—Packed Multiply High Signed Word
- (V)PMULHRSW—Packed Multiply High with Round and Scale Words
- (V)PMULLW—Packed Multiply Low Signed Word
- (V)PMULHUW—Packed Multiply High Unsigned Word
- (V)PMULUDQ—Packed Multiply Unsigned Doubleword to Quadword
- (V)PMULLD—Packed Multiply Low Signed Doubleword
- (V)PMULDQ—Packed Multiply Double Quadword

The (V)PMULHW instruction multiplies each 16-bit signed integer value in the first operand by the corresponding 16-bit integer in the second operand, producing a 32-bit intermediate result. The instruction then writes the high-order 16 bits of the 32-bit intermediate result of each multiplication to the corresponding word of the destination. The (V)PMULHRSW instruction performs the same multiplication as (V)PMULHW but rounds and scales the 32-bit intermediate result prior to truncating it to 16 bits. The (V)PMULLW instruction performs the same multiplication as (V)PMULHW but writes the low-order 16 bits of the 32-bit intermediate result to the corresponding word of the destination.

Figure 4-33 below shows the (V)PMULHW, (V)PMULLW, and (V)PMULHW instruction operations. The difference between the instructions is the manner in which the intermediate element result is reduced to 16 bits prior writing it to the destination.

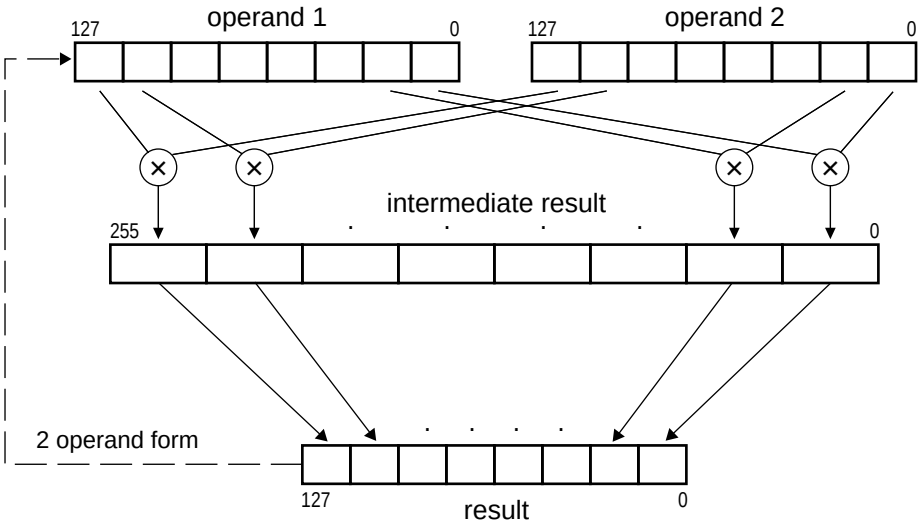


Figure 4-33. (V)PMULHW, (V)PMULLW, and (V)PMULHRSW Instructions

The (V)PMULHUW instruction performs the same multiplication as (V)PMULHW but on unsigned operands. Without this instruction, it is difficult to perform unsigned integer multiplies using SSE instructions. The instruction is useful in 3D rasterization, which operates on unsigned pixel values.

The (V)PMULUDQ instruction preserves the full precision of results by multiplying only half of the source-vector elements. It multiplies together the least significant doubleword (treating each as an unsigned 32-bit integer) of each quadword in the two source operands, writes the full 64-bit result of the low-order multiply to the low-order quadword of the destination, and writes the high-order product to the high-order quadword of the destination. Figure 4-34 below shows a (V)PMULUDQ operation using the example of 128-bit operands.

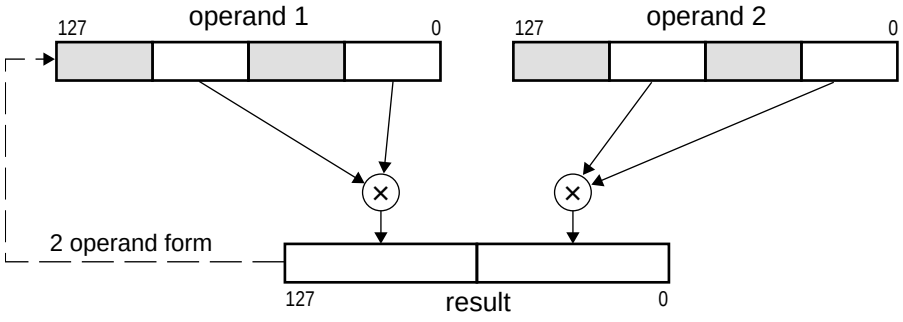


Figure 4-34. (V)PMULUDQ Multiply Operation

The 256-bit form of VPMULUDQ instruction performs the same operation on each half of the source operands to produce a 256-bit packed quadword result.

The (V)PMULLD instruction writes the lower 32 bits of the 64-bit product of a signed 32-bit integer multiplication of the corresponding doublewords of the source operands to each element of the destination.

PMULDQ and the 128-bit form of VPMULDQ writes the 64-bit signed product of the least-significant doubleword of the two source operands to the low quadword of the result and the 64-bit signed product of the low doubleword of the upper quadword of two source operands (bits [95:64]) to the upper quadword of the result. The 256-bit form of VPMULDQ performs similar operations on the upper 128 bits of the source operands to produce the upper 128 bits of the result.

For the legacy form and 128-bit extended form of these instructions, the first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. For the 256-bit extended form, the first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. For the legacy form, the first source XMM register is also the destination. For the extended form, a separate destination register is specified in the instruction.

AVX2 adds support for 256-bit operands to the AVX forms of these instructions.

See “Shift and Rotate” on page 175 for shift instructions that can be used to perform multiplication and division by powers of 2.

4.6.4.5 Multiply-Add

This instruction multiplies the elements of two source vectors and adds their intermediate results in a single operation.

- (V)PMADDWD—Packed Multiply Words and Add Doublewords

The (V)PMADDWD instruction multiplies each 16-bit signed value in the first source operand by the corresponding 16-bit signed value in the second source operand. The instruction then adds the adjacent 32-bit intermediate results of each multiplication, and writes the 32-bit result of each addition into the corresponding doubleword of the destination. For vectors of *n* number of source elements (src), *m* number of destination elements (dst), and n = 2m, the operation is:

```
dst[j] = ((src1[i] * src2[i]) + (src1[i+1] * src2[i+1]))
```

where: j = 0 to m – 1
i = 2j

(V)PMADDWD thus performs four or eight signed multiply-adds in parallel. Figure 4-35 below diagrams the operation using the example of 128-bit operands.

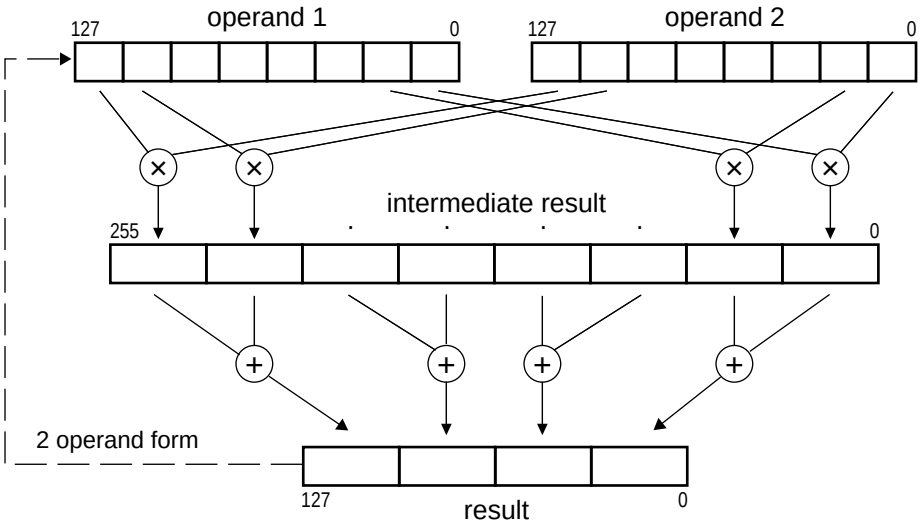


Figure 4-35. (V)PMADDWD Multiply-Add Operation

(V)PMADDWD can be used with one source operand (for example, a coefficient) taken from memory and the other source operand (for example, the data to be multiplied by that coefficient) taken from an XMM/YMM register. The instruction can also be used together with the (V)PADDD instruction (page 165) to compute dot products. Scaling can be done, before or after the multiply, using a vector-shift instruction (page 175).

For PMADDWD, the first source XMM register is also the destination. VPMADDWD specifies a separate destination XMM/YMM register encoded in the instruction. AVX2 adds support for 256-bit operands to the AVX forms of these instructions.

4.6.5 Enhanced Media

4.6.5.1 Multiply-Add and Accumulate

The multiply and accumulate and multiply, add and accumulate instructions operate on and produce packed signed integer values. These instructions allow the accumulation of results from (possibly) many iterations of similar operations without a separate intermediate addition operation to update the accumulator register.

The operation of a typical XOP integer multiply and accumulate instruction is shown in Figure 4-36 on page 171. The multiply and accumulate instructions operate on and produce packed signed integer values. These instructions first multiply the value in the first source operand by the corresponding value in the second source operand. Each signed integer product is then added to the corresponding value in the third source operand, which is the accumulator and is identical to the destination operand. The results may or may not be saturated prior to being written to the destination register, depending on the instruction.

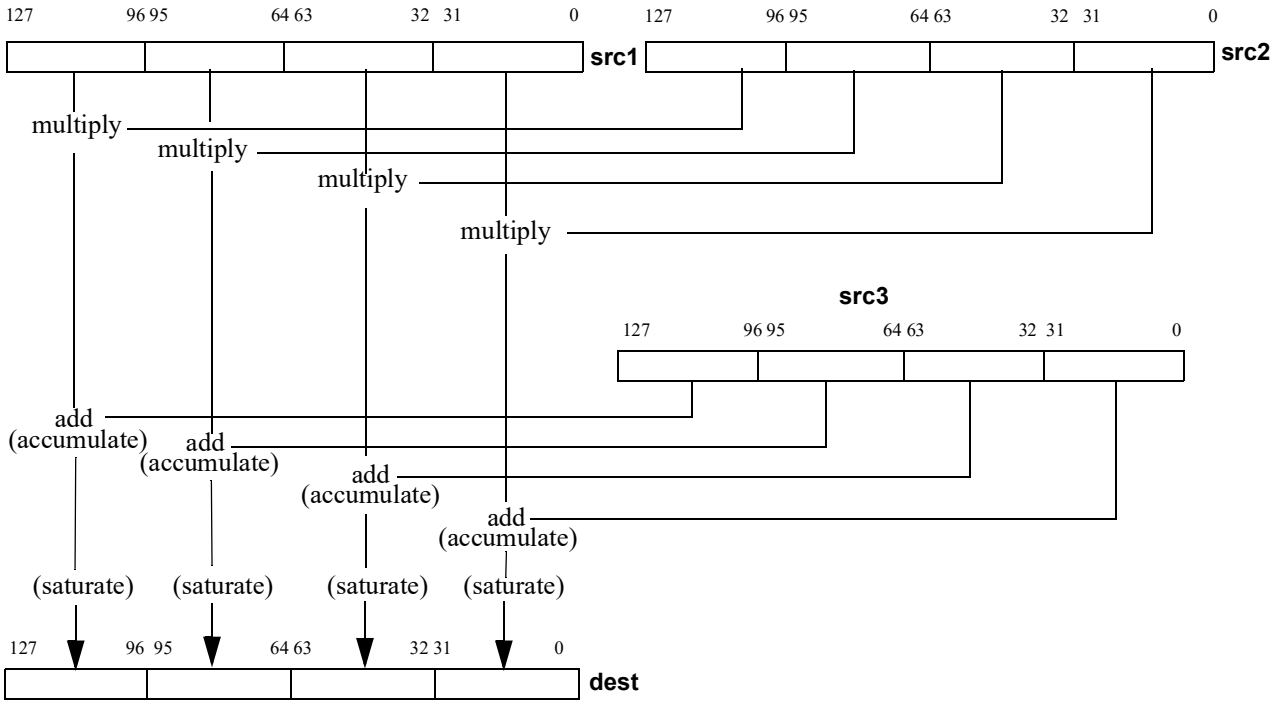


Figure 4-36. Operation of Multiply and Accumulate Instructions

The XOP instruction extensions provide the following integer multiply and accumulate instructions.

- VPMACSSWW—Packed Multiply Accumulate Signed Word to Signed Word with Saturation
- VPMACSWW—Packed Multiply Accumulate Signed Word to Signed Word
- VPMACSSWD—Packed Multiply Accumulate Signed Word to Signed Doubleword with Saturation
- VPMACSWD—Packed Multiply Accumulate Signed Word to Signed Doubleword
- VPMACSSDD—Packed Multiply Accumulate Signed Doubleword to Signed Doubleword with Saturation
- VPMACSDD—Packed Multiply Accumulate Signed Doubleword to Signed Doubleword
- VPMACSSDQL—Packed Multiply Accumulate Signed Low Doubleword to Signed Quadword with Saturation
- VPMACSSDQH—Packed Multiply Accumulate Signed High Doubleword to Signed Quadword with Saturation
- VPMACSDQL—Packed Multiply Accumulate Signed Low Doubleword to Signed Quadword
- VPMACSDQH—Packed Multiply Accumulate Signed High Doubleword to Signed Quadword

The operation of the multiply, add and accumulate instructions is illustrated in Figure 4-37.

The multiply, add and accumulate instructions first multiply each packed signed integer value in the first source operand by the corresponding packed signed integer value in the second source operand. The odd and even adjacent resulting products are then added. Each resulting sum is then added to the corresponding packed signed integer value in the third source operand.

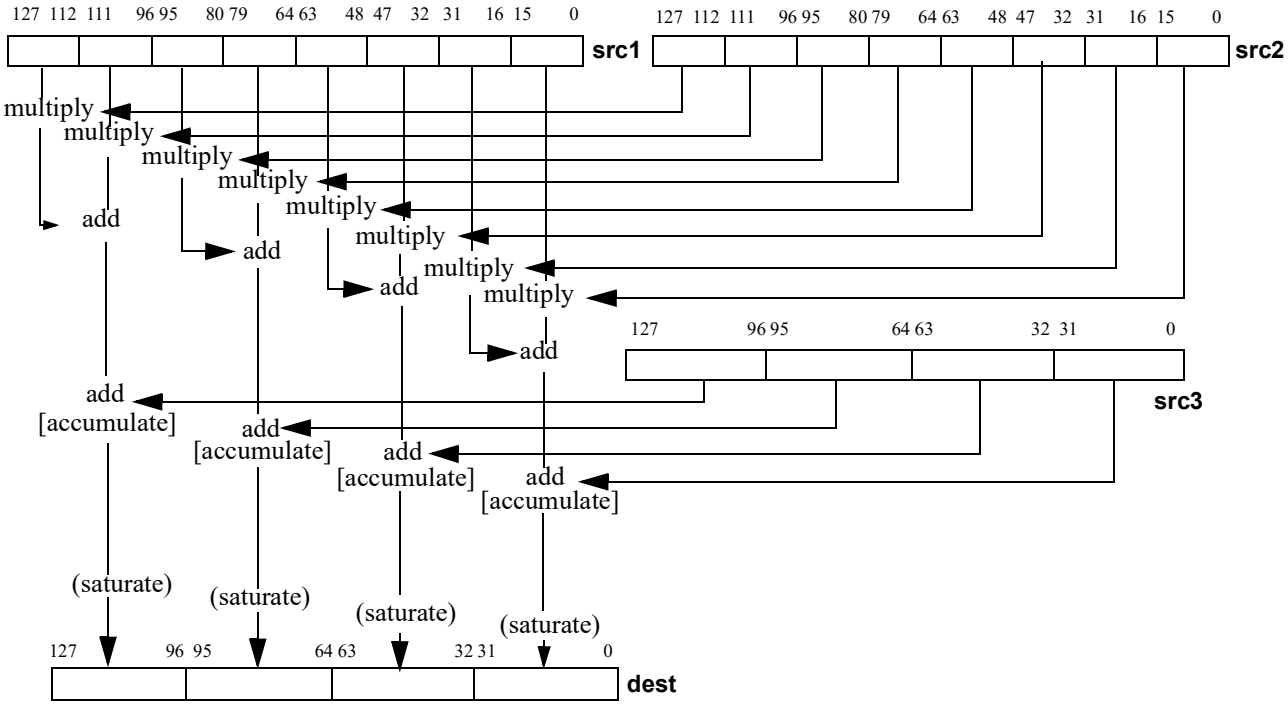


Figure 4-37. Operation of Multiply, Add and Accumulate Instructions

The XOP instruction set provides the following integer multiply, add and accumulate instructions.

- VPMADCSSWD—Packed Multiply Add and Accumulate Signed Word to Signed Doubleword with Saturation
- VPMADCSWD—Packed Multiply Add and Accumulate Signed Word to Signed Doubleword

4.6.5.2 Packed Integer Horizontal Add and Subtract

The packed horizontal add and subtract signed byte instructions successively add adjacent pairs of signed integer values from the source XMM register or 128-bit memory operand and pack the (sign extended) integer result of each addition in the destination.

- VPHADDBW—Packed Horizontal Add Signed Byte to Signed Word
- VPHADDBD—Packed Horizontal Add Signed Byte to Signed Doubleword
- VPHADDBQ—Packed Horizontal Add Signed Byte to Signed Quadword
- VPHADDDQ—Packed Horizontal Add Signed Doubleword to Signed Quadword

- VPHADDUBW—Packed Horizontal Add Unsigned Byte to Word
- VPHADDUBD—Packed Horizontal Add Unsigned Byte to Doubleword
- VPHADDUBQ—Packed Horizontal Add Unsigned Byte to Quadword
- VPHADDUWD—Packed Horizontal Add Unsigned Word to Doubleword
- VPHADDUWQ—Packed Horizontal Add Unsigned Word to Quadword
- VPHADDUDQ—Packed Horizontal Add Unsigned Doubleword to Quadword
- VPHADDWD—Packed Horizontal Add Signed Word to Signed Doubleword
- VPHADDWQ—Packed Horizontal Add Signed Word to Signed Quadword
- VPHSUBBW—Packed Horizontal Subtract Signed Byte to Signed Word
- VPHSUBWD—Packed Horizontal Subtract Signed Word to Signed Doubleword
- VPHSUBDQ—Packed Horizontal Subtract Signed Doubleword to Signed Quadword

4.6.5.3 Average

- (V)PAVGB—Packed Average Unsigned Bytes
- (V)PAVGW—Packed Average Unsigned Words

The (V)PAVG*x* instructions compute the rounded average of each unsigned 8-bit ((V)PAVGB) or 16-bit ((V)PAVGW) integer value in the first operand and the corresponding, same-sized unsigned integer in the second operand and write the result in the corresponding, same-sized element of the destination. The rounded average is computed by adding each pair of operands, adding 1 to the temporary sum, and then right-shifting the temporary sum by one bit-position. For vectors of *n* number of elements, the operation is:

```
operand1[i] = ((operand1[i] + operand2[i]) + 1) ÷ 2
```

where: i = 0 to n – 1

The (V)PAVGB instruction is useful for MPEG decoding, in which motion compensation performs many byte-averaging operations between and within macroblocks. In addition to speeding up these operations, (V)PAVGB can free up registers and make it possible to unroll the averaging loops.

The legacy form of these instructions support 128-bit operands. AVX2 adds support for 256-bit operands to the AVX forms of these instructions.

4.6.5.4 Sum of Absolute Differences

- (V)PSADBW—Packed Sum of Absolute Differences of Bytes into a Word

The (V)PSADBW instruction computes the absolute values of the differences of corresponding 8-bit signed integer values in the two quadword halves of both source operands, sums the differences for each quadword half, and writes the two unsigned 16-bit integer results in the destination. The sum for the high-order half is written in the least-significant word of the destination's high-order quadword, with the remaining bytes cleared to all 0s. The sum for the low-order half is written in the least-significant word of the destination's low-order quadword, with the remaining bytes cleared to all 0s.

Figure 4-38 shows the (V)PSADBW operation. Sums of absolute differences are useful, for example, in computing the L1 norm in motion-estimation algorithms for video compression.

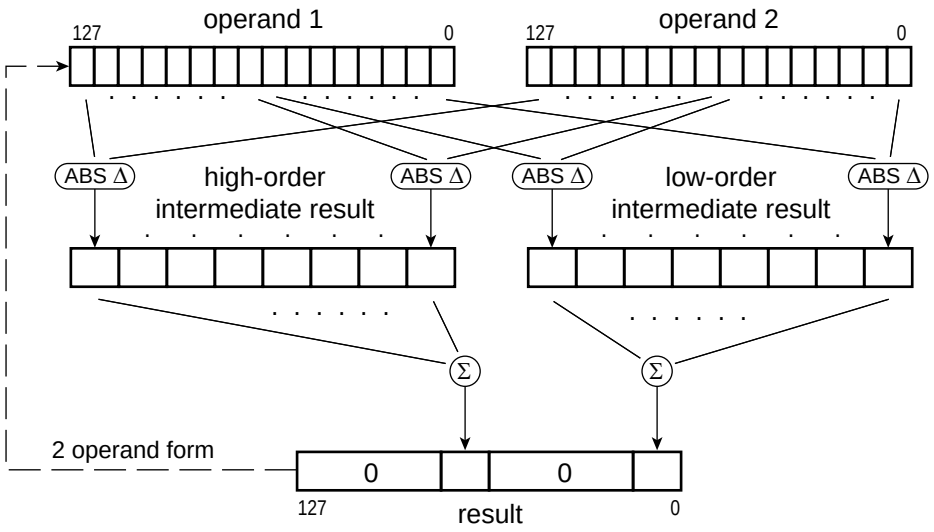


Figure 4-38. (V)PSADBW Sum-of-Absolute-Differences Operation

For PSADBW, the first source XMM register is also the destination. VPSADBW specifies a separate destination YMM/XMM register encoded in the instruction.

AVX2 extends the function of VPSADBW to operate on 256-bit operands and produce 4 sums of absolute differences.

4.6.5.5 Improved Sums of Absolute Differences for 4-Byte Blocks

- (V)MPSADBW—Performs eight 4-byte wide Sum of Absolute Differences (SAD) operations to produce eight word integers.

The (V)MPSADBW instruction performs eight 4-byte wide SAD operations per instruction to produce eight results. Compared to (V)PSADBW, (V)MPSADBW operates on smaller chunks (4-byte instead of 8-byte chunks). This makes the instruction better suited to video coding standards such as VC.1 and H.264.

(V)MPSADBW performs four times the number of absolute difference operations than that of (V)PSADBW (per instruction). This can improve performance for dense motion searches.

(V)MPSADBW uses a 4-byte wide field from a source operand. The offset of the 4-byte field within the 128-bit source operand is specified by two immediate control bits. (V)MPSADBW produces eight 16-bit SAD results. Each 16-bit SAD result is formed from overlapping pairs of 4 bytes in the destination with the 4-byte field from the source operand. (V)MPSADBW uses eleven consecutive bytes in the destination operand. Its offset is specified by a control bit in the immediate byte (i.e. the offset can be from byte 0 or from byte 4).

For MPSADBW, the first source XMM register is also the destination. VMPSADBW specifies a separate destination YMM/XMM register encoded in the instruction.

AVX2 extends the function of VMPSADBW to operate on 256-bit operands and produce 16 sums of absolute differences.

4.6.6 Shift and Rotate

The vector-shift instructions are useful for scaling vector elements to higher or lower precision, packing and unpacking vector elements, and multiplying and dividing vector elements by powers of 2.

4.6.6.1 Left Logical Shift

- (V)PSLLW—Packed Shift Left Logical Words
- (V)PSLLD—Packed Shift Left Logical Doublewords
- (V)PSLLQ—Packed Shift Left Logical Quadwords
- (V)PSLLDQ—Packed Shift Left Logical Double Quadword

The (V)PSLLW, (V)PSLLD, and (V)PSLLQ instructions left-shift each of the 16-bit, 32-bit, or 64-bit values, respectively, in the first source operand by the number of bits specified in the second source operand. The instructions then write each shifted value into the corresponding, same-sized element of the destination. The low-order bits that are emptied by the shift operation are cleared to 0. The shift count (second source operand) is specified by the contents of a register, a value loaded from memory, or an immediate byte.

In integer arithmetic, left logical shifts effectively multiply unsigned operands by positive powers of 2. Thus, for vectors of *n* number of elements, the operation is:

$$operand1[i] = operand1[i] * 2^{operand2}$$

where: i = 0 to n – 1

The (V)PSLLDQ instruction differs from the other three left-shift instructions because it operates on bytes rather than bits. It left-shifts the value in a YMM/XMM register by the number of bytes specified in an immediate byte value.

In the legacy form of these instructions, the first source XMM register is also the destination. The extended form specifies a separate destination YMM/XMM register encoded in the instruction. AVX2 adds support for 256-bit operands to the extended form of these instructions.

4.6.6.2 Right Logical Shift

- (V)PSRLW—Packed Shift Right Logical Words
- (V)PSRLD—Packed Shift Right Logical Doublewords
- (V)PSRLQ—Packed Shift Right Logical Quadwords
- (V)PSRLDQ—Packed Shift Right Logical Double Quadword

The (V)PSRLW, (V)PSRLD, and (V)PSRLQ instructions right-shift each of the 16-bit, 32-bit, or 64-bit values, respectively, in the first source operand by the number of bits specified in the second source operand. The instructions then write each shifted value into the corresponding, same-sized element of the destination. The high-order bits that are emptied by the shift operation are cleared to 0. The shift count (second source operand) is specified by the contents of a register, a value loaded from memory, or an immediate byte.

In integer arithmetic, right logical bit-shifts effectively divide unsigned or positive-signed operands by positive powers of 2. Thus, for vectors of *n* number of elements, the operation is:

$$operand1[i] = operand1[i] \div 2^{operand2}$$

where: i = 0 to n – 1

The (V)PSRLDQ instruction differs from the other three right-shift instructions because it operates on bytes rather than bits. It right-shifts the value in a YMM/XMM register by the number of bytes specified in an immediate byte value. (V)PSRLDQ can be used, for example, to move the high 8 bytes of an XMM register to the low 8 bytes of the register. In some implementations, however, (V)PUNPCKHQDQ may be a better choice for this operation.

In the legacy form of these instructions, the first source XMM register is also the destination. The extended form specifies a separate destination YMM/XMM register encoded in the instruction. AVX2 adds support for 256-bit operands to the extended form of these instructions.

4.6.6.3 Right Arithmetic Shift

- (V)PSRAW—Packed Shift Right Arithmetic Words
- (V)PSRAD—Packed Shift Right Arithmetic Doublewords

The (V)PSRA*x* instructions right-shift each of the 16-bit ((V)PSRAW) or 32-bit ((V)PSRAD) values in the first operand by the number of bits specified in the second operand. The instructions then write each shifted value into the corresponding, same-sized element of the destination. The high-order bits that are emptied by the shift operation are filled with the sign bit of the initial value.

In integer arithmetic, right arithmetic shifts effectively divide signed operands by positive powers of 2. Thus, for vectors of *n* number of elements, the operation is:

$$operand1[i] = operand1[i] \div 2^{operand2}$$

where: i = 0 to n – 1

In the legacy form of these instructions, the first source XMM register is also the destination. The extended form specifies a separate destination YMM/XMM register encoded in the instruction. AVX2 adds support for 256-bit operands to the extended form of these instructions.

4.6.6.4 Packed Integer Shifts

The packed integer shift instructions shift each element of the vector in the first source XMM or 128-bit memory operand by the amount specified by a control byte contained in the least significant byte of the corresponding element of the second source operand. The result of each shift operation is returned

in the destination XMM register. This allows load-and-shift from memory operations, with either the source operand or the shift-count operand being memory-based, as indicated by the XOP.W bit. The XOP instruction set provides the following packed integer shift instructions:

- VPSHLB—Packed Shift Logical Bytes
- VPSHLW—Packed Shift Logical Words
- VPSHLD—Packed Shift Logical Doublewords
- VPSHLQ—Packed Shift Logical Quadwords
- VPSHAB—Packed Shift Arithmetic Bytes
- VPSHAW—Packed Shift Arithmetic Words
- VPSHAD—Packed Shift Arithmetic Doublewords
- VPSHAQ—Packed Shift Arithmetic Quadwords

There is no legacy form for these instructions.

4.6.6.5 Packed Integer Rotate

There are two variants of the packed integer rotate instructions. The first is identical to that described above (see “Packed Integer Shifts”). In the second variant, the control byte is supplied as an 8-bit immediate operand that specifies a single rotate amount for every element in the first source operand.

The XOP instruction set provides the following packed integer rotate instructions:

- VPROTB—Packed Rotate Bytes
- VPROTW—Packed Rotate Words
- VPROTD—Packed Rotate Doublewords
- VPROTQ—Packed Rotate Quadwords

There is no legacy form for these instructions.

4.6.7 Compare

The integer vector-compare instructions compare two operands and either write a mask or the maximum or minimum value.

4.6.7.1 Compare and Write Mask

- (V)PCMPEQB—Packed Compare Equal Bytes
- (V)PCMPEQW—Packed Compare Equal Words
- (V)PCMPEQD—Packed Compare Equal Doublewords
- (V)PCMPEQQ—Packed Compare Equal Quadwords
- (V)PCMPGTB—Packed Compare Greater Than Signed Bytes
- (V)PCMPGTW—Packed Compare Greater Than Signed Words
- (V)PCMPGTD—Packed Compare Greater Than Signed Doublewords

- (V)PCMPGTQ—Packed Compare Greater Than Signed Quadwords

The (V)PCMPEQ*x* and (V)PCMPGT*x* instructions compare corresponding bytes, words, doublewords, or quadwords in the two source operands. The instructions then write a mask of all 1s or 0s for each compare into the corresponding, same-sized element of the destination. Figure 4-39 shows a (V)PCMPEQ*x* compare operation.

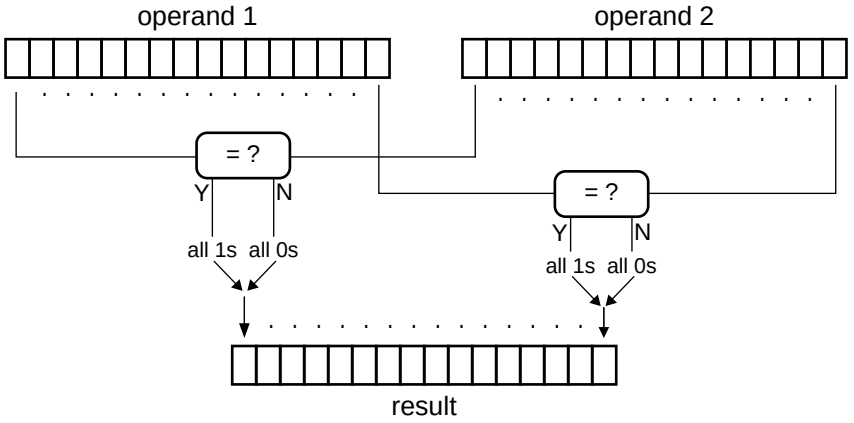


Figure 4-39. (V)PCMPEQ*x* Compare Operation

For the (V)PCMPEQ*x* instructions, if the compared values are equal, the result mask is all 1s. If the values are not equal, the result mask is all 0s. For the (V)PCMPGT*x* instructions, if the signed value in the first operand is greater than the signed value in the second operand, the result mask is all 1s. If the value in the first operand is less than or equal to the value in the second operand, the result mask is all 0s.

For the legacy form and the 128-bit extended form of these instructions, the first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. For the 256-bit extended form, the first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. In the legacy form of these instructions, the first source XMM register is also the destination. The extended form specifies a separate destination YMM/XMM register encoded in the instruction.

By specifying the same register for both operands, (V)PCMPEQ*x* can be used to set the bits in a register to all 1s.

Figure 4-21 on page 148 shows an example of a non-branching sequence that implements a two-way multiplexer—one that is equivalent to the following sequence of ternary operators in C or C++:

```
r0 = a0 > b0 ? a0 : b0
r1 = a1 > b1 ? a1 : b1
r2 = a2 > b2 ? a2 : b2
r3 = a3 > b3 ? a3 : b3
r4 = a4 > b4 ? a4 : b4
r5 = a5 > b5 ? a5 : b5
```

```
r6 = a6 > b6 ? a6 : b6
r7 = a7 > b7 ? a7 : b7
```

Assuming xmm0 contains the vector a, and xmm1 contains the vector b, the above C sequence can be implemented with the following assembler sequence:

```
MOVQ        xmm3, xmm0
PCMPGTW     xmm3, xmm1  ; a > b ? 0xffff : 0
PAND        xmm0, xmm3  ; a > b ? a: 0
PANDN       xmm3, xmm1  ; a > b ? 0 : b
POR         xmm0, xmm3  ; r = a > b ? a: b
```

In the above sequence, (V)PCMPGTW, (V)PAND, (V)PANDN, and (V)POR operate, in parallel, on all four elements of the vectors.

4.6.7.2 Compare and Write Minimum or Maximum

- (V)PMAXUB—Packed Maximum Unsigned Bytes
- (V)PMAXUW—Packed Maximum Unsigned Words
- (V)PMAXUD— Packed Maximum Unsigned Doublewords
- (V)PMAXSB—Packed Maximum Signed Bytes
- (V)PMAXSW—Packed Maximum Signed Words
- (V)PMAXSD—Packed Maximum Signed Doublewords
- (V)PMINUB—Packed Minimum Unsigned Bytes
- (V)PMINUW—Packed Minimum Unsigned Words
- (V)PMINUD—Packed Minimum Unsigned Doublewords
- (V)PMINSB—Packed Minimum Signed Bytes
- (V)PMINSW—Packed Minimum Signed Words
- (V)PMINSD—Packed Minimum Signed Doublewords

The (V)PMAXUB, (V)PMAXUW, and (V)PMAXUD and the (V)PMINUB, (V)PMINUW, (V)PMINUD instructions compare each of the 8-bit, 16-bit, or 32-bit unsigned integer values in the first operand with the corresponding 8-bit, 16-bit, or 32-bit unsigned integer values in the second operand. The instructions then write the maximum ((V)PMAXU*x*) or minimum ((V)PMINU*x*) of the two values for each comparison into the corresponding element of the destination.

The (V)PMAXSB, (V)PMAXSW, (V)PMAXSD and the (V)PMINSB, (V)PMINSW, (V)PMINSD instructions perform operations analogous to the (V)PMAXU*x* and (V)PMINU*x* instructions, except on 16-bit signed integer values.

For the legacy form and the 128-bit extended form of these instructions, the first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. For the 256-bit extended form, the first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. In the legacy form of these

instructions, the first source XMM register is also the destination. The extended form specifies a separate destination YMM/XMM register encoded in the instruction.

4.6.7.3 Packed Integer Comparison and Predicate Generation

- VPCOMUB—Compare Vector Unsigned Bytes
- VPCOMUW—Compare Vector Unsigned Words
- VPCOMUD—Compare Vector Unsigned Doublewords
- VPCOMUQ—Compare Vector Unsigned Quadwords
- VPCOMB—Compare Vector Signed Bytes
- VPCOMW—Compare Vector Signed Words
- VPCOMD—Compare Vector Signed Doublewords
- VPCOMQ—Compare Vector Signed Quadwords

These XOP comparison instructions compare packed integer values in the first source XMM register with corresponding packed integer values in the second source XMM register or 128-bit memory. The type of comparison is specified by the immediate-byte operand. The resulting predicate is placed in the destination XMM register. If the condition is true, all bits in the corresponding field in the destination register are set to 1s; otherwise all bits in the field are set to 0s.

Table 4-10. Immediate Operand Values for Unsigned Vector Comparison Operations

Immediate Operand Byte		Comparison Operation
Bits 7:3	Bits 2:0	
00000b	000b	Less Than
	001b	Less Than or Equal
	010b	Greater Than
	011b	Greater Than or Equal
	100b	Equal
	101b	Not Equal
	110b	False
	111b	True

The integer comparison and predicate generation instructions compare corresponding packed signed or unsigned bytes in the first and second source operands and write the result of each comparison in the corresponding element of the destination. The result of each comparison is a value of all 1s (TRUE) or all 0s (FALSE). The type of comparison is specified by the three low-order bits of the immediate-byte operand.

4.6.7.4 String and Text Processing Instructions

- (V)PCMPESTRI — Packed compare explicit-length strings, return index in ECX/RCX

- (V)PCMPESTRM — Packed compare explicit-length strings, return mask in XMM0
- (V)PCMPISTRI — Packed compare implicit-length strings, return index in ECX/RCX
- (V)PCMPISTRM — Packed compare implicit-length strings, return mask in XMM0

These four instructions use XMM registers to process string or text elements of up to 128-bits (16 bytes or 8 words). Each instruction uses an immediate byte to support an extensive set of programmable controls. These instructions return the result of processing each pair of string elements using either an index or a mask. Each instruction has an extended SSE (AVX) counterpart with the same functionality.

The capabilities of these instructions include:

- Handling string/text fragments consisting of bytes or words, either signed or unsigned
- Support for partial string or fragments less than 16 bytes in length, using either explicit length or implicit null-termination
- Four types of string compare operations on word/byte elements
- Up to 256 compare operations performed in a single instruction on all string/text element pairs
- Built-in aggregation of intermediate results from comparisons
- Programmable control of processing on intermediate results
- Programmable control of output formats in terms of an index or mask
- Bidirectional support for the index format
- Support for two mask formats: bit or natural element width
- Does not require 16-byte alignment for memory operand

All four instructions require the use of an immediate byte to control operation. The first source operand is an XMM register The second source operand can be either an XMM register or a 128-bit memory location. The immediate byte provides programmable control with the following attributes:

- Input data format
- Compare operation mode
- Intermediate result processing
- Output selection

Depending on the output format associated with the instruction, the text/string processing instructions implicitly uses either a general-purpose register (ECX/RCX) or an XMM register (XMM0) to return the final result. Neither of the source operands are modified.

Two of the four text-string processing instructions specify string length explicitly. They use two general-purpose registers (EDX, EAX) to specify the number of valid data elements (either word or byte) in the source operands. The other two instructions specify valid string elements using null termination. A data element is considered valid only if it has a lower index than the least significant null data element.

These instructions do not perform alignment checking on memory operands.

4.6.8 Logical

The vector-logic instructions perform Boolean logic operations, including AND, OR, and exclusive OR.

4.6.8.1 AND

- (V)PAND—Packed Logical Bitwise AND
- (V)PANDN—Packed Logical Bitwise AND NOT
- (V)PTEST—Packed Bit Test

The (V)PAND instruction performs a logical bitwise AND of the values in the first and second operands and writes the result to the destination.

The (V)PANDN instruction inverts the first operand (creating a ones-complement of the operand), ANDs it with the second operand, and writes the result to the destination. Table 4-11 shows an example.

Table 4-11. Example PANDN Bit Values

Operand1 Bit	Operand1 Bit (Inverted)	Operand2 Bit	PANDN Result Bit
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0

For the legacy form and the 128-bit extended form of (V)PAND and (V)PANDN, the first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. For the 256-bit extended form, the first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. In the legacy form of these instructions, the first source XMM register is also the destination. The extended form specifies a separate destination YMM/XMM register encoded in the instruction.

AVX2 adds support for 256-bit operands to the VPAND and VPANDN instructions.

The packed bit test instruction (V)PTEST is similar to the general-purpose instruction TEST. Using the first source operand as a bit mask, (V)PTEST may be used to test whether the bits in the second source operand that correspond to the set bits in the mask are all zeros. If this is true rFLAGS[ZF] is set. If all the bits in the second source operand that correspond to the cleared bits in the mask are all zeros, then the rFLAGS[CF] bit is set.

Because neither source operand is modified, (V)PTEST simplifies branching operations, such as branching on signs of packed floating-point numbers, or branching on zero fields.

The AVX instruction VPTEST has both a 128-bit and a 256-bit form.

4.6.8.2 OR and Exclusive OR

- (V)POR—Packed Logical Bitwise OR
- (V)PXOR—Packed Logical Bitwise Exclusive OR

The (V)POR instruction performs a logical bitwise OR of the values in the first and second operands and writes the result to the destination.

The (V)PXOR instruction is analogous to (V)POR except it performs a bit-wise exclusive OR of the two source operands. (V)PXOR can be used to clear all bits in an XMM register by specifying the same register for both operands.

For the legacy form and the 128-bit extended form of these instructions, the first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. For the 256-bit extended form, the first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. In the legacy form of these instructions, the first source XMM register is also the destination. The extended form specifies a separate destination YMM/XMM register encoded in the instruction.

AVX2 adds support for 256-bit operands to the extended forms of these instructions.

4.6.9 Save and Restore State

These instructions save and restore the entire processor state for legacy SSE instructions.

4.6.9.1 Save and Restore 128-Bit, 64-Bit, and x87 State

- FXSAVE—Save XMM, MMX, and x87 State
- FXRSTOR—Restore XMM, MMX, and x87 State

The FXSAVE and FXRSTOR instructions save and restore the entire 512-byte processor state for legacy SSE instructions, 64-bit media instructions, and x87 floating-point instructions. The architecture supports two memory formats for FXSAVE and FXRSTOR, a 512-byte 32-bit legacy format and a 512-byte 64-bit format. Selection of the 32-bit or 64-bit format is determined by the effective operand size for the FXSAVE and FXRSTOR instructions. For details, see “FXSAVE and FXRSTOR Instructions” in Volume 2.

4.6.9.2 Save and Restore Extended Processor Context

- XSAVE—Save Extended Processor Context.
- XRSTOR—Restore Extended Processor Context.

The XSAVE and XRSTOR instructions provide a flexible means of saving and restoring not only the x87, 64-bit media, and legacy SSE state, but also the extended SSE context. The first 512 bytes of the save area supports the FXSAVE/FXRSTOR 512-byte 64-bit format. Subsequent bytes support an extensible data structure to be used for extended processor context such as the extended SSE context including the contents of the YMM registers. XSAVEOPT, XSAVEC, XSAVES and XRSTORS are

optional optimized variants of XSAVE and XRSTOR. For details, see the descriptions of these instructions in Volume 4.

4.6.9.3 Save and Restore Control and Status

- (V)STMXCSR—Store MXCSR Control/Status Register
- (V)LDMXCSR—Load MXCSR Control/Status Register

The (V)STMXCSR and (V)LDMXCSR instructions save and restore the 32-bit contents of the MXCSR register. For further information, see Section 4.2.2 “MXCSR Register” on page 115.

4.7 Instruction Summary—Floating-Point Instructions

This section summarizes the SSE instructions that operate on scalar and packed floating-point values. Software running at any privilege level can use any of the instructions discussed below, given that hardware and system software support is provided and the appropriate instruction subset is enabled. Detection and enablement of instruction subsets is normally handled by operating system software. Hardware support for each instruction subset is indicated by processor feature bits. These are accessed via the CPUID instruction. See Volume 3 for details on the CPUID instruction and the feature bits associated with the SSE instruction set.

The SSE instructions discussed below include those that use the YMM/XMM registers as well as instructions that convert data from floating-point to integer formats. For more detail on each instruction, see individual instruction reference pages in the Instruction Reference chapter of Volume 4, “128-Bit and 256-Bit Media Instructions.”

For a summary of the integer SSE instructions including instructions that convert from integer to floating-point formats, see Section 4.6 “Instruction Summary—Integer Instructions” on page 149.

For a summary of the 64-bit media floating-point instructions, see “Instruction Summary—Floating-Point Instructions” on page 270. For a summary of the x87 floating-point instructions, see “Instruction Summary” on page 310.

The following subsections are organized by functional groups. These are:

- Data Transfer
- Data Conversion
- Data Reordering
- Arithmetic
- Fused Multiply-Add Instructions
- Compare
- Logical

Most of the instructions described below have both legacy and AVX versions. For the 128-bit media instructions, the extended version is functionally equivalent to the legacy version except legacy

instructions leave the upper octword of the YMM register that overlays the destination XMM register unchanged, while the 128-bit AVX instruction always clears the upper octword. The descriptions that follow apply equally to the legacy instruction and its extended AVX version. Generally, for those extended instructions that have 256-bit variants, the number of elements operated upon in parallel doubles. Other differences will be noted at the end of the discussion.

4.7.1 Data Transfer

The data-transfer instructions copy 32-bit, 64-bit, 128-bit, or 256-bit data from memory to a YMM/XMM register, from a YMM/XMM register to memory, or from one register to another. The MOV mnemonic, which stands for *move*, is a misnomer. A copy function is actually performed instead of a move. A new copy of the source value is created at the destination address, and the original copy remains unchanged at its source location.

4.7.1.1 Move

- (V)MOVAPS—Move Aligned Packed Single-Precision Floating-Point
- (V)MOVAPD—Move Aligned Packed Double-Precision Floating-Point
- (V)MOVUPS—Move Unaligned Packed Single-Precision Floating-Point
- (V)MOVUPD—Move Unaligned Packed Double-Precision Floating-Point
- (V)MOVHPS—Move High Packed Single-Precision Floating-Point
- (V)MOVHPD—Move High Packed Double-Precision Floating-Point
- (V)MOVLPS—Move Low Packed Single-Precision Floating-Point
- (V)MOVLPD—Move Low Packed Double-Precision Floating-Point
- (V)MOVHLPS—Move Packed Single-Precision Floating-Point High to Low
- (V)MOVLHPS—Move Packed Single-Precision Floating-Point Low to High
- (V)MOVSS—Move Scalar Single-Precision Floating-Point
- (V)MOVSD—Move Scalar Double-Precision Floating-Point
- (V)MOVDDUP—Move Double-Precision and Duplicate
- (V)MOVSLDUP—Move Single-Precision High and Duplicate
- (V)MOVSHDUP—Move Single-Precision Low and Duplicate

Figure 4-40 on page 187 summarizes the capabilities of the floating-point move instructions except (V)MOVDDUP, (V)MOVSLDUP, (V)MOVSHDUP which are described in the following section.

The (V)MOVAPx instructions copy a vector of four (eight, for 256-bit form) single-precision floating-point values ((V)MOVAPS) or a vector of two (four) double-precision floating-point values ((V)MOVAPD) from the second operand to the first operand—i.e., from an YMM/XMM register or 128-bit (256-bit) memory location or to another YMM/XMM register, or vice versa. A general-protection exception occurs if a memory operand is not aligned on a 16-byte (32-byte) boundary, unless alternate alignment checking behavior is enabled by setting MSCSR[MM]. See Section 4.3.2 “Data Alignment” on page 120.

The (V)MOVUP*x* instructions perform operations analogous to the (V)MOVAP*x* instructions, except that unaligned memory operands do not cause a general-protection exception or invoke the alignment checking mechanism.

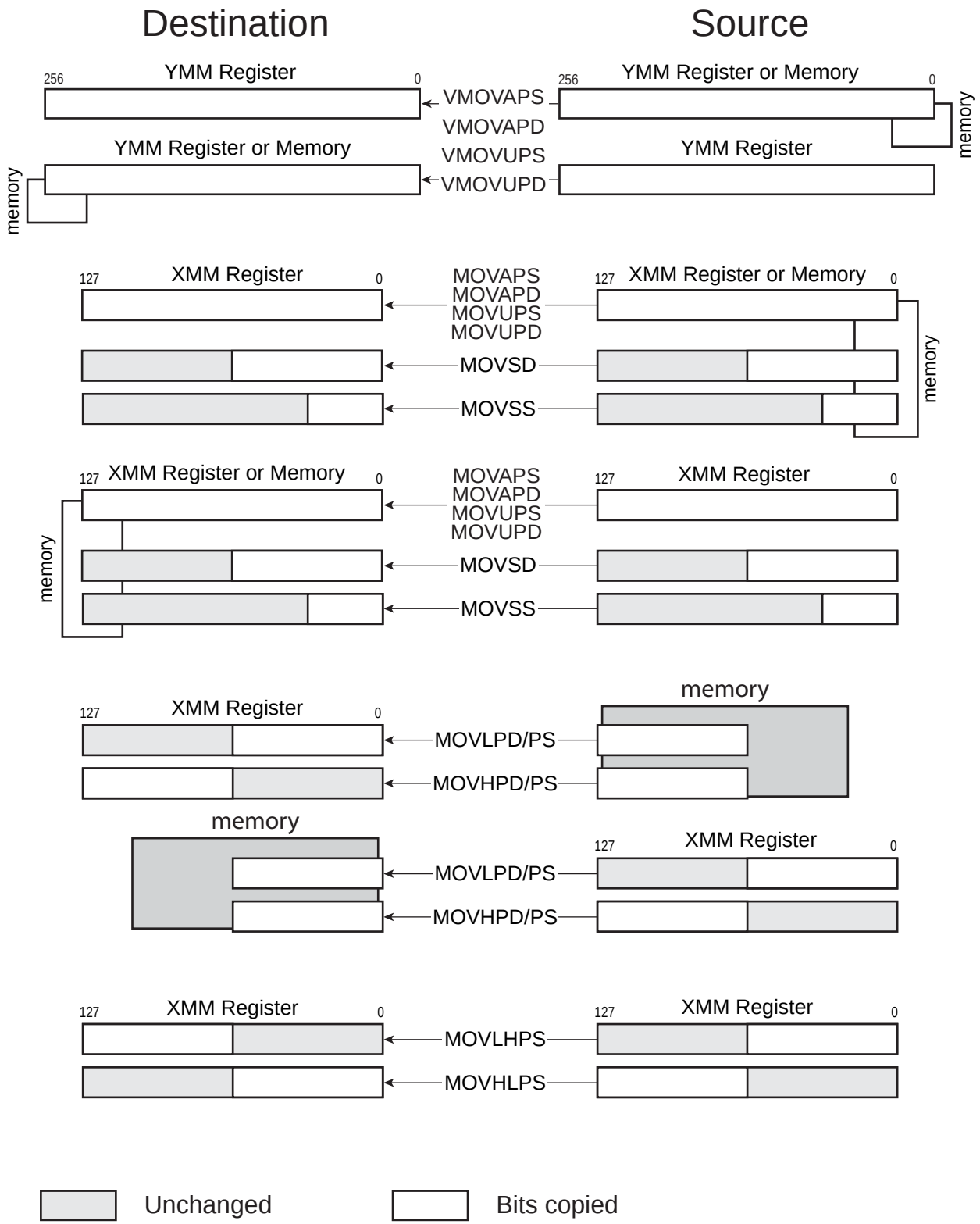


Figure 4-40. Floating-Point Move Operations

The (V)MOVHPS and (V)MOVHPD instructions copy a vector of two single-precision floating-point values ((V)MOVHPS) or one double-precision floating-point value ((V)MOVHPD) from a 64-bit memory location to the high-order 64 bits of an XMM register, or from the high-order 64 bits of an XMM register to a 64-bit memory location. In the memory-to-register case, the low-order 64 bits of the destination XMM register are not modified.

The (V)MOVLPS and (V)MOVLPD instructions copy a vector of two single-precision floating-point values ((V)MOVLPS) or one double-precision floating-point value ((V)MOVLPD) from a 64-bit memory location to the low-order 64 bits of an XMM register, or from the low-order 64 bits of an XMM register to a 64-bit memory location. In the memory-to-register case, the high-order 64 bits of the destination XMM register are not modified.

The (V)MOVHLPS instruction copies a vector of two single-precision floating-point values from the high-order 64 bits of an XMM register to the low-order 64 bits of another XMM register. The high-order 64 bits of the destination XMM register are not modified. The (V)MOVLHPS instruction performs an analogous operation except in the opposite direct (low-order to high-order), and the low-order 64 bits of the destination XMM register are not modified.

The (V)MOVSS instruction copies a scalar single-precision floating-point value from the low-order 32 bits of an XMM register or a 32-bit memory location to the low-order 32 bits of another XMM register, or vice versa. If the source operand is an XMM register, the high-order 96 bits of the destination XMM register are either cleared or left unmodified based on the instruction encoding. If the source operand is a 32-bit memory location, the high-order 96 bits of the destination XMM register are cleared to all 0s.

The (V)MOVSD instruction copies a scalar double-precision floating-point value from the low-order 64 bits of an XMM register or a 64-bit memory location to the low-order 64 bits of another XMM register, or vice versa. If the source operand is an XMM register, the high-order 64 bits of the destination XMM register are not modified. If the source operand is a memory location, the high-order 64 bits of the destination XMM register are cleared to all 0s.

The above MOVSD instruction should not be confused with the same-mnemonic MOVSD (move string doubleword) instruction in the general-purpose instruction set. Assemblers distinguish the two instructions by their operand data types.

The basic function of each corresponding extended (V) form instruction is the same as the legacy form. The instructions VMOVSS, VMOVSD, VMOVHPS, VMOVHPD, VMOVLPS, VMOVHLPS, VMOVLHPS provide additional function, supporting the merging in of bits from a second register source operand.

4.7.1.2 Move with Duplication

These instructions move two copies of the affected data segments from the source XMM register or 128-bit memory operand to the target destination register.

The (V)MOVDDUP moves one copy of the lower quadword of the source operand into each quadword half of the destination operand. The 256-bit version of VMOVDDUP copies and duplicates the two even-indexed quadwords.

The (V)MOVSLDUP instruction moves two copies of the first (least significant) doubleword of the source operand into the first two doubleword segments of the destination operand and moves two copies of the third doubleword of the source operand into the third and fourth doubleword segments of the destination operand. The 256-bit version of VMOVSLDUP writes two copies the even-indexed doubleword elements of the source YMM register to ascending quadwords of the destination YMM register.

The (V)MOVSHDUP instruction moves two copies of the second doubleword of the source operand into the first two doubleword segments of the destination operand and moves two copies of the fourth doubleword of the source operand into the upper two doubleword segments of the destination operand. The 256-bit version of VMOVSHDUP writes two copies the odd-indexed doubleword elements of the source YMM register to ascending quadwords of the destination YMM register.

4.7.1.3 Move Non-Temporal

The move non-temporal instructions are *streaming-store* instructions. They minimize pollution of the cache.

- (V)MOVNTPD—Move Non-temporal Packed Double-Precision Floating-Point
- (V)MOVNTPS—Move Non-temporal Packed Single-Precision Floating-Point
- MOVNTSD—Move Non-temporal Scalar Double-Precision Floating-Point
- MOVNTSS—Move Non-temporal Scalar Single-Precision Floating-Point

These instructions indicate to the processor that their data is *non-temporal*, which assumes that the data they reference will be used only once and is therefore not subject to cache-related overhead (as opposed to *temporal* data, which assumes that the data will be accessed again soon and should be cached). The non-temporal instructions use weakly-ordered, write-combining buffering of write data, and they minimize cache pollution. The exact method by which cache pollution is minimized depends on the hardware implementation of the instruction. For further information, see “Memory Optimization” on page 98.

The (V)MOVNTP*x* instructions copy four packed single-precision floating-point ((V)MOVNTPS) or two packed double-precision floating-point ((V)MOVNTPD) values from an XMM register into a 128-bit memory location. The 256-bit form of the VMOVNTP*x* instructions store the contents of the specified source YMM register to memory.

The MOVNTS*x* instructions store one double precision floating point XMM register value into a 64 bit memory location or one single precision floating point XMM register value into a 32-bit memory location.

4.7.1.4 Move Mask

- (V)MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask

- (V)MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask

The (V)MOVMSKPS instruction copies the sign bits of four (eight, for the 256-bit form) single-precision floating-point values in an XMM (YMM) register to the four (eight) low-order bits of a 32-bit or 64-bit general-purpose register, with zero-extension. The (V)MOVMSKPD instruction copies the sign bits of two (four) double-precision floating-point values in an XMM (YMM) register to the two (four) low-order bits of a general-purpose register, with zero-extension. The result of either instruction is a sign-bit mask that can be used for data-dependent branching. Figure 4-41 shows the MOVMSKPS operation.

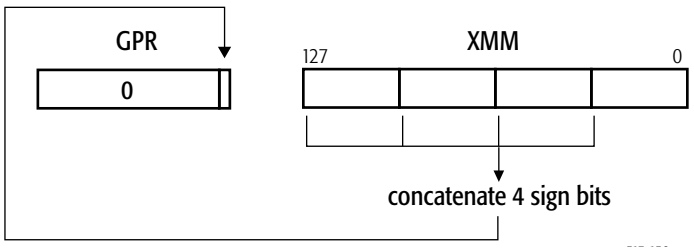


Figure 4-41. (V)MOVMSKPS Move Mask Operation

4.7.2 Data Conversion

The floating-point data-conversion instructions convert floating-point operands to integer operands.

These data-conversion instructions take 128-bit floating-point source operands. For data-conversion instructions that take 128-bit integer source operands, see “Data Conversion” on page 155. For data-conversion instructions that take 64-bit source operands, see “Data Conversion” on page 257 and “Data Conversion” on page 271.

4.7.2.1 Convert Floating-Point to Floating-Point

These instructions convert floating-point data types in XMM registers or memory into different floating-point data types in XMM registers.

- (V)CVTPS2PD—Convert Packed Single-Precision Floating-Point to Packed Double-Precision Floating-Point
- (V)CVTPD2PS—Convert Packed Double-Precision Floating-Point to Packed Single-Precision Floating-Point
- (V)CVTSS2SD—Convert Scalar Single-Precision Floating-Point to Scalar Double-Precision Floating-Point
- (V)CVTSD2SS—Convert Scalar Double-Precision Floating-Point to Scalar Single-Precision Floating-Point

The (V)CVTPS2PD instruction converts two (four, for the 256-bit form) single-precision floating-point values in the low-order 64 bits (entire 128 bits) of the source operand–either a XMM register or a

64-bit (128-bit) memory location–to two (four) double-precision floating-point values in the destination operand–an XMM (YMM) register.

The (V)CVTPD2PS instruction converts two (four, for the 256-bit form) double-precision floating-point values in the source operand –either an XMM (YMM) register or a 64-bit (128-bit) memory location–to two (four) single-precision floating-point values. The 128-bit form zero-extends the 64-bit packed result to 128 bits before writing it to the destination XMM register. The 256-bit form writes the 128-bit packed result to the destination XMM register. If the result of the conversion is an inexact value, the value is rounded.

The (V)CVTSS2SD instruction converts a single-precision floating-point value in the low-order 32 bits of the source operand to a double-precision floating-point value in the low-order 64 bits of the destination. For the legacy form of the instruction, the high-order 64 bits in the destination XMM register are not modified. In the extended form, the high-order 64 bits are copied from another source XMM register.

The (V)CVTSD2SS instruction converts a double-precision floating-point value in the low-order 64 bits of the source operand to a single-precision floating-point value in the low-order 32 bits of the destination. For the legacy form of the instruction, the three high-order doublewords in the destination XMM register are not modified. In the extended form, the three high-order doublewords are copied from another source XMM register. If the result of the conversion is an inexact value, the value is rounded.

4.7.2.2 Convert Floating-Point to XMM Integer

These instructions convert floating-point data types in YMM/XMM registers or memory into integer data types in YMM/XMM registers.

- (V)CVTPS2DQ—Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers
- (V)CVTPD2DQ—Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers
- (V)CVTTPS2DQ—Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers, Truncated
- (V)CVTTPD2DQ—Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers, Truncated

The (V)CVTPS2DQ and (V)CVTTPS2DQ instructions convert four (eight, in the 256-bit version) single-precision floating-point values in the source operand to four (eight) 32-bit signed integer values in the destination. For the 128-bit form, the source operand is either an XMM register or a 128-bit memory location and the destination is an XMM register. For the 256-bit form, the source operand is either a YMM register or a 256-bit memory location and the destination is a YMM register. For the (V)CVTPS2DQ instruction, if the result of the conversion is an inexact value, the value is rounded, but for the (V)CVTTPS2DQ instruction such a result is truncated (rounded toward zero).

The (V)CVTPD2DQ and (V)CVTTPD2DQ instructions convert two (four, in 256-bit version) double-precision floating-point values in the source operand to two (four) 32-bit signed integer values in the destination. For the 128-bit form, the source operand is either an XMM register or a 128-bit memory location and the destination is the low-order 64 bits of an XMM register. The high-order 64 bits in the destination XMM register are cleared to all 0s. For the 256-bit form, the source operand is either a YMM register or a 256-bit memory location and the destination is a XMM register. For the (V)CVTPD2DQ instruction, if the result of the conversion is an inexact value, the value is rounded, but for the (V)CVTTPD2DQ instruction such a result is truncated (rounded toward zero).

For a description of SSE instructions that convert in the opposite direction—integer to floating-point—see “Convert Integer to Floating-Point” on page 155.

4.7.2.3 Convert Floating-Point to MMX™ Integer

These instructions convert floating-point data types in XMM registers or memory into integer data types in MMX registers.

- CVTPS2PI—Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers
- CVTPD2PI—Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers
- CVTTPS2PI—Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers, Truncated
- CVTTPD2PI—Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers, Truncated

The CVTPS2PI and CVTTPS2PI instructions convert two single-precision floating-point values in the low-order 64 bits of an XMM register or a 64-bit memory location to two 32-bit signed integer values in an MMX register. For the CVTPS2PI instruction, if the result of the conversion is an inexact value, the value is rounded, but for the CVTTPS2PI instruction such a result is truncated (rounded toward zero).

The CVTPD2PI and CVTTPD2PI instructions convert two double-precision floating-point values in an XMM register or a 128-bit memory location to two 32-bit signed integer values in an MMX register. For the CVTPD2PI instruction, if the result of the conversion is an inexact value, the value is rounded, but for the CVTTPD2PI instruction such a result is truncated (rounded toward zero).

Before executing a CVTxPS2PI or CVTxPD2PI instruction, software should ensure that the MMX registers are properly initialized so as to prevent conflict with their aliased use by x87 floating-point instructions. This may require clearing the MMX state, as described in “Accessing Operands in MMX™ Registers” on page 232.

For a description of SSE instructions that convert in the opposite direction—integer in MMX registers to floating-point in XMM registers—see “Convert MMX Integer to Floating-Point” on page 155. For a summary of instructions that operate on MMX registers, see Chapter 5, “64-Bit Media Programming.”

4.7.2.4 Convert Floating-Point to GPR Integer

These instructions convert floating-point data types in XMM registers or memory into integer data types in GPR registers.

- (V)CVTSS2SI—Convert Scalar Single-Precision Floating-Point to Signed Doubleword or Quadword Integer
- (V)CVTSD2SI—Convert Scalar Double-Precision Floating-Point to Signed Doubleword or Quadword Integer
- (V)CVTTSS2SI—Convert Scalar Single-Precision Floating-Point to Signed Doubleword or Quadword Integer, Truncated
- (V)CVTTSD2SI—Convert Scalar Double-Precision Floating-Point to Signed Doubleword or Quadword Integer, Truncated

The (V)CVTSS2SI and (V)CVTTSS2SI instructions convert a single-precision floating-point value in the low-order 32 bits of an XMM register or a 32-bit memory location to a 32-bit or 64-bit signed integer value in a general-purpose register. For the (V)CVTSS2SI instruction, if the result of the conversion is an inexact value, the value is rounded, but for the (V)CVTTSS2SI instruction such a result is truncated (rounded toward zero).

The (V)CVTSD2SI and (V)CVTTSD2SI instructions convert a double-precision floating-point value in the low-order 64 bits of an XMM register or a 64-bit memory location to a 32-bit or 64-bit signed integer value in a general-purpose register. For the (V)CVTSD2SI instruction, if the result of the conversion is an inexact value, the value is rounded, but for the (V)CVTTSD2SI instruction such a result is truncated (rounded toward zero).

For a description of SSE instructions that convert in the opposite direction—integer in GPR registers to floating-point in XMM registers—see “Convert GPR Integer to Floating-Point” on page 156. For a summary of instructions that operate on GPR registers, see Chapter 3, “General-Purpose Programming.”

4.7.2.5 Half-Precision Floating-Point Conversion

The F16C instruction subset supports the 16-bit floating-point data type with two instructions (VCVTPH2PS and VCVTPS2PH) to convert 16-bit floating-point values to and from single-precision format. The half-precision floating point data type is discussed in “Half-Precision Floating-Point Data Type” on page 130.

- VCVTPH2PS—Convert Half-Precision Floating-Point to Single-Precision Floating Point
- VCVTPS2PH—Convert Single-Precision Floating-Point to Half-Precision Floating Point

These two instructions are provided for the purpose of moving data to or from memory, while converting a single-precision floating point operand to a half-precision floating-point operand or vice versa in one instruction. These instructions allow the storage of floating point data in half-precision format, thereby conserving memory space. These instructions have both 128-bit and 256-bit forms utilizing the three-byte VEX prefix (C4h).

4.7.3 Data Reordering

The floating-point data-reordering instructions insert, extract, pack, unpack and interleave, or shuffle the elements of vector operands.

4.7.3.1 Insertion and Extraction from XMM Registers

These instructions simplify data insertion and extraction between general-purpose registers (GPR) and XMM registers. When accessing memory, no alignment is required for these instructions (unless alignment checking is enabled).

- (V)EXTRACTPS—Extracts a single-precision floating-point value from any doubleword offset in an XMM register and stores the result to memory or a general-purpose register.
- (V)INSERTPS—Inserts a single floating-point value from either a 32-bit memory location or from a specified element in an XMM register to a selected element in the destination XMM register based on a mask specified in an immediate byte. In addition, the ZMASK field in the mask allows the insertion of +0.0 into any element in the destination. In the legacy form, any doublewords in destination that do not receive either a selected doubleword from the source or +0.0 based on the ZMASK field are not modified. In the extended form, these doublewords are copied from another XMM register.

4.7.3.2 Packed Blending

These instructions conditionally copy a data element in a source operand to the same element in the destination.

- (V)BLENDPS—Packed Single-Precision Floating-Point
- (V)BLENDPD—Packed Double-Precision Floating-Point
- (V)BLENDVPS—Packed Variable Blend Single-Precision Floating-Point
- (V)BLENDVPD—Packed Variable Blend Double-Precision Floating-Point

(V)BLENDPS, (V)BLENDPD, (V)BLENDVPS, and (V)BLENDVPD copy single- or double-precision floating point elements from either of two source operands to the specified destination register based on selector bits in a mask. The mask for (V)BLENDPS, (V)BLENDPD is contained in an immediate byte. For (V)BLENDVPS and (V)BLENDVPD the mask is composed of the sign bits of the floating-point elements of an operand register. The variable blend instructions BLENDVPS and PBLENDVPD use the implicit operand XMM0 to provide the selector mask.

In the legacy form of these instructions, the first source XMM register is also the destination. The extended form specifies a separate destination XMM register encoded in the instruction.

The VBLENDPS, VBLENDVPS, VBLENDPD, and VBLENDVPD instructions have 256-bit forms which copy eight or four selected floating-point elements from one of two 256-bit source operands to the destination YMM register.

4.7.3.3 Unpack and Interleave

These instructions interleave vector elements from the high or low halves of two floating-point source operands.

- (V)UNPCKHPS—Unpack High Single-Precision Floating-Point
- (V)UNPCKHPD—Unpack High Double-Precision Floating-Point
- (V)UNPCKLPS—Unpack Low Single-Precision Floating-Point
- (V)UNPCKLPD—Unpack Low Double-Precision Floating-Point

The (V)UNPCKHPx instructions copy the high-order two (four, in the 256-bit form) single-precision floating-point values ((V)UNPCKHPS) or one (two) double-precision floating-point value(s) ((V)UNPCKHPD) in the first and second source operands and interleave them in the destination register. The low-order 64 bits of the source operands are ignored. The first source is an XMM (YMM) register and the second is either an XMM (YMM) or a 128-bit (256-bit) memory location.

The (V)UNPCKLPx instructions are analogous to their high-element counterparts except that they take elements from the low quadword of each source vector and ignore elements in the high quadword.

In the legacy form of these instructions, the first source XMM register is also the destination. In the extended form, a separate destination XMM (YMM) register is specified via the instruction encoding.

Figure 4-42 below shows an example of one of these instructions, (V)UNPCKLPS. The elements written to the destination register are taken from the low half of the source operands. Note that elements from operand 2 are placed to the left of elements from operand 1.

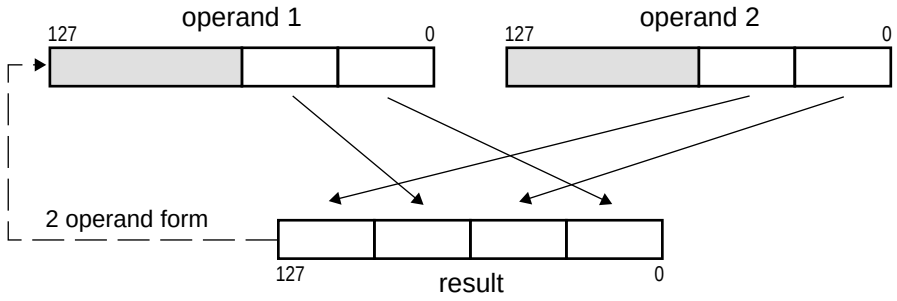


Figure 4-42. (V)UNPCKLPS Unpack and Interleave Operation

4.7.3.4 Shuffle

These instructions reorder the elements of a vector.

- (V)SHUFPS—Shuffle Packed Single-Precision Floating-Point
- (V)SHUFPD—Shuffle Packed Double-Precision Floating-Point

The (V)SHUFPS instruction moves any two of the four single-precision floating-point values in the first source operand to the low-order quadword of the destination and moves any two of the four

single-precision floating-point values in the second source operand to the high-order quadword of the destination. The source element selected for each doubleword of the destination is determined by a 2-bit field in an immediate byte.

Figure 4-43 below shows the (V)SHUFPS shuffle operation. (V)SHUFPS is useful, for example, in color imaging when computing alpha saturation of RGB values. In this case, (V)SHUFPS can replicate an alpha value in a register so that parallel comparisons with three RGB values can be performed.

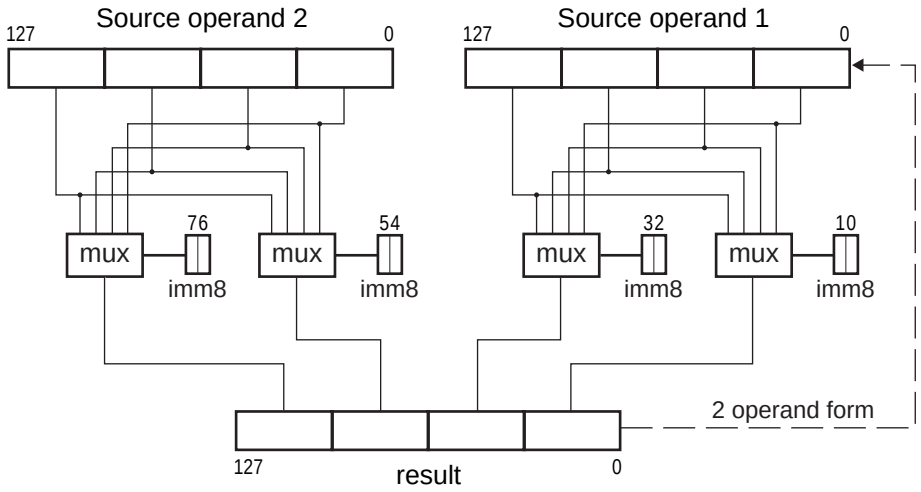


Figure 4-43. (V)SHUFPS Shuffle Operation

The (V)SHUFPD instruction moves either of the two double-precision floating-point values in the first source operand to the low-order quadword of the destination and moves either of the two double-precision floating-point values in the second source operand to the high-order quadword of the destination. The source element selected for each doubleword of the destination is determined by a bit field in an immediate byte.

For both instructions the first source operand is an XMM register and the second is either an XMM register or a 128-bit memory location. In the legacy form of these instructions, the first source XMM register is also the destination. In the extended form, a separate destination XMM register is specified via the instruction encoding.

The 256-bit forms of VSHFPS and VSHUFPD replicate the operation of each instruction's 128-bit form on the high-order octword of the 256-bit operands. The destination is a YMM register.

4.7.3.5 Fraction Extract

The fraction extract instructions isolate the fractional portions of vector or scalar floating point operands. The XOP instruction set provides the following fraction extract instructions:

- VFRCZPD—Extract Fraction Packed Double-Precision Floating-Point
- VFRCZPS—Extract Fraction Packed Single-Precision Floating-Point

- VFRCZSD— Extract Fraction Scalar Double-Precision Floating-Point
- VFRCZSS— Extract Fraction Scalar Single-Precision Floating Point

The result of the VFRCZPD and VFRCZPS instructions is a vector of integer numbers. The result of the VFRCZSD and VFRCZSS instructions is a scalar integer number.

The VFRCZPD and VFRCZPS instructions extract the fractional portions of a vector of double-precision or single-precision floating-point values in an XMM or YMM register or a 128-bit or 256-bit memory location and write the results in the corresponding field in the destination register.

The VFRCZSS and VFRCZSD instructions extract the fractional portion of the single-precision or double-precision scalar floating-point value in an XMM register or 32-bit or 64-bit memory location and writes the result in the lower element of the destination register. The upper elements of the destination XMM register are unaffected by the operation, while the upper 128 bits of the corresponding YMM register are cleared to zeros.

4.7.4 Arithmetic

The floating-point arithmetic instructions operate on two vector or scalar floating-point operands and produce a floating-point result of the same data type. For two operand forms, the result overwrites the first source operand. Vector arithmetic instructions apply the same arithmetic operation on pairs of elements from two floating-point vector operands and produce a vector result. Figure 4-44 below provides a schematic for the vector floating-point arithmetic instructions. The figure depicts 4 elements in each operand, but the actual number can be 2, 4, or 8 depending on the element size (either single precision or double precision) and vector size (128 bits or 256 bits). Each arithmetic instruction performs a unique arithmetic operation.

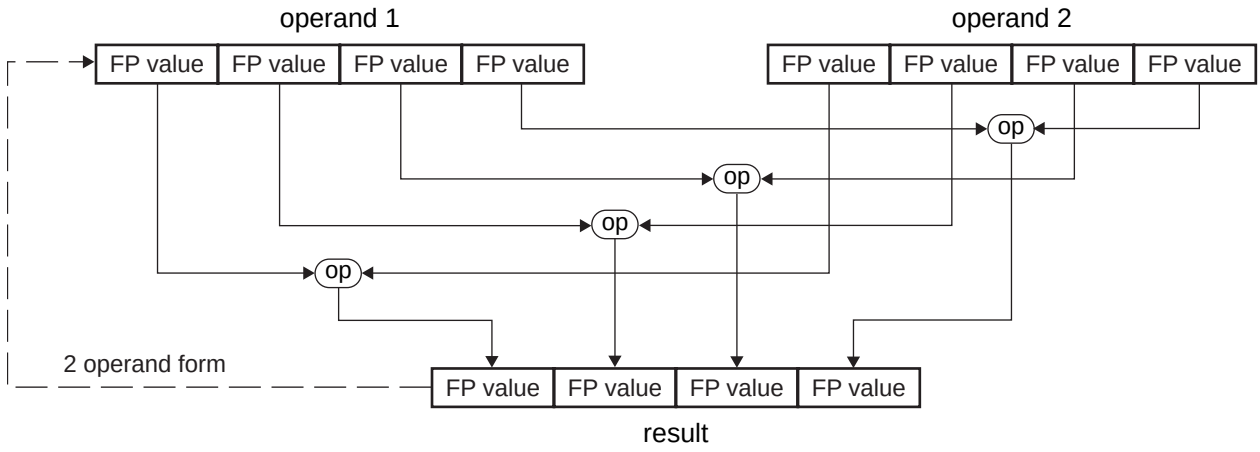


Figure 4-44. Vector Arithmetic Operation

The extended SSE versions of the arithmetic instructions that operate on packed data types support 256-bit data types. For the vector instructions this means that both the operands and the results have

twice the number of elements as the 128-bit forms. Legacy SSE instructions and extended scalar instructions support only 128-bit operands.

4.7.4.1 Addition

- (V)ADDPS—Add Packed Single-Precision Floating-Point
- (V)ADDPD—Add Packed Double-Precision Floating-Point
- (V)ADDSS—Add Scalar Single-Precision Floating-Point
- (V)ADDSD—Add Scalar Double-Precision Floating-Point

The (V)ADDPS instruction adds each of four (eight, for 256-bit form) single-precision floating-point values in the first source operand (an XMM or YMM register) to the corresponding single-precision floating-point values in the second source operand (either a YMM/XMM register or a 128-bit or 256-bit memory location) and writes the result in the corresponding doubleword of the destination.

Figure 4-45 below provides a schematic representation of the (V)ADDPS instruction. The instruction performs four addition operations in parallel. The 256-bit form of VADDPS doubles the number of operations and result elements to eight.

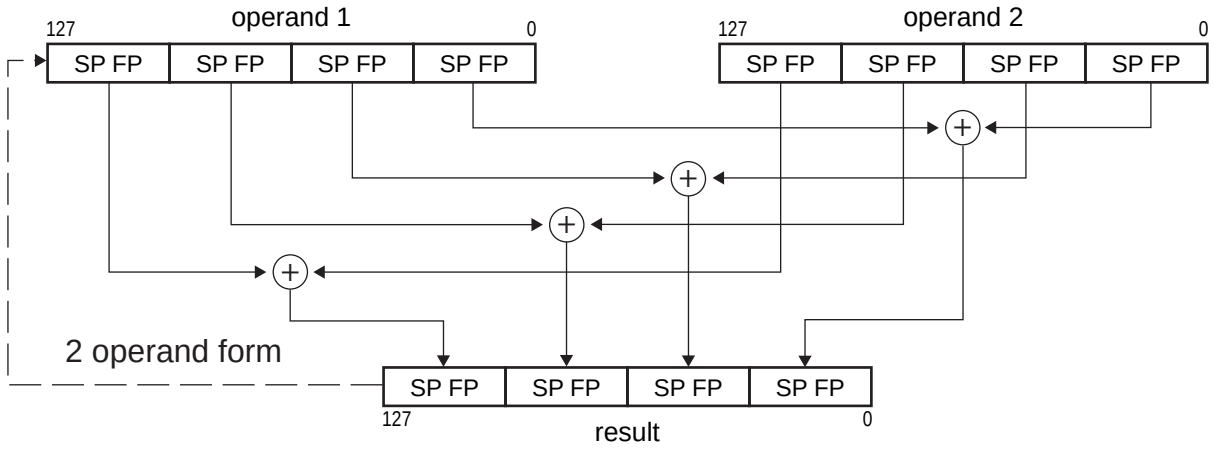


Figure 4-45. (V)ADDPS Arithmetic Operation

The (V)ADDPD instruction performs an analogous operation for double-precision floating-point values.

(V)ADDSS and (V)ADDSD operate respectively on single-precision and double-precision floating-point (scalar) values in the low-order bits of their operands. Each adds two floating-point values together and produces a single floating-point result. These extended instructions VADDSS and VADDSD have no 256-bit form.

The (V)ADDSS instruction adds the single-precision floating-point value in the first source operand (an XMM register) to the single-precision floating-point value in the second source operand (an XMM register or a doubleword memory location) and writes the result in the low-order doubleword of the

destination XMM register. For the legacy form, the three high-order doublewords of the destination are not modified. VADDSS copies the three high-order doublewords of the source operand to the destination.

The (V)ADDSD instruction adds the double-precision floating-point value in the first source operand (an XMM register) to the double-precision floating-point value in the low-order quadword of the second source operand (an XMM register or quadword memory location) and writes the result in the low-order quadword of the destination XMM register. For the legacy form, the high-order quadword of the destination is not modified. VADDSD copies the high-order quadword of the source operand to the destination.

For the legacy instructions, the first source register is also the destination. In the extended form, a separate destination XMM or YMM register is specified via the instruction encoding.

4.7.4.2 Horizontal Addition

- (V)HADDPS—Horizontal Add Packed Single-Precision Floating-Point
- (V)HADDPD—Horizontal Subtract Packed Double-Precision Floating-Point

The (V)HADDPS instruction adds the single-precision floating point values in the first and second doublewords of the first source operand (an XMM register) and stores the sum in the first doubleword of the destination XMM register. It adds the single-precision floating point values in the third and fourth doublewords of the first source operand and stores the sum in the second doubleword of the destination XMM register. It adds the single-precision floating point values in the first and second doublewords of the second source operand (either an XMM register or a 128-bit memory location) and stores the sum in the third doubleword of the destination XMM register. It adds single-precision floating point values in the third and fourth doublewords of the second source operand and stores the sum in the fourth doubleword of the destination XMM register.

The (V)HADDPD instruction adds the two double-precision floating point values in the upper and lower quadwords of the first source operand (an XMM register) and stores the sum in the first quadword of the destination XMM register. It adds the two values in the upper and lower quadwords of the second source operand (either an XMM register or a 128-bit memory location) and stores the sum in the second quadword of the destination XMM register.

The 256-bit forms of VHADDPS and VHADDPD perform the same operation as described on both the upper and lower octword of the 256-bit source operands and store the result to the destination YMM register.

For the legacy instructions, the first source register is also the destination. For the extended instructions, a separate destination register is specified by the instruction encoding.

4.7.4.3 Subtraction

- (V)SUBPS—Subtract Packed Single-Precision Floating-Point
- (V)SUBPD—Subtract Packed Double-Precision Floating-Point
- (V)SUBSS—Subtract Scalar Single-Precision Floating-Point

- (V)SUBSD—Subtract Scalar Double-Precision Floating-Point

The (V)SUBPS instruction subtracts each of four (eight, for 256-bit form) single-precision floating-point values in the second source operand (either a YMM/XMM register or a 128-bit or 256-bit memory location) from the corresponding single-precision floating-point value in the first source operand (an XMM or YMM register) and writes the result in the corresponding quadword of the destination XMM or YMM register. For vectors of *n* number of elements, the operations are:

```
operand1[i] = operand1[i] - operand2[i]
```

where: i = 0 to n – 1

The (V)SUBPD instruction performs an analogous operation for two (four, for 256-bit form) double-precision floating-point values.

(V)SUBSS and (V)SUBSD operate respectively on single-precision and double-precision floating-point (scalar) values in the low-order bits of their operands. Each subtracts the floating-point value in the second source operand from the first and produces a single floating-point result. The extended instructions VADDSS and VADDSD have no 256-bit form.

The (V)SUBSS instruction subtracts the single-precision floating-point value in the second source operand (an XMM register or a doubleword memory location) from the single-precision floating-point value in the first source operand (an XMM register) and writes the result in the low-order doubleword of the destination (an XMM register). In the legacy form, the three high-order doublewords of the destination are not modified. VADDSS copies the upper three doublewords of the source to the destination.

The (V)SUBSD instruction subtracts the double-precision floating-point value in the second source operand (an XMM register or a quadword memory location) from the double-precision floating-point value in the first source operand (an XMM register) and writes the result in the low-order quadword of the destination. In the legacy form, the high-order quadword of the destination is not modified. VADDSD copies the upper quadword of the source to the destination.

For the legacy instructions, the first source register is also the destination. In the extended form, a separate destination register is specified via the instruction encoding.

4.7.4.4 Horizontal Subtraction

- (V)HSUBPS—Horizontal Subtract Packed Single-Precision Floating-Point
- (V)HSUBPD—Horizontal Subtract Packed Double-Precision Floating-Point

The (V)HSUBPS instruction subtracts the single-precision floating-point value in the second doubleword of the first source operand (an XMM register) from that in the first doubleword of the first source operand and stores the result in the first doubleword of the destination XMM register. It subtracts the fourth doubleword of the first source operand from the third doubleword of the first source operand and stores the result in the second doubleword of the destination. It subtracts the single-precision floating-point value in the second doubleword of the second source operand (an XMM register or 128-bit memory location) from that in the first doubleword of the second source

operand and stores the result in the third doubleword of the destination register. It subtracts the fourth doubleword of the second source operand from the third doubleword of the second source operand and stores the result in the fourth doubleword of the destination.

The (V)HSUBPD instruction subtracts the double-precision floating-point value in the upper quadword of the first source operand (an XMM register) from that in the lower quadword of the first source operand and stores the difference in the low-order quadword of the destination XMM register. The difference from the subtraction of the double-precision floating-point value in the upper quadword of the second source operand (an XMM register or 128-bit memory location) from that in the lower quadword of the second source operand is stored in the second quadword of the destination operand.

VHSUBPS and VHSUBPD each have a 256-bit form. For these instructions the first source operand is a YMM register and the second is either a YMM or a 256-bit memory location. These instructions perform the same operation as their 128-bit counterparts on both the lower and upper quadword of their operands.

For the legacy instructions, the first source register is also the destination. For the extended instructions, a separate destination register is specified by the instruction encoding.

4.7.4.5 Horizontal Search

- (V)PHMINPOSUW—Packed Horizontal Minimum and Position Unsigned Word

(V)PHMINPOSUW finds the value and location of the minimum unsigned word from one of 8 horizontally packed unsigned words in its source operand (an XMM register or a 128-bit memory location). The resulting value and location (offset within the source) are packed into the low doubleword of the destination XMM register. Video encoding can be improved by using (V)MPSADBW and (V)PHMINPOSUW together.

4.7.4.6 Simultaneous Addition and Subtraction

- (V)ADDSUBPS—Add/Subtract Packed Single-Precision Floating-Point
- (V)ADDSUBPD—Add/Subtract Packed Double-Precision Floating-Point

The (V)ADDSUBPS instruction adds two (four, for the 256-bit form) pairs of odd-indexed single-precision floating-point elements from the source operands and writes the sums to the corresponding elements of the destination; subtracts the even-indexed elements of the second operand from the corresponding elements of the first operand and writes the differences to the corresponding elements of the destination. The first source operand is an XMM (YMM) register and the second operand is either an XMM (YMM) register or a 128-bit (256-bit) memory location. For the legacy instruction, the first source operand is also the destination. For the extended forms, the result is written to the specified separate destination YMM/XMM register.

The (V)ADDSUBPS instruction adds one (two, for the 256-bit form) pair(s) of odd-indexed double-precision floating-point elements from the source operands and writes the sums to the corresponding elements of the destination; subtracts the even-indexed elements of the second operand from the corresponding elements of the first operand and writes the differences to the corresponding elements of the destination.The first source operand is an XMM (YMM) register and the second operand is

either an XMM (YMM) register or a 128-bit (256-bit) memory location. For the legacy instruction, the first source operand is also the destination. For the extended forms, the result is written to the specified separate destination YMM/XMM register.

4.7.4.7 Multiplication

- (V)MULPS—Multiply Packed Single-Precision Floating-Point
- (V)MULPD—Multiply Packed Double-Precision Floating-Point
- (V)MULSS—Multiply Scalar Single-Precision Floating-Point
- (V)MULSD—Multiply Scalar Double-Precision Floating-Point

The (V)MULPS instruction multiplies each of four (eight for the 256-bit form) single-precision floating-point values in the first source operand (XMM or YMM register) operand by the corresponding single-precision floating-point value in the second source operand (either a register or a memory location) and writes the result to the corresponding doubleword of the destination XMM (YMM) register. The (V)MULPD instruction performs an analogous operation for two (four) double-precision floating-point values.

VMULSS and VMULLSD have no 256-bit form.

The (V)MULSS instruction multiplies the single-precision floating-point value in the low-order doubleword of the first source operand (an XMM register) by the single-precision floating-point value in the low-order doubleword of the second source operand (an XMM register or a 32-bit memory location) and writes the result in the low-order doubleword of the destination XMM register. MULSS leaves the three high-order doublewords of the destination unmodified. VMULSS copies the three high-order doublewords of the first source operand to the destination.

The (V)MULSD instruction multiplies the double-precision floating-point value in the low-order quadword of the first source operand (an XMM register) by the double-precision floating-point value in the low-order quadword of the second source operand (an XMM register or a 64-bit memory location) and writes the result in the low-order quadword of the destination XMM register. MULSD leaves the high-order quadword of the destination unmodified. VMULSD copies the upper quadword of the first source operand to the destination.

For the legacy instructions, the first source register is also the destination. For the extended instructions, a separate destination register is specified by the instruction encoding.

4.7.4.8 Division

- (V)DIVPS—Divide Packed Single-Precision Floating-Point
- (V)DIVPD—Divide Packed Double-Precision Floating-Point
- (V)DIVSS—Divide Scalar Single-Precision Floating-Point
- (V)DIVSD—Divide Scalar Double-Precision Floating-Point

The (V)DIVPS instruction divides each of the four (eight for the 256-bit form) single-precision floating-point values in the first source operand (an XMM or a YMM register) by the corresponding

single-precision floating-point value in the second source operand (either a register or a memory location) and writes the result in the corresponding doubleword of the destination XMM (YMM) register. For vectors of *n* number of elements, the operations are:

```
operand1[i] = operand1[i] ÷ operand2[i]
```

where: i = 0 to n – 1

The (V)DIVPD instruction performs an analogous operation for two (four) double-precision floating-point values.

VDIVSS and VDIVSD have no 256-bit form.

The (V)DIVSS instruction divides the single-precision floating-point value in the low-order doubleword of the first source operand (an XMM register) by the single-precision floating-point value in the low-order doubleword of the second source operand (an XMM register or a 32-bit memory location) and writes the result in the low-order doubleword of the destination XMM register. DIVSS leaves the three high-order doublewords of the destination unmodified. VDIVSS copies the three high-order doublewords of the first source operand to the destination.

The (V)DIVSD instruction divides the double-precision floating-point value in the low-order quadword of the first source operand (an XMM register) by the double-precision floating-point value in the low-order quadword of the second source operand (an XMM register or a 64-bit memory location) and writes the result in the low-order quadword of the destination XMM register. DIVSS leaves the high-order quadword of the destination unmodified. VDIVSD copies the upper quadword of the first source operand to the destination.

For the legacy instructions, the first source XMM register is also the destination. For the extended instructions, a separate destination register is specified by the instruction encoding.

If accuracy requirements allow, convert floating-point division by a constant to a multiply by the reciprocal. Divisors that are powers of two and their reciprocals are exactly representable, and therefore do not cause an accuracy issue, except for the rare cases in which the reciprocal overflows or underflows.

4.7.4.9 Square Root

- (V)SQRTPS—Square Root Packed Single-Precision Floating-Point
- (V)SQRTPD—Square Root Packed Double-Precision Floating-Point
- (V)SQRTSS—Square Root Scalar Single-Precision Floating-Point
- (V)SQRTSD—Square Root Scalar Double-Precision Floating-Point

The (V)SQRTPS instruction computes the square root of each of four (eight for the 256-bit form) single-precision floating-point values in the source operand (an XMM register or 128-bit memory location) and writes the result in the corresponding doubleword of the destination XMM (YMM) register. The (V)SQRTPD instruction performs an analogous operation for two double-precision floating-point values.

VSQRTSS and VSQRTSD have no 256-bit form.

The (V)SQRTSS instruction computes the square root of the low-order single-precision floating-point value in the source operand (an XMM register or 32-bit memory location) and writes the result in the low-order doubleword of the destination XMM register. SQRTSS leaves the three high-order doublewords of the destination XMM register unmodified. VSQRTSS copies the three high-order doublewords of the first source operand to the destination.

The (V)SQRTSD instruction computes the square root of the low-order double-precision floating-point value in the source operand (an XMM register or 64-bit memory location) and writes the result in the low-order quadword of the destination XMM register. SQRTSD leaves the high-order quadword of the destination XMM register unmodified. VSQRTSD copies the upper quadword of the first source operand to the destination.

For the legacy instructions, the first source XMM register is also the destination. For the extended instructions, a separate destination register is specified by the instruction encoding.

4.7.4.10 Reciprocal Square Root

- (V)RSQRTPS—Reciprocal Square Root Packed Single-Precision Floating-Point
- (V)RSQRTSS—Reciprocal Square Root Scalar Single-Precision Floating-Point

The (V)RSQRTPS instruction computes the approximate reciprocal of the square root of each of four (eight for the 256-bit form) single-precision floating-point values in the source operand (an XMM register or 128-bit memory location) and writes the result in the corresponding doubleword of the destination XMM (YMM) register.

The (V)RSQRTSS instruction computes the approximate reciprocal of the square root of the low-order single-precision floating-point value in the source operand (an XMM register or 32-bit memory location) and writes the result in the low-order doubleword of the destination XMM register. RSQRTSS leaves the three high-order doublewords in the destination XMM register unmodified. VRSQRTSS copies the three high-order doublewords from the source operand to the destination.

For the legacy instructions, the first source register is also the destination. For the extended instructions, a separate destination register is specified by the instruction encoding.

For both (V)RSQRTPS and (V)RSQRTSS, the maximum relative error is less than or equal to $1.5 * 2^{-12}$.

4.7.4.11 Reciprocal Estimation

- (V)RCPPS—Reciprocal Packed Single-Precision Floating-Point
- (V)RCPSS—Reciprocal Scalar Single-Precision Floating-Point

The (V)RCPPS instruction computes the approximate reciprocal of each of the four (eight, for the 256-bit form) single-precision floating-point values in the source operand (an XMM register or 128-bit memory location) and writes the result in the corresponding doubleword of the destination XMM (YMM) register.

The (V)RCPSS instruction computes the approximate reciprocal of the low-order single-precision floating-point value in the source operand (an XMM register or 32-bit memory location) and writes the result in the low-order doubleword of the destination XMM register. RCPSS leaves the three high-order doublewords in the destination unmodified. VRCPSS copies the three high-order doublewords from the source to the destination

For the legacy instructions, the first source register is also the destination. For the extended instructions, a separate destination register is specified by the instruction encoding.

For both (V)RCPPS and (V)RCPSS, the maximum relative error is less than or equal to $1.5 * 2^{-12}$.

4.7.4.12 Dot Product

- (V)DPPS—Dot Product Single-Precision Floating-Point
- (V)DPPD—Dot Product Double-Precision Floating-Point

The (V)DPPS instruction computes one (two, for the 256-bit form) single-precision dot product(s), selectively summing one, two, three, or four products of the corresponding source elements of the source operands and then copies this dot product to any combination of four elements in (the upper and lower octword of) the destination. An immediate byte selects which products are computed and to which elements of the destination the dot product is copied. The 256-bit form utilizes the single immediate byte to control the computation of both the upper and the lower octword of the result.

The first source operand is an XMM (YMM) register. The second source operand is either an XMM register or a 128-bit memory location (YMM register or a 256-bit memory location). For the legacy instructions, the first source register is also the destination. For the extended instructions, a separate destination register is specified by the instruction encoding.

The (V)DPPD instruction performs the analogous operation on packed double-precision floating-point operands.

As an example, a single DPPS instruction can be used to compute a two, three, or four element dot product. A single 256-bit VDPPS instruction can be used to compute two dot products of up to four elements each.

4.7.4.13 Floating-Point Round Instructions with Selectable Rounding Mode

- (V)ROUNDPS—Round Packed Single-Precision Floating-Point
- (V)ROUNDPD—Round Packed Double-Precision Floating-Point
- (V)ROUNDSS—Round Scalar Single-Precision
- (V)ROUNDSD—Round Scalar Double-Precision

High level languages and libraries often expose rounding operations that have a variety of numeric rounding and exception behaviors. These four rounding instructions cover scalar and packed single and double-precision floating-point operands. The rounding mode can be selected using one of the IEEE-754 modes (Nearest, -Inf, +Inf, and Truncate) without changing the current rounding mode. Alternately, the instruction can be forced to use the current rounding mode.

The (V)ROUNDPS and (V)ROUNDPD instructions round each of the four (eight, for the 256-bit form) single-precision values or two (four) double-precision values in the source operand (either an XMM register or a 128-bit memory location or, for the 256-bit form, a YMM register or 256-bit memory location) based on controls in an immediate byte and write the results to the respective elements of the destination XMM (YMM) register.

The (V)ROUNDSS and (V)ROUNDSD instructions round the single-precision or double-precision floating-point value from the source operand based on the rounding control specified in an immediate byte and write the results to the low-order doubleword or quadword of an XMM register. The source operand is either the low-order doubleword or quadword of an XMM register or a 32-bit or 64-bit memory location. For the legacy forms of these instructions, the upper three doublewords or high-order quadword of the destination are not modified. VROUNDSS copies the upper three doublewords of a second XMM register to the destination. VROUNDSD copies the high-order quadword of a second XMM register to the destination.

4.7.5 Fused Multiply-Add Instructions

The fused multiply-add (FMA) instructions comprise AMD's four operand FMA4 instruction set and the three operand FMA instruction set.

The FMA instructions provide a set of fused multiply-add mathematical operations. The basic FMA instruction performs a multiply of two floating-point scalar or vector operands followed by a second operation in which the product of the first operation is added to a third scalar or vector floating-point operand. The result is rounded to the precision of the source operands and stored in either a distinct destination register or in the register that sourced the first operand. Variants of the basic FMA operation allow for the negation (sign inversion) of the products and the negation of the third scalar operand or elements within the third vector operand.

Figure 4-46 below provides a schematic representation of the scalar FMA instructions. Note that the (× 1/-1) operator in the diagram denotes a negation (sign inversion) operation performed in some of the instructions.

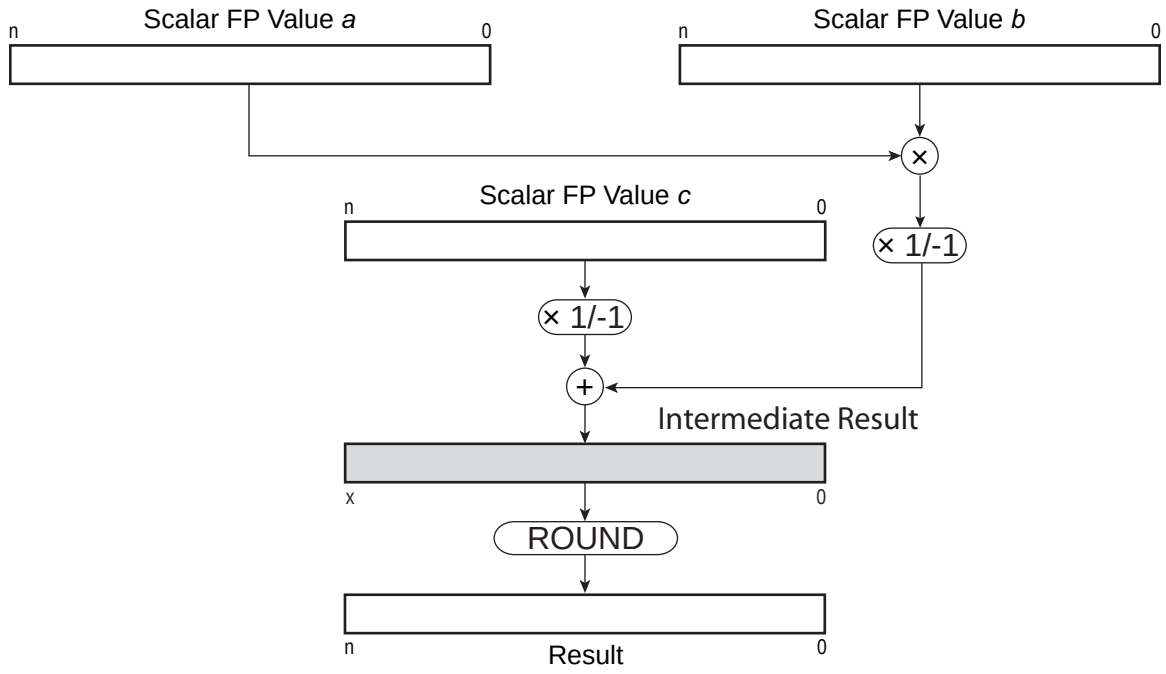


Figure 4-46. Scalar FMA Instructions

Figure 4-47 below provides a schematic representation of the vector FMA instructions. Note that the (× 1/-1) operator in the diagram denotes a negation (sign inversion) operation performed in some of the instructions. For illustrative purposes, four-element vectors are shown in the figure. The 128- and 256-bit data types support from 2 to 8 elements per vector.

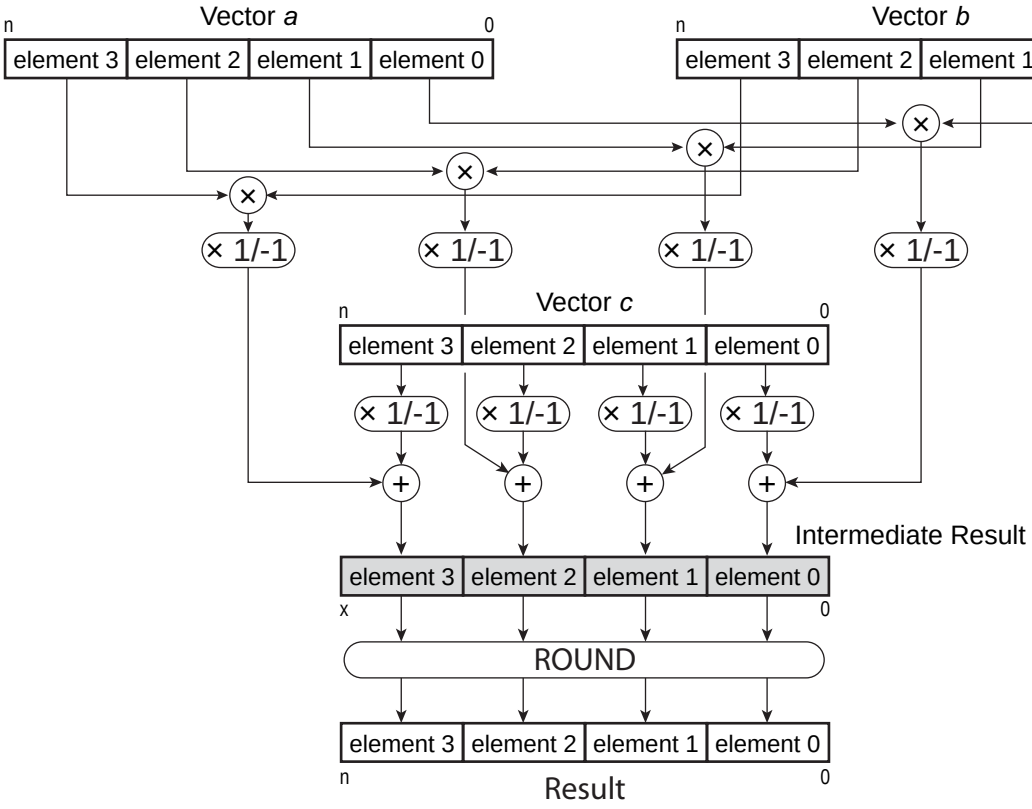


Figure 4-47. Vector FMA Instructions

Fused multiply-add instructions can improve the performance and accuracy of many computations that involve the accumulation of multiple products, such as the dot product operation and matrix multiplication. Intermediate results may utilize a non-standard (higher) precision (using more significant bits) than the standard single-precision or double-precision floating-point formats allow. A fused multiply-add can be faster and more precise than the equivalent operations performed serially because the step of rounding intermediate results can be skipped.

The FMA4 instructions support the specification of three operand sources (YMM/XMM registers or memory) and a distinct destination register. This enables non-destructive computations where the result does not overwrite one of the source operand registers. For the three operand FMA instructions the result always overwrites the first source register. Variants within the set allow for the negation (sign inversion) of operands or vector operand elements and/or intermediate values.

Six basic instruction variants are defined. These are:

- Fused multiply-add of scalar and vector (packed) single- and double-precision floating-point values:
- Fused multiply-alternating add/subtract of vector (packed) single- and double-precision floating-point values
- Fused multiply-alternating subtract/add of vector (packed) single- and double-precision floating-point values
- Fused multiply-subtract of scalar and vector (packed) single- and double-precision floating-point values
- Fused negative multiply-add of scalar and vector (packed) single- and double-precision floating-point values
- Fused negative multiply-subtract of scalar and vector (packed) single- and double-precision floating-point values

Note that a scalar operation is not defined for the fused multiply-alternating add/subtract and the fused multiply-alternating subtract/add instructions. Each variant will be discussed below.

4.7.5.1 Operand Source Specification

Each instruction operates on three operands to produce a result. Individual instruction forms within a variant allow either the second or third operand to be sourced from memory. In the following descriptions, the first operand will be referred to as operand *a*, the second will be referred to as operand *b* and the third operand *c*.

The instruction syntax for specifying this alternate sourcing of the second and third operands differs between the FMA4 and the three operand FMA instructions.

The FMA4 instructions use the same instruction mnemonic allowing the memory operand source to appear in either the second or third operand position:

```
VF(N)Moptxx dest_reg, src_reg1, src_reg2/mem, src_reg3
VF(N)Moptxx dest_reg, src_reg1, src_reg2, src_reg3/mem
```

The three-operand instructions utilize three different instruction mnemonics having the following syntax:

```
VF(N)Mopt132xx src_reg1, src_reg2, src_reg3/mem
VF(N)Mopt213xx src_reg1, src_reg2, src_reg3/mem
VF(N)Mopt231xx src_reg1, src_reg2, src_reg3/mem
```

Where *opt* represents the instruction operation and *xx* represents the operand data type.

Operand sourcing for each instruction form is described in the following table:

Figure 4-48. Operand Source / Destination Specification

Instruction	Operand a	Operand b	Operand c	Result
VF(N)M*opt**xx***	*src_reg1*	*src_reg2 / mem*	*src_reg3*	*dest_reg*
VF(N)M*opt**xx***	*src_reg1*	*src_reg2*	*src_reg3 / mem*	*dest_reg*
VF(N)M*opt*132*xx*	*src_reg1*	*src_reg3 / mem*	*src_reg2*	*src_reg1*
VF(N)M*opt*213*xx*	*src_reg2*	*src_reg1*	*src_reg3 / mem*	*src_reg1*
VF(N)M*opt*231*xx*	*src_reg2*	*src_reg3 / mem*	*src_reg1*	*src_reg1*

The specific operations performed by the six variants are described in the next sections.

4.7.5.2 Multiply and Add Instructions

VFMADDPD / VFMADD132PD / VFMADD213PD / VFMADD231PD

Multiplies together the double-precision floating-point vectors *a* and *b*, adds the product to the double-precision floating-point vector *c*, and performs rounding to produce the double-precision floating-point vector result.

VFMADDPS / VFMADD132PS / VFMADD213PS / VFMADD231PS

Multiplies together the single-precision floating-point vectors *a* and *b*, adds the product to the single-precision floating-point vector *c*, and performs rounding to produce the single-precision floating-point vector result.

VFMADDSD / VFMADD132SD / VFMADD213SD / VFMADD231SD

Multiplies together the double-precision floating-point scalars *a* and *b*, adds the product to the double-precision floating-point scalar *c*, and performs rounding to produce the double-precision floating-point scalar result.

VFMADDSS / VFMADD132SS / VFMADD213SS / VFMADD231SS

Multiplies together the single-precision floating-point scalars *a* and *b*, adds the product to the single-precision floating-point scalar *c*, and performs rounding to produce the single-precision floating-point scalar result.

4.7.5.3 Multiply with Alternating Add/Subtract Instructions

VFMADDSUBPD / VFMADDSUB132PD / VFMADDSUB213PD / VFMADDSUB231PD

Multiplies together the double-precision floating-point vectors *a* and *b*, adds each odd-numbered element of the double-precision floating-point vector *c* to the corresponding element of the product, subtracts each even-numbered element of double-precision floating-point vector *c* from the corresponding element of the product, and performs rounding to produce the double-precision floating-point vector result.

VFMADDSUBPS / VFMADDSUB132PS / VFMADDSUB213PS / VFMADDSUB231PS

Multiplies together the single-precision floating-point vectors *a* and *b*, adds each odd-numbered element of the single-precision floating-point vector *c* to the corresponding element of the product, subtracts each even-numbered element of single-precision floating-point vector *c* from the corresponding element of the product, and performs rounding to produce the single-precision floating-point vector result.

4.7.5.4 Multiply with Alternating Subtract/Add Instructions

VFMSUBADDPD / VFMSUBADD132PD / VFMSUBADD213PD / VFMSUBADD231PD

Multiplies together the double-precision floating-point vectors *a* and *b*, adds each even-numbered element of the double-precision floating-point vector *c* to the corresponding element of the product, subtracts each odd-numbered element of double-precision floating-point vector *c* from the corresponding element of the product, and performs rounding to produce the double-precision floating-point vector result.

VFMSUBADDPS / VFMSUBADD132PS / VFMSUBADD213PS / VFMSUBADD231PS

Multiplies together the single-precision floating-point vectors *a* and *b*, adds each even-numbered element of the single-precision floating-point vector *c* to the corresponding element of the product, subtracts each odd-numbered element of single-precision floating-point vector *c* from the corresponding element of the product, and performs rounding to produce the single-precision floating-point vector result.

4.7.5.5 Multiply and Subtract Instructions

VFMSUBPD / VFMSUB132PD / VFMSUB213PD / VFMSUB231PD

Multiplies together the double-precision floating-point vectors *a* and *b*, subtracts the double-precision floating-point vector *c* from the product, and performs rounding to produce the double-precision floating-point vector result.

VFMSUBPS / VFMSUB132PS / VFMSUB213PS / VFMSUB231PS

Multiplies together the single-precision floating-point vectors *a* and *b*, subtracts the single-precision floating-point vector *c* from the product, and performs rounding to produce the single-precision floating-point vector result.

VFMSUBSD / VFMSUB132SD / VFMSUB213SD / VFMSUB231SD

Multiplies together the double-precision floating-point scalars *a* and *b*, subtracts the double-precision floating-point scalar *c* from the product, and performs rounding to produce the double-precision floating-point result.

VFMSUBSS / VFMSUB132SS / VFMSUB213SS / VFMSUB231SS

Multiplies together the single-precision floating-point scalars *a* and *b*, subtracts the single-precision floating-point scalar *c* from the product, and performs rounding to produce the single-precision floating-point result.

4.7.5.6 Negative Multiply and Add Instructions

VFNMADDPD / VFNMADD132PD / VFNMADD213PD / VFNMADD231PD

Multiplies together the double-precision floating-point vectors *a* and *b*, negates (inverts the sign) the product, adds this intermediate result to the double-precision floating-point vector *c*, and performs rounding to produce the double-precision floating-point vector result.

VFNMADDPS / VFNMADD132PS / VFNMADD213PS / VFNMADD231PS

Multiplies together the single-precision floating-point vectors *a* and *b*, negates (inverts the sign) the product, adds this intermediate result to the single-precision floating-point vector *c*, and performs rounding to produce the single-precision floating-point vector result.

VFNMADDSD / VFNMADD132SD / VFNMADD213SD / VFNMADD231SD

Multiplies together the double-precision floating-point scalars *a* and *b*, negates (inverts the sign) the product, adds this intermediate result to the double-precision floating-point scalar *c*, and performs rounding to produce the double-precision floating-point result.

VFNMADDSS / VFNMADD132SS / VFNMADD213SS / VFNMADD231SS

Multiplies together the single-precision floating-point scalars *a* and *b*, negates (inverts the sign) the product, adds this intermediate result to the single-precision floating-point scalar *c*, and performs rounding to produce the single-precision floating-point result.

4.7.5.7 Negative Multiply and Subtract Instructions

VFNMSUBPD / VFNMSUB132PD / VFNMSUB213PD / VFNMSUB231PD

Multiplies together the double-precision floating-point vectors *a* and *b*, negates the product (inverts the sign), adds this intermediate result to the negated double-precision floating-point vector *c*, and performs rounding to produce the double-precision floating-point vector result.

VFNMSUBPS / VFNMSUB132PS / VFNMSUB213PS / VFNMSUB231PS

Multiplies together the single-precision floating-point vectors *a* and *b*, negates the product (inverts the sign), adds this intermediate result to the negated single-precision floating-point vector *c*, and performs rounding to produce the single-precision floating-point vector result.

VFNMSUBSD / VFNMSUB132SD / VFNMSUB213SD / VFNMSUB231SD

Multiplies together the double-precision floating-point scalars *a* and *b*, negates the product (inverts the sign), adds this intermediate result to the negated double-precision floating-point scalar *c*, and performs rounding to produce the double-precision floating-point result.

VFNMSUBSS / VFNMSUB132SS / VFNMSUB213SS / VFNMSUB231SS

Multiplies together the single-precision floating-point scalars *a* and *b*, negates the product (inverts the sign), adds this intermediate result to the negated single-precision floating-point scalar *c*, and performs rounding to produce the single-precision floating-point result.

4.7.6 Compare

The floating-point vector-compare instructions compare two operands, and they either write a mask, or they write the maximum or minimum value, or they set flags. Compare instructions can be used to avoid branches. Figure 4-21 on page 148 shows an example of using compare instructions.

4.7.6.1 Compare and Write Mask

- (V)CMPPS—Compare Packed Single-Precision Floating-Point
- (V)CMPPD—Compare Packed Double-Precision Floating-Point
- (V)CMPSS—Compare Scalar Single-Precision Floating-Point
- (V)CMPSD—Compare Scalar Double-Precision Floating-Point

The (V)CMPPS instruction compares each of four (eight, for 256-bit form) single-precision floating-point values in the first source operand with the corresponding single-precision floating-point value in the second source operand and writes the result in the corresponding 32 bits of the destination. The type of comparison is specified by the three (five, for the AVX forms) low-order bits of the immediate-byte operand. The result of each compare is a 32-bit value of all 1s (TRUE) or all 0s (FALSE). Some compare operations that are not directly supported by the immediate-byte encodings can be implemented by swapping the contents of the source and destination operands before executing the compare.

The (V)CMPPD instruction performs an analogous operation for two (four, for the 256-bit form) double-precision floating-point values.

The first source operand is an XMM (YMM) register. The second source operand is either an XMM register or a 128-bit memory location (YMM register or a 256-bit memory location). For the legacy form of the instructions, the first source register is also the destination. The extended form of the instructions encodes a separate destination XMM (YMM) register.

The (V)CMPSS instruction performs an analogous operation for single-precision floating-point values. The first source operand is the low-order doubleword of an XMM register. The second source operand is either the low-order doubleword of an XMM register or a 32-bit memory location. CMPSS leaves the three high-order doublewords of the destination unmodified. VCMPSS copies the three

high-order doublewords of the first source operand to the destination. For CMPSS the first source operand is also the destination.

The (V)CMPSD instruction performs an analogous operation on double-precision floating-point values. CMPSD leaves the high-order quadword of the destination XMM register unmodified. VCMPSD copies the high-order quadword of the first source operand to the destination.

Figure 4-49 shows a (V)CMPPD compare operation.

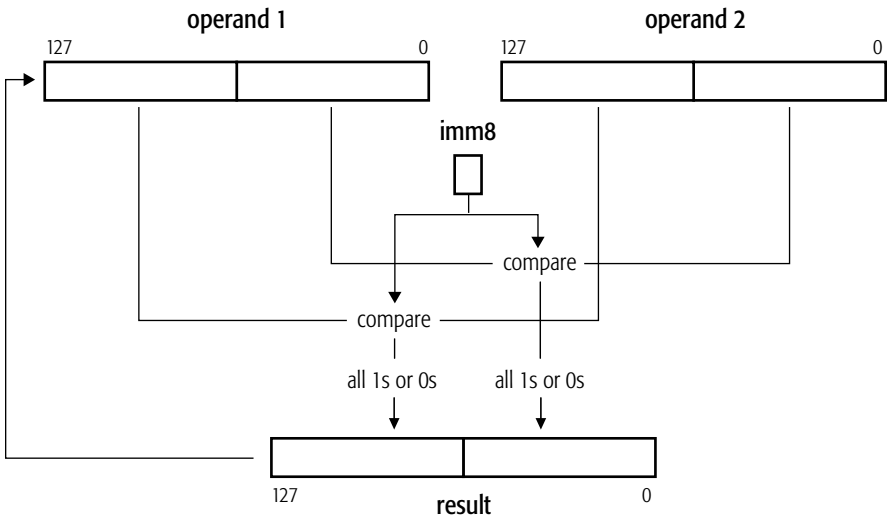


Figure 4-49. (V)CMPPD Compare Operation

4.7.6.2 Compare and Write Minimum or Maximum

- (V)MAXPS—Maximum Packed Single-Precision Floating-Point
- (V)MAXPD—Maximum Packed Double-Precision Floating-Point
- (V)MAXSS—Maximum Scalar Single-Precision Floating-Point
- (V)MAXSD—Maximum Scalar Double-Precision Floating-Point
- (V)MINPS—Minimum Packed Single-Precision Floating-Point
- (V)MINPD—Minimum Packed Double-Precision Floating-Point
- (V)MINSS—Minimum Scalar Single-Precision Floating-Point
- (V)MINSD—Minimum Scalar Double-Precision Floating-Point

The (V)MAXPS and (V)MINPS instructions compare each of four (eight, for the 256-bit form) single-precision floating-point values in the first source operand with the corresponding single-precision floating-point value in the second source operand and write the maximum or minimum, respectively, of the two (four) values to the corresponding doubleword of the destination. The (V)MAXPD and (V)MINPD instructions perform analogous operations on pairs of double-precision floating-point

values. The first source operand is an XMM (YMM) register and the second is either an XMM register or a 128-bit memory location (or for the 256-bit form, a YMM register or a 256-bit memory location). The destination for the legacy forms is the source operand register. The extended instructions specify a separate destination register in their encoding.

The (V)MAXSS and (V)MINSS instructions compare the single-precision floating-point value of the first source operand with the single-precision floating-point value in the second source operand and write the maximum or minimum, respectively, of the two values to the low-order 32 bits of the destination XMM register. The first source operand is the low-order doubleword of an XMM register and the second is either the low-order doubleword of an XMM register or a 32-bit memory location. The legacy forms do not modify the three high-order doublewords of the destination. The extended forms merge in the corresponding bits from an additional XMM source operand.

The (V)MAXSD and (V)MINSD instructions compare the double-precision floating-point value of the first source operand with the double-precision floating-point value in the second source operand and write the maximum or minimum, respectively, of the two values to the low-order quadword of the destination XMM register. The first source operand is the low-order doubleword of an XMM register and the second is either the low-order doubleword of an XMM register or a 32-bit memory location. The legacy forms do not modify the high-order quadword of the destination XMM register. The extended forms merge in the corresponding bits from an additional XMM source operand.

The destination for the legacy forms is the source operand register. The extended instructions specify a separate destination register in their encoding.

The (V)MIN*x* and (V)MAX*x* instructions are useful for clamping (saturating) values, such as color values in 3D geometry and rasterization.

4.7.6.3 Compare and Write rFLAGS

- (V)COMISS—Compare Ordered Scalar Single-Precision Floating-Point
- (V)COMISD—Compare Ordered Scalar Double-Precision Floating-Point
- (V)UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point
- (V)UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point

The (V)COMISS instruction performs an ordered compare of the single-precision floating-point value in the low-order 32 bits of the first operand with the single-precision floating-point value in the second operand (either the low-order 32 bits of an XMM register or a 32-bit memory location) and sets the zero flag (ZF), parity flag (PF), and carry flag (CF) bits in the rFLAGS register to reflect the result of the compare. The OF, AF, and SF bits in rFLAGS are set to zero.

The (V)COMISD instruction performs an analogous operation on the double-precision floating-point source operands. The (V)UCOMISS and (V)UCOMISD instructions perform an analogous, but unordered, compare operations. Figure 4-50 on page 216 shows a (V)COMISD compare operation.

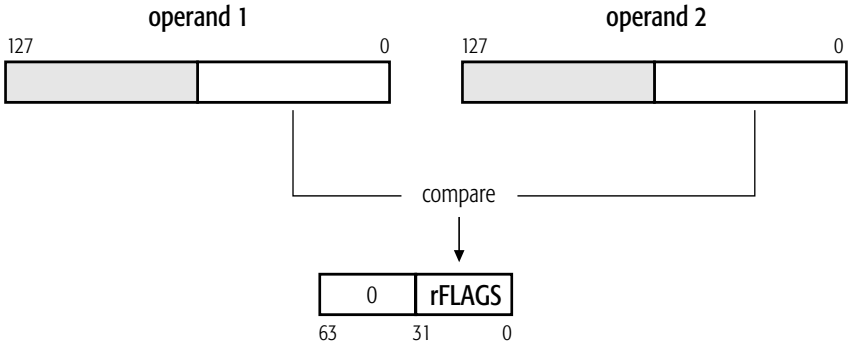


Figure 4-50. (V)COMISD Compare Operation

The difference between an ordered and unordered comparison has to do with the conditions under which a floating-point invalid-operation exception (IE) occurs. In an ordered comparison ((V)COMISS or (V)COMISD), an IE exception occurs if either of the source operands is either type of NaN (QNaN or SNaN). In an unordered comparison, the exception occurs only if a source operand is an SNaN. For a description of NaNs, see Section "Floating-Point Number Types" on page 124. For a description of exceptions, see "Exceptions" on page 218.

4.7.7 Logical

The vector-logic instructions perform Boolean logic operations, including AND, OR, and exclusive OR. The extended forms of the instructions support both 128-bit and 256-bit operands.

4.7.7.1 And

- (V)ANDPS—Logical Bitwise AND Packed Single-Precision Floating-Point
- (V)ANDPD—Logical Bitwise AND Packed Double-Precision Floating-Point
- (V)ANDNPS—Logical Bitwise AND NOT Packed Single-Precision Floating-Point
- (V)ANDNPD—Logical Bitwise AND NOT Packed Double-Precision Floating-Point

The (V)ANDPS instruction performs a logical bitwise AND of the four (eight, for the 256-bit form) packed single-precision floating-point values in the first source operand and the corresponding four (eight) single-precision floating-point values in the second source operand and writes the result to the destination. The (V)ANDPD instruction performs an analogous operation on the two (four) packed double-precision floating-point values. The (V)ANDNPS and (V)ANDNPD instructions invert the elements of the first source vector (creating a one's complement of each element), AND them with the elements of the second source vector, and write the result to the destination. The first source operand is an XMM (YMM) register. The second source operand is either another XMM (YMM) register or a 128-bit (256-bit) memory location. For the legacy instructions, the destination is also the first source operand. For the extended forms, the destination is a separately specified XMM (YMM) register.

4.7.7.2 Or

- (V)ORPS—Logical Bitwise OR Packed Single-Precision Floating-Point
- (V)ORPD—Logical Bitwise OR Packed Double-Precision Floating-Point

The (V)ORPS instruction performs a logical bitwise OR of four (eight, for the 256-bit form) single-precision floating-point values in the first source operand and the corresponding four (eight) single-precision floating-point values in the second operand and writes the result to the destination. The (V)ORPD instruction performs an analogous operation on pairs of two double-precision floating-point values. The first source operand is an XMM (YMM) register. The second source operand is either another XMM (YMM) register or a 128-bit (256-bit) memory location. For the legacy instructions, the destination is also the first source operand. For the extended forms, the destination is a separately specified XMM (YMM) register.

4.7.7.3 Exclusive Or

- (V)XORPS—Logical Bitwise Exclusive OR Packed Single-Precision Floating-Point
- (V)XORPD—Logical Bitwise Exclusive OR Packed Double-Precision Floating-Point

The (V)XORPS instruction performs a logical bitwise exclusive OR of four (eight, for the 256-bit form) single-precision floating-point values in the first operand and the corresponding four (eight) single-precision floating-point values in the second operand and writes the result to the destination. The (V)XORPD instruction performs an analogous operation on pairs of two double-precision floating-point values. The first source operand is an XMM (YMM) register. The second source operand is either another XMM (YMM) register or a 128-bit (256-bit) memory location. For the legacy instructions, the destination is also the first source operand. For the extended forms, the destination is a separately specified XMM (YMM) register.

4.8 Instruction Prefixes

Instruction prefixes, in general, are described in “Instruction Prefixes” on page 76. The following restrictions apply to the use of instruction prefixes with SSE instructions.

4.8.1 Supported Prefixes

The following prefixes can be used with SSE instructions:

- *Address-Size Override*—The 67h prefix affects only operands in memory. The prefix is ignored by all other SSE instructions.
- *Operand-Size Override*—The 66h prefix is used to form the opcodes of certain SSE instructions. The prefix is ignored by all other SSE instructions.
- *Segment Overrides*—The 2Eh (CS), 36h (SS), 3Eh (DS), 26h (ES), 64h (FS), and 65h (GS) prefixes affect only operands in memory. In 64-bit mode, the contents of the CS, DS, ES, SS segment registers are ignored.

- *REP*—The F2 and F3h prefixes do not function as repeat prefixes for the SSE instructions. Instead, they are used to form the opcodes of certain SSE instructions. The prefixes are ignored by all other SSE instructions.
- *REX*—The REX prefixes affect operands that reference a GPR or XMM register when running in 64-bit mode. It allows access to the full 64-bit width of any of the 16 extended GPRs and to any of the 16 extended XMM registers. The REX prefix also affects the FXSAVE and FXRSTOR instructions, in which it selects between two types of 512-byte memory-image format, as described in "Media and x87 Processor State" in Volume 2. The prefix is ignored by all other SSE instructions.

4.8.1.1 Special-Use and Reserved Prefixes

The following prefixes are used as opcode bytes in some SSE instructions and are reserved in all other SSE instructions:

- *Operand-Size Override*—The 66h prefix.
- *REP*—The F2 and F3h prefixes.

4.8.1.2 Prefixes That Cause Exceptions

The following prefixes cause an exception:

- *LOCK*—The F0h prefix causes an invalid-opcode exception when used with SSE instructions.

4.9 Feature Detection

As discussed in Section 4.1.2 "Origins" on page 112, the SSE instruction set is composed of a large number of subsets. To avoid a #UD fault when attempting to execute any of these instructions, hardware must support the instruction subset, system software must indicate its support of SSE context management, and the subset must be enabled. Hardware support for each subset is indicated by a processor feature bit. These are accessed via the CPUID instruction. See Volume 3 for details on the CPUID instruction and the feature bits associated with the SSE Instruction set.

4.10 Exceptions

Types of Exceptions

SSE instructions can generate two types of exceptions:

- *General-Purpose Exceptions*, described below in "General-Purpose Exceptions"
- *SIMD Floating-Point Exception*, described below in "SIMD Floating-Point Exception Causes" on page 220

Relation to x87 Exceptions

Although the SSE instructions and the x87 floating-point instructions each have certain exceptions with the same names, the exception-reporting and exception-handling methods used by the two instruction subsets are distinct and independent of each other. If procedures using both types of instructions are run in the same operating environment, separate services routines should be provided for the exceptions of each type of instruction subset.

4.10.1 General-Purpose Exceptions

The sections below list general-purpose exceptions generated and not generated by SSE instructions. For a summary of the general-purpose exception mechanism, see “Interrupts and Exceptions” on page 91. For details about each exception and its potential causes, see “Exceptions and Interrupts” in Volume 2.

4.10.1.1 Exceptions Generated

The SSE instructions can generate the following general-purpose exceptions:

- #DB—Debug Exception (Vector 1)
- #UD—Invalid-Opcode Exception (Vector 6)
- #NM—Device-Not-Available Exception (Vector 7)
- #DF—Double-Fault Exception (Vector 8)
- #SS—Stack Exception (Vector 12)
- #GP—General-Protection Exception (Vector 13)
- #PF—Page-Fault Exception (Vector 14)
- #MF—x87 Floating-Point Exception-Pending (Vector 16)
- #AC—Alignment-Check Exception (Vector 17)
- #MC—Machine-Check Exception (Vector 18)
- #XF—SIMD Floating-Point Exception (Vector 19)

A device not available exception (#NM) can occur if:

- an attempt is made to execute a SSE instruction when the task switch bit (TS) of the control register (CR0) is set to 1 (CR0.TS = 1), or
- an attempt is made to execute an FXSAVE or FXRSTOR instruction when the floating-point software-emulation (EM) bit in control register 0 is set to 1 (CR0.EM = 1).

An invalid-opcode exception (#UD) can occur if:

- a CPUID feature flag indicates that a feature is not supported (see “Feature Detection” on page 218), or
- a SIMD floating-point exception occurs when the operating-system XMM exception support bit (OSXMMEXCPT) in control register 4 is cleared to 0 (CR4.OSXMMEXCPT = 0).
- an instruction subset is supported but not enabled.

Only the following SSE instructions, all of which can access an MMX register, can cause an #MF exception:

- Data Conversion: CVTPD2PI, CVTPS2PI, CPTPI2PD, CVTPI2PS, CVTTPD2PI, CVTTPS2PI.
- Data Transfer: MOVDQ2Q, MOVQ2DQ.

For details on the system control-register bits, see "System-Control Registers" in Volume 2. For details on the machine-check mechanism, see "Machine Check Mechanism" in Volume 2.

For details on #XF exceptions, see "SIMD Floating-Point Exception Causes" on page 220.

4.10.1.2 Exceptions Not Generated

The SSE instructions do not generate the following general-purpose exceptions:

- #DE—Divide-by-zero-error exception (Vector 0)
- Non-Maskable-Interrupt Exception (Vector 2)
- #BP—Breakpoint Exception (Vector 3)
- #OF—Overflow exception (Vector 4)
- #BR—Bound-range exception (Vector 5)
- Coprocessor-segment-overrun exception (Vector 9)
- #TS—Invalid-TSS exception (Vector 10)
- #NP—Segment-not-present exception (Vector 11)
- #MC—Machine-check exception (Vector 18)

For details on all general-purpose exceptions, see "Exceptions and Interrupts" in Volume 2.

4.10.2 SIMD Floating-Point Exception Causes

The SIMD floating-point exception is the logical OR of the six floating-point exceptions (IE, DE, ZE, OE, UE, PE) that are reported (signalled) in the MXCSR register's exception flags (See Section 4.2.2 "MXCSR Register" on page 115). Each of these six exceptions can be either masked or unmasked by software, using the mask bits in the MXCSR register.

4.10.2.1 Exception Vectors

The SIMD floating-point exception is listed above as #XF (Vector 19) but it actually causes either an #XF exception or a #UD (Vector 6) exception, if an unmasked IE, DE, ZE, OE, UE, or PE exception is reported. The choice of exception vector is determined by the operating-system XMM exception support bit (OSXMMEXCPT) in control register 4 (CR4):

- When CR4.OSXMMEXCPT = 1, a #XF exception occurs.
- When CR4.OSXMMEXCPT = 0, a #UD exception occurs.

SIMD floating-point exceptions are precise. If an exception occurs when it is masked, the processor responds in a default way that does not invoke the SIMD floating-point exception service routine. If an

exception occurs when it is unmasked, the processor suspends processing of the faulting instruction precisely and invokes the exception service routine.

4.10.2.2 Exception Types and Flags

SIMD floating-point exceptions are differentiated into six types, five of which are mandated by the IEEE 754 standard. These six types and their bit-flags in the MXCSR register are shown in Table 4-12. The causes and handling of such exceptions are described below.

Table 4-12. SIMD Floating-Point Exception Flags

Exception and Mnemonic	MXCSR Bit[1]	Comparable IEEE 754 Exception
Invalid-operation exception (IE)	0	Invalid Operation
Denormalized operation exception (DE)	1	*none*
Zero-divide exception (ZE)	2	Division by Zero
Overflow exception (OE)	3	Overflow
Underflow exception (UE)	4	Underflow
Precision exception (PE)	5	Inexact

Note:
1. *See "MXCSR Register" on page 115 for a summary of each exception.*

The sections below describe the causes for the SIMD floating-point exceptions. The pseudocode equations in these descriptions assume logical TRUE = 1 and the following definitions:

Max_{normal}

The largest normalized number that can be represented in the destination format. This is equal to the format's largest representable finite, positive or negative value. (Normal numbers are described in "Normalized Numbers" on page 125.)

Min_{normal}

The smallest normalized number that can be represented in the destination format. This is equal to the format's smallest precisely representable positive or negative value with an unbiased exponent of 1.

$Result_{infinite}$

A result of infinite precision, which is representable when the width of the exponent and the width of the significand are both infinite.

$Result_{round}$

A result, after rounding, whose unbiased exponent is infinitely wide and whose significand is the width specified for the destination format. (Rounding is described in "Floating-Point Rounding" on page 129.)

$Result_{round,\ denormal}$

A result, after rounding and denormalization. (Denormalization is described in "Denormalized (Tiny) Numbers" on page 125.)

Masked and unmasked responses to the exceptions are described in "SIMD Floating-Point Exception Masking" on page 226. The priority of the exceptions is described in "SIMD Floating-Point Exception Priority" on page 224.

4.10.2.3 Invalid-Operation Exception (IE)

The IE exception occurs due to one of the attempted invalid operations shown in Table 4-13.

Table 4-13. Invalid-Operation Exception (IE) Causes

Operation	Condition
Any Arithmetic Operation, and (V)CVTPS2PD, (V)CVTPD2PS, (V)CVTSS2SD, (V)CVTSD2SS	A source operand is an SNaN
(V)MAXPS, (V)MAXPD, (V)MAXSS, (V)MAXSD (V)MINPS, (V)MINPD, (V)MINSS, (V)MINSD (V)CMPPS, (V)CMPPD, (V)CMPSS, (V)CMPSD (V)COMISS, (V)COMISD	A source operand is a NaN (QNaN or SNaN)
(V)ADDPS, (V)ADDPD, (V)ADDSS, (V)ADDSD, (V)ADDSUBPS. (V)ADDSUBPD, (V)HADDPS, (V)HADDPD	Source operands are infinities with opposite signs
(V)SUBPS, (V)SUBPD, (V)SUBSS, (V)SUBSD, (V)ADDSUBPS, (V)ADDSUBPD, (V)HSUBPS, (V)HSUBPD	Source operands are infinities with same sign
(V)MULPS, (V)MULPD, (V)MULSS, (V)MULSD	Source operands are zero and infinity
(V)DIVPS, (V)DIVPD, (V)DIVSS, (V)DIVSD	Source operands are both infinity or both zero
(V)SQRTPS, (V)SQRTPD, (V)SQRTSS, (V)SQRTSD	Source operand is less than zero (except ±0, which returns ±0)
Data conversion from floating-point to integer: CVTPS2PI, CVTPD2PI, (V)CVTSS2SI, (V)CVTSD2SI, (V)CVTPS2DQ, (V)CVTPD2DQ, CVTTPS2PI, CVTTPD2PI, (V)CVTTPD2DQ, (V)CVTTPS2DQ, (V)CVTTSS2SI, (V)CVTTSD2SI)	Source operand is a NaN, infinite, or not representable in destination data type

4.10.2.4 Denormalized-Operand Exception (DE)

The DE exception occurs when one of the source operands of an instruction is in denormalized form, as described in "Denormalized (Tiny) Numbers" on page 125.

4.10.2.5 Zero-Divide Exception (ZE)

The ZE exception occurs when an instruction attempts to divide zero into a non-zero finite dividend.

4.10.2.6 Overflow Exception (OE)

The OE exception occurs when the value of a rounded floating-point result is larger than the largest representable normalized positive or negative floating-point number in the destination format. Specifically:

$$OE = Result_{round} > Max_{normal}$$

An overflow can occur through computation or through conversion of higher-precision numbers to lower-precision numbers.

4.10.2.7 Underflow Exception (UE)

The UE exception occurs when the value of a rounded, non-zero floating-point result is too small to be represented as a normalized positive or negative floating-point number in the destination format. Such a result is called a *tiny* number, associated with the Precision Exception (PE) described immediately below.

If UE exceptions are masked by the underflow mask (UM) bit, a UE exception occurs only if the denormalized form of the rounded result is imprecise. Specifically:

$$UE = ((UM=0 \text{ and } (Result_{round} < Min_{normal}) \text{ or } ((UM=1 \text{ and } (Result_{round,\ denormal}) != Result_{infinite})$$

Underflows can occur, for example, by taking the reciprocal of the largest representable number, or by converting small numbers in double-precision format to a single-precision format, or simply through repeated division. The flush-to-zero (FZ) bit in the MXCSR offers additional control of underflows that are masked. See Section 4.2.2 “MXCSR Register” on page 115 for details.

4.10.2.8 Precision Exception (PE)

The PE exception, also called the *inexact-result* exception, occurs when a rounded floating-point result differs from the infinitely precise result and thus cannot be represented precisely in the destination format. This exception is caused by—among other things—rounding of underflow or overflow results according to the rounding control (RC) field in the MXCSR, as described in “Floating-Point Rounding” on page 129.

If an overflow or underflow occurs and the OE or UE exceptions are masked by the overflow mask (OM) or underflow mask (UM) bit, a PE exception occurs only if the rounded result (for OE) or the denormalized form of the rounded result (for UE) is imprecise. Specifically:

$$PE = ((Result_{round,\ denormal} \text{ or } Result_{round}) != Result_{infinite}) \text{ or } (OM=1 \text{ and } (Result_{round} > Max_{normal})) \text{ or } (UM=1 \text{ and } (Result_{round,\ denormal} < Min_{normal}))$$

Software that does not require exact results normally masks this exception.

4.10.3 SIMD Floating-Point Exception Priority

Figure 4-14 on page 224 shows the priority with which the processor recognizes multiple, simultaneous SIMD floating-point exceptions and operations involving QNaN operands. Each exception type is characterized by its timing, as follows:

- *Pre-Computation*—an exception that is recognized before an instruction begins its operation.
- *Post-Computation*—an exception that is recognized after an instruction completes its operation.

For masked (but not unmasked) post-computation exceptions, a result may be written to the destination, depending on the type of exception. Operations involving QNaNs do not necessarily cause exceptions, but the processor handles them with the priority shown in Table 4-14 relative to the handling of exceptions.

Table 4-14. Priority of SIMD Floating-Point Exceptions

Priority	Exception or Operation	Timing
1	Invalid-operation exception (IE) when accessing SNaN operand	Pre-Computation
2	Operation involving a QNaN operand[1]	—
3	Any other type of invalid-operation exception (IE)	Pre-Computation
	Zero-divide exception (ZE)	Pre-Computation
4	Denormalized operation exception (DE)	Pre-Computation
5	Overflow exception (OE)	Post-Computation
	Underflow exception (UE)	Post-Computation
6	Precision (inexact) exception (PE)	Post-Computation

Note:
1. *Operations involving QNaN operands do not, in themselves, cause exceptions but they are handled with this priority relative to the handling of exceptions.*

Figure 4-51 on page 225 shows the prioritized procedure used by the processor to detect and report SIMD floating-point exceptions. Each of the two types of exceptions—pre-computation and post-computation—is handled independently and completely in the sequence shown. If there are no unmasked exceptions, the processor responds to masked exceptions. Because of this two-step process, up to two exceptions—one pre-computation and one post-computation—can be caused by each operation performed by a single SIMD instruction.

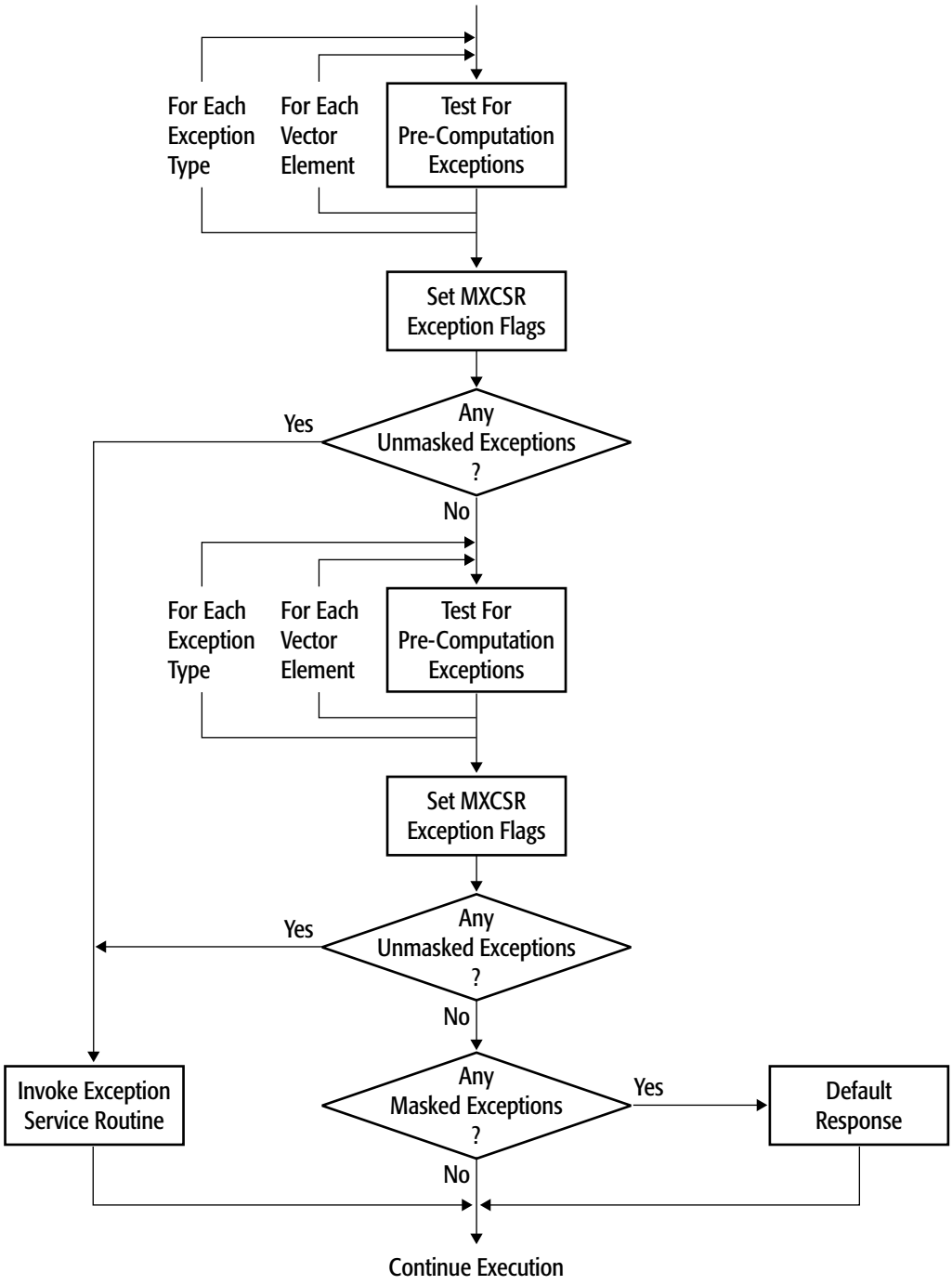


Figure 4-51. SIMD Floating-Point Detection Process

4.10.4 SIMD Floating-Point Exception Masking

The six floating-point exception flags have corresponding exception-flag masks in the MXCSR register, as shown in Table 4-15.

Table 4-15. SIMD Floating-Point Exception Masks

Exception Mask and Mnemonic	MXCSR Bit	Comparable IEEE 754 Exception
Invalid-operation exception mask (IM)	7	Invalid Operation
Denormalized-operand exception mask (DM)	8	*none*
Zero-divide exception mask (ZM)	9	Division by Zero
Overflow exception mask (OM)	10	Overflow
Underflow exception mask (UM)	11	Underflow
Precision exception mask (PM)	12	Inexact

Each mask bit, when set to 1, inhibits invocation of the exception handler for that exception and instead causes a default response. Thus, an unmasked exception is one that invokes its exception handler when it occurs, whereas a masked exception continues normal execution using the default response for the exception type. During power-on initialization, all exception-mask bits in the MXCSR register are set to 1 (masked).

4.10.4.1 Masked Responses

The occurrence of a masked exception does not invoke its exception handler when the exception condition occurs. Instead, the processor handles masked exceptions in a default way, as shown in Table 4-16 on page 227.

Table 4-16. Masked Responses to SIMD Floating-Point Exceptions

Exception	Operation[1]		Processor Response[2]
Invalid-operation exception (IE)	Any of the following, in which one or both operands is an SNaN: • Addition: (V)ADDPS, (V)ADDPD, (V)ADDSS, (V)ADDSD, (V)ADDSUBPD, (V)ADDSUBPS, (V)HADDPS, (V)HADDPD • Subtraction: (V)SUBPS, (V)SUBPD, (V)SUBSS, (V)SUBSD, (V)ADDSUBPD, (V)ADDSUBPS, (V)HSUBPD, (V)HSUBPS • Multiplication: (V)MULPS, (V)MULPD, (V)MULSS, (V)MULSD • Division: (V)DIVPS, (V)DIVPD, (V)DIVSS, (V)DIVSD • Square-root: (V)SQRTPS, (V)SQRTPD, (V)SQRTSS, (V)SQRTSD • Data conversion of floating-point to floating-point: (V)CVTPS2PD, (V)CVTPD2PS, (V)CVTSS2SD, (V)CVTSD2SS		Return a QNaN, based on the rules in Table 4-5 on page 127.
	• Addition of infinities with opposite sign: (V)ADDPS, (V)ADDPD, (V)ADDSS, (V)ADDSD, (V)ADDSUBPS, (V)ADDSUBPD, (V)HADDPD, (V)HADDPS • Subtraction of infinities with same sign: (V)SUBPS, (V)SUBPD, (V)SUBSS, (V)SUBSD, (V)ADDSUBPS, (V)ADDSUBPD, (V)HSUBPS, (V)HSUBPD • Multiplication of zero by infinity: (V)MULPS, (V)MULPD, (V)MULSS, (V)MULSD • Division of zero by zero or infinity by infinity: (V)DIVPS, (V)DIVPD, (V)DIVSS, (V)DIVSD • Square-root in which the operand is non-zero negative: (V)SQRTPS, (V)SQRTPD, (V)SQRTSS, (V)SQRTSD		Return the floating-point indefinite value.
	Any of the following, in which one or both operands is a NaN: • Maximum or Minimum: (V)MAXPS, (V)MAXPD, (V)MAXSS, (V)MAXSD, (V)MINPS, (V)MINPD, (V)MINSS, (V)MINSD		Return second source operand.
	Compare, in which one or both operands is a NaN: (V)CMPPS, (V)CMPPD, (V)CMPSS, (V)CMPSD	Compare is unordered or not-equal	Return mask of all 1s.
		All other compares	Return mask of all 0s.

Note:

1. *For complete details about operations, see “SIMD Floating-Point Exception Causes” on page 220.*
2. *In all cases, the processor sets the associated exception flag in MXCSR. For details about number representation, see “Floating-Point Number Types” on page 124 and “Floating-Point Number Encodings” on page 127.*
3. *This response does not comply with the IEEE 754 standard, but it offers higher performance.*

Table 4-16. Masked Responses to SIMD Floating-Point Exceptions (continued)

Exception	Operation[1]		Processor Response[2]
Invalid-operation exception (IE)	Ordered or unordered scalar compare, in which one or both operands is a NaN ((V)COMISS, (V)COMISD, (V)UCOMISS, (V)UCOMISD).		Sets the result in rFLAGS to "unordered." Clear the overflow (OF), sign (SF), and auxiliary carry (AF) flags in rFLAGS.
	Data conversion from floating-point to integer, in which source operand is a NaN, infinity, or is larger than the representable value of the destination (CVTPS2PI, CVTPD2PI, (V)CVTSS2SI, (V)CVTSD2SI, (V)CVTPS2DQ, (V)CVTPD2DQ, CVTTPS2PI, CVTTPD2PI, (V)CVTTPD2DQ, (V)CVTTPS2DQ, (V)CVTTSS2SI, (V)CVTTSD2SI).		Return the integer indefinite value.
Denormalized-operand exception (DE)	One or both operands is denormal		Return the result using the denormal operand(s).
Zero-divide exception (ZE)	Divide (DIV*x*) zero with non-zero finite dividend		Return signed infinity, with sign bit = XOR of the operand sign bits.
Overflow exception (OE)	Overflow when rounding mode = round to nearest	Sign of result is positive	Return +∞.
		Sign of result is negative	Return –∞.
	Overflow when rounding mode = round toward +∞	Sign of result is positive	Return +∞.
		Sign of result is negative	Return finite negative number with largest magnitude.
	Overflow when rounding mode = round toward -∞	Sign of result is positive	Return finite positive number with largest magnitude.
		Sign of result is negative	Return –∞.
	Overflow when rounding mode = round toward 0	Sign of result is positive	Return finite positive number with largest magnitude.
		Sign of result is negative	Return finite negative number with largest magnitude.

Note:

1. *For complete details about operations, see "SIMD Floating-Point Exception Causes" on page 220.*
2. *In all cases, the processor sets the associated exception flag in MXCSR. For details about number representation, see "Floating-Point Number Types" on page 124 and "Floating-Point Number Encodings" on page 127.*
3. *This response does not comply with the IEEE 754 standard, but it offers higher performance.*

Table 4-16. Masked Responses to SIMD Floating-Point Exceptions (continued)

Exception	Operation[1]		Processor Response[2]
Underflow exception (UE)	Inexact denormalized result	MXCSR flush-to-zero (FZ) bit = 0	Set PE flag and return denormalized result.
		MXCSR flush-to-zero (FZ) bit = 1	Set PE flag and return zero, with sign of true result.[3]
Precision exception (PE)	Inexact normalized or denormalized result	Without OE or UE exception	Return rounded result.
		With masked OE or UE exception	Respond as for OE or UE exception.
		With unmasked OE or UE exception	Respond as for OE or UE exception, and invoke SIMD exception handler.

Note:

1. *For complete details about operations, see "SIMD Floating-Point Exception Causes" on page 220.*
2. *In all cases, the processor sets the associated exception flag in MXCSR. For details about number representation, see "Floating-Point Number Types" on page 124 and "Floating-Point Number Encodings" on page 127.*
3. *This response does not comply with the IEEE 754 standard, but it offers higher performance.*

4.10.4.2 Unmasked Responses

If the processor detects an unmasked exception, it sets the associated exception flag in the MXCSR register and invokes the SIMD floating-point exception handler. The processor does not write a result or change any of the source operands for any type of unmasked exception. The exception handler must determine which exception occurred (by examining the exception flags in the MXCSR register) and take appropriate action.

In all cases of unmasked exceptions, before calling the exception handler, the processor examines the CR4.OSXMMEXCPT bit to see if it is set to 1. If it is set, the processor calls the #XF exception (vector 19). If it is cleared, the processor calls the #UD exception (vector 6). See "System-Control Registers" in Volume 2 for details.

For details about the operations that can cause unmasked exceptions, see "SIMD Floating-Point Exception Causes" on page 220 and Table 4-16 on page 227.

4.10.4.3 Using NaNs in IE Diagnostic Exceptions

Both SNaNs and QNaNs can be encoded with many different values to carry diagnostic information. By means of appropriate masking and unmasking of the invalid-operation exception (IE), software can use signaling NaNs to invoke an exception handler. Within the constraints imposed by the encoding of SNaNs and QNaNs, software may freely assign the bits in the significand of a NaN. See Section "Floating-Point Number Encodings" on page 127 for format details.

For example, software can pre-load each element of an array with a signaling NaN that encodes the array index. When an application accesses an uninitialized array element, the invalid-operation exception is invoked and the service routine can identify that element. A service routine can store

debug information in memory as the exceptions occur. The routine can create a QNaN that references its associated debug area in memory. As the program runs, the service routine can create a different QNaN for each error condition, so that a single test-run can identify a collection of errors.

4.11 Saving, Clearing, and Passing State

4.11.1 Saving and Restoring State

In general, system software should save and restore SSE state between task switches or other interventions in the execution of SSE procedures. Virtually all modern operating systems running on x86 processors implement preemptive multitasking that handle saving and restoring of state across task switches, independent of hardware task-switch support. However, application procedures are also free to save and restore SSE state at any time they deem useful.

Software running at any privilege level may save and restore legacy SSE state by executing the FXSAVE instruction, which saves not only legacy SSE state but also x87 floating-point state. To save and restore the entire SSE context, including the contents of the YMM registers, software must use the XSAVE/XRSTOR instructions (or their optimized variants). These instructions are discussed in Volume 4. Alternatively, software may use multiple move instructions for saving only the contents of selected SSE data registers, or the STMXCSR instruction for saving the MXCSR register state. For details, see “Save and Restore State” on page 183.

4.11.2 Parameter Passing

SSE procedures can use (V)MOV*x* instructions to pass data to other such procedures. This can be done directly, via the YMM/XMM registers, or indirectly by storing data on the procedure stack. When storing to the stack, software should use the rSP register for the memory address and, after the save, explicitly decrement rSP by 16 for each 128-bit XMM register parameter stored on the stack or by 32 for each 256-bit YMM register parameter stored on the stack. Likewise, to load a 128-bit XMM register from the stack, software should increment rSP by 16 after the load or by 32 for each 256-bit YMM register. There is a choice of (V)MOV*x* instructions designed for aligned and unaligned moves, as described in “Data Transfer” on page 150 and “Data Transfer” on page 185.

The processor does not check the data type of instruction operands prior to executing instructions. It only checks them at the point of execution. For example, if the processor executes an arithmetic instruction that takes double-precision operands but is provided with single-precision operands by MOV*x* instructions, the processor will first convert the operands from single precision to double precision prior to executing the arithmetic operation, and the result will be correct. However, the required conversion may cause degradation of performance.

Because of this possibility of data-type mismatching between (V)MOV*x* instructions used to pass parameters and the instructions in the called procedure that subsequently operate on the moved data, the calling procedure should save its own state prior to the call. The called procedure cannot determine the caller’s data types, and thus it cannot optimize its choice of instructions for storing a caller’s state.

For further information, see the software optimization documentation for particular hardware implementations.

4.11.3 Accessing Operands in MMX™ Registers

Software may freely mix SSE instructions (integer or floating-point) with 64-bit media instructions (integer or floating-point) and general-purpose instructions in a single procedure. There are no restrictions on transitioning from SSE procedures to x87 procedures, except when a SSE procedure accesses an MMX register by means of a data-transfer or data-conversion instruction.

In such cases, software should separate such procedures or dynamic link libraries (DLLs) from x87 floating-point procedures or DLLs by clearing the MMX state with the EMMS instruction, as described in Section 5.6.2 “Exit Media State” on page 255. For further details, see Section 5.13 “Mixing Media Code with x87 Code” on page 280.

4.12 Performance Considerations

In addition to typical code optimization techniques, such as those affecting loops and the inlining of function calls, the following considerations may help improve the performance of application programs written with SSE instructions.

These are implementation-independent performance considerations. Other considerations depend on the hardware implementation. For information about such implementation-dependent considerations and for more information about application performance in general, see the data sheets and the software-optimization guides relating to particular hardware implementations.

4.12.1 Use Small Operand Sizes

The performance advantages available with SSE operations is to some extent a function of the data sizes operated upon. The smaller the data size, the more data elements that can be packed into a single vector. The parallelism of computation increases as the number of elements per vector increases.

4.12.2 Reorganize Data for Parallel Operations

Much of the performance benefit from the SSE instructions comes from the parallelism inherent in vector operations. It can be advantageous to reorganize data before performing arithmetic operations so that its layout after reorganization maximizes the parallelism of the arithmetic operations.

The speed of memory access is particularly important for certain types of computation, such as graphics rendering, that depend on the regularity and locality of data-memory accesses. For example, in matrix operations, performance is high when operating on the rows of the matrix, because row bytes are contiguous in memory, but lower when operating on the columns of the matrix, because column bytes are not contiguous in memory and accessing them can result in cache misses. To improve performance for operations on such columns, the matrix should first be transposed. Such transpositions can, for example, be done using a sequence of unpacking or shuffle instructions.

4.12.3 Remove Branches

Branch can be replaced with SSE instructions that simulate predicated execution or conditional moves, as described in “Branch Removal” on page 147. The branch can be replaced with SSE instructions that

simulate predicated execution or conditional moves. Figure 4-21 on page 148 shows an example of a non-branching sequence that implements a two-way multiplexer.

Where possible, break long dependency chains into several shorter dependency chains that can be executed in parallel. This is especially important for floating-point instructions because of their longer latencies.

4.12.4 Use Streaming Loads and Stores

The (V)MOVNTDQ, (V)MOVNTDQA and (V)MASKMOVDQU instructions load or store streaming (non-temporal) data from or to memory. These instructions indicate to the processor that the data they reference will be used only once and is therefore not subject to cache-related overhead (such as write-allocation). A typical case benefitting from streaming stores occurs when data written by the processor is never read by the processor, such as data written to a graphics frame buffer.

CPU read accesses of WC memory type regions normally have significantly lower throughput than accesses to cacheable memory. However, the (V)MOVNTDQA instruction provides a non-temporal hint that can cause adjacent 16-byte items within an aligned 64-byte region of WC memory type (a streaming line) to be fetched and held in a small set of temporary buffers (streaming load buffers). Subsequent streaming loads to other aligned 16-byte items in the same streaming line may be supplied from the streaming load buffer and can improve throughput.

The following programming practices can improve efficiency of (V)MOVNTDQA streaming loads from WC memory:

- Streaming loads must be 16-byte aligned.
- Group streaming loads of the same streaming cache line for effective use of the small number of streaming load buffers. If loads to the same streaming line are excessively spaced apart, it may cause the streaming line to be re-fetched from memory.
- Group streaming loads from at most a few streaming lines together. The number of streaming load buffers is small; grouping a modest number of streams will avoid running out of streaming load buffers and the resultant re-fetching of streaming lines from memory.
- Avoid writing to a streaming line until all 16-byte-aligned reads from the streaming line have occurred. Reading a 16-byte item from a streaming line that has been written, may cause the streaming line to be re-fetched.
- Avoid reading a given 16-byte item within a streaming line more than once; repeated loads of a particular 16-byte item are likely to cause the streaming line to be re-fetched.
- Streaming load buffers, reflecting the WC memory type characteristics, are not required to be snooped by operations from other agents. Software should not rely upon such coherency actions to provide any data coherency with respect to other logical processors or bus agents. Rather, software must insure the consistency of WC memory accesses between producers and consumers.
- Streaming loads may be weakly ordered and may appear to software to execute out of order with respect to other memory operations. Software must explicitly use fences (e.g. MFENCE) if it

needs to preserve order among streaming loads or between streaming loads and other memory operations.

- Streaming loads must not be used to reference memory addresses that are mapped to I/O devices having side effects or when reads to these devices are destructive. This is because MOVNTDQA is speculative in nature.

The following two code examples demonstrate the basic assembly sequences that depict the principles of using MOVNTDQA with a producer-consumer pair accessing a WC memory region.

Example 1: Using MOVNTDQA with a Consumer and PCI Producer

```
// P0: producer is a PCI device writing into the WC space
# the PCI device updates status through a UC flag, "u_dev_status"
# the protocol for "u_dev_status" : 0: produce; 1: consume; 2: all done
  mov eax, $0
  mov [u_dev_status], eax
producerStart:
  mov eax, [u_dev_status] # poll status flag to see if consumer is requesting data
  cmp eax, $0
  jne done # no longer need to produce commence PCI writes to WC region
  mov eax, $1 # producer ready to notify the consumer via status flag
  mov [u_dev_status], eax
# now wait for consumer to signal its status
spinloop:
  cmp [u_dev_status], $1 # was signal received from consumer
  jne producerStart # yes
  jmp spinloop # check again
done:
// producer is finished at this point

// P1: consumer check PCI status flag to consume WC data
  mov eax, $0 # request to the producer
  mov [u_dev_status], eax
consumerStart:
  mov; eax, [u_dev_status] # reads the value of the PCI status
  cmp eax, $1            # has producer written
  jne consumerStart      # tight loop; make it more efficient with pause, etc.
  mfence # producer finished device writes to WC, ensure WC region is coherent
ntread:
  movntdqa xmm0, [addr]
  movntdqa xmm1, [addr + 16]
  movntdqa xmm2, [addr + 32]
  movntdqa xmm3, [addr + 48]
  …          # do more NT reads as needed
  mfence     # ensure PCI device reads the correct value of [u_dev_status]
# now decide whether done or need the producer to produce more data
# if done write a 2 into the variable, otherwise write a 0 into the variable
  mov eax, $0/$2 # end or continue producing
  mov [u_dev_status], eax
# to consume again jump back to consumerStart after storing a 0 into eax
# otherwise, done
```

Example 2: Using MOVNTDQA with Producer-Consumer Threads

```
// P0: producer writes into the WC space
# xchg is an implicitly locked operation
producerStart:
# use a locked operation to prevent races between producer and consumer
# updating this variable. Assume initial value is 0
  mov eax, $0
  xchg eax, [signalVariable] # signalVariable is used for communicating
  cmp eax, $0 # am I supposed to be writing for the consumer
  jne done # I no longer need to produce
  movntdq [addr1], xmm0 # producer writes the data
  movntdq [addr2], xmm1 # ...
# Again use a locked instruction. Serves 2 purposes. Updated value signals
# to consumer and serialization of the lock flushes all WC stores to memory
  mov eax, $1
  xchg [signalVariable], eax # signal to the consumer
# a more efficient spin loop can be done using PAUSE
spinloop:
  cmp [signalVariable], $1 # did I get signal from consumer
  jne producerStart      # yes
  jmp spinloop           # check again
done:
// producer is finished at this point

// P1: consumer reads from write combining space
  mov eax, $0
consumerStart:
  lock; xadd [signalVariable], eax # reads the value of the signal variable in
  cmp eax, $1      # has producer written to signal its state
  jne consumerStart # simple loop; replace with PAUSE to make it more efficient
# read data from WC memory space with MOVNTDQA to achieve higher throughput
ntread: # keep reads from same cache line as close together as possible
  movntdqa xmm0, [addr]
  movntdqa xmm1, [addr + 16]
  movntdqa xmm2, [addr + 32]
  movntdqa xmm3, [addr + 48]
# since a lock prevents younger MOVNTDQA from passing it, the
# above non temporal loads will happen only after producer has signaled

… # do more NT reads as needed

# now decide whether done or need producer to produce more data
# if done, write a 2 into the variable, otherwise write a 0 into the variable
  mov eax, $0/$2 # end or continue producing
  xchg [signalVariable], eax
# to consume again, jump back to consumerStart after storing a 0 into eax
# otherwise, done
```

4.12.5 Align Data

Data alignment is particularly important for performance when data written by one instruction is read by a subsequent instruction soon after the write, or when accessing streaming (non-temporal) data. These cases may occur frequently in 256-bit and 128-bit media procedures.

Accesses to data stored at unaligned locations may benefit from on-the-fly software alignment or from repetition of data at different alignment boundaries, as required by different loops that process the data.

4.12.6 Organize Data for Cacheability

Pack small data structures into cache-line-size blocks. Organize frequently accessed constants and coefficients into cache-line-size blocks and prefetch them. Procedures that access data arranged in memory-bus-sized blocks, or memory-burst-sized blocks, can make optimum use of the available memory bandwidth.

For data that will be used only once in a procedure, consider using non-cacheable memory. Accesses to such memory are not burdened by the overhead of cache protocols.

4.12.7 Prefetch Data

Media applications typically operate on large data sets. Because of this, they make intensive use of the memory bus. Memory latency can be substantially reduced—especially for data that will be used only once—by prefetching such data into various levels of the cache hierarchy. Software can use the PREFETCH*x* instructions very effectively in such cases, as described in “Cache and Memory Management” on page 71.

Some of the best places to use prefetch instructions are inside loops that process large amounts of data. If the loop goes through less than one cache line of data per iteration, partially unroll the loop. Try to use virtually all of the prefetched data. This usually requires unit-stride memory accesses—those in which all accesses are to contiguous memory locations. Exactly one PREFETCH*x* instruction per cache line must be used. For further details, see the *Optimization Guide for AMD Athlon™ 64 and AMD Opteron™ Processors*, order# 25112.

4.12.8 Use SSE Code for Moving Data

Movements of data between memory, GPR, XMM, and MMX registers can take advantage of the parallel vector operations supported by the SSE MOV*x* instructions. Figure 4-13 on page 142 illustrates the range of move operations available.

4.12.9 Retain Intermediate Results in SSE Registers

Keep intermediate results in the SSE (YMM/XMM) registers as much as possible, especially if the intermediate results are used shortly after they have been produced. Avoid spilling intermediate results to memory and reusing them shortly thereafter. Take advantage of the increased number of addressable SSE registers available in 64-bit mode.

4.12.10 Replace GPR Code with SSE Code.

In 64-bit mode, the AMD64 architecture provides twice the number of general-purpose registers (GPRs) as the legacy x86 architecture, thereby reducing potential pressure on GPRs. Nevertheless, general-purpose instructions do not operate in parallel on vectors of elements, as do SSE instructions. Thus, SSE code supports parallel operations and can perform better with algorithms and data that are organized for parallel operations.

4.12.11 Replace x87 Code with SSE Code

One of the most useful advantages of SSE instructions is the ability to intermix integer and floating-point instructions in the same procedure, using a register set that is separate from the GPR, MMX, and x87 register sets. Code written with SSE floating-point instructions can operate in parallel on eight times as many single-precision floating-point operands as can x87 floating-point code. This achieves potentially eight times the computational work of x87 instructions that take single-precision operands. Also, the higher density of SSE floating-point operands may make it possible to remove local temporary variables that would otherwise be needed in x87 floating-point code. SSE code is also easier to write than x87 floating-point code, because the SSE register file is flat, rather than stack-oriented, and in 64-bit mode there are twice the number of SSE registers as x87 registers. Moreover, when integer and floating-point instructions must be used together, SSE floating-point instructions avoid the potential need to save and restore state between integer operations and floating-point procedures.

5 64-Bit Media Programming

This chapter describes the 64-bit media programming model. This model includes all instructions that access the MMX™ registers, including the MMX and 3DNow!™ instructions. Subsequent extensions, part of the Streaming SIMD Extensions (SSE), added new instructions that also utilize MMX registers.

The 64-bit media instructions perform integer and floating-point operations primarily on vector operands (a few of the instructions take scalar operands). The MMX integer operations produce signed, unsigned, and/or saturating results. The 3DNow! floating-point operations take single-precision operands and produce saturating results without generating floating-point exceptions. The instructions that take vector operands can speed up certain types of procedures by significant factors, depending on data-element size and the regularity and locality of data accesses to memory.

The term *64-bit* is used in two different contexts within the AMD64 architecture: the 64-bit media instructions, described in this chapter, and the 64-bit operating mode, described in “64-Bit Mode” on page 6.

5.1 Origins

The 64-bit media instructions were introduced in the following extensions to the legacy x86 architecture:

- *MMX*. The original MMX programming model defined eight 64-bit MMX registers and a number of vector instructions that operate on packed integers held in the MMX registers or sourced from memory. This subset was subsequently extended.
- *3DNow!*. Added vector floating-point instructions, most of which take vector operands in MMX registers or memory locations. This instruction set was subsequently extended.
- *SSE.* The original Steaming SIMD Extensions (SSE1) and the subsequent SSE2 added instructions that perform conversions between operands in the 64-bit MMX registers and other registers.

For details on the extension-set origin of each instruction, see “Instruction Subsets vs. CPUID Feature Sets” in Volume 3.

5.2 Compatibility

64-bit media instructions can be executed in any of the architecture’s operating modes. Existing MMX and 3DNow! binary programs run in legacy and compatibility modes without modification. The support provided by the AMD64 architecture for such binaries is identical to that provided by legacy x86 architectures.

To run in 64-bit mode, 64-bit media programs must be recompiled. The recompilation has no side effects on such programs, other than to make available the extended general-purpose registers and 64-bit virtual address space.

The MMX and 3DNow! instructions introduce no additional registers, status bits, or other processor state to the legacy x86 architecture. Instead, they use the x87 floating-point registers that have long been a part of most x86 architectures. Because of this, 64-bit media procedures require no special operating-system support or exception handlers. When state-saves are required between procedures, the same instructions that system software uses to save and restore x87 floating-point state also save and restore the 64-bit media-programming state.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. Relevant recommendations are provided below and in the *AMD64 Programmer's Manual Volume 4: 64-Bit Media and x87 Floating-Point Instructions*.

5.3 Capabilities

The 64-bit media instructions are designed to support multimedia and communication applications that operate on vectors of small-sized data elements. For example, 8-bit and 16-bit integer data elements are commonly used for pixel information in graphics applications, and 16-bit integer data elements are used for audio sampling. The 64-bit media instructions allow multiple data elements like these to be packed into single 64-bit vector operands located in an MMX register or in memory. The instructions operate in parallel on each of the elements in these vectors. For example, 8-bit integer data can be packed in vectors of eight elements in a single 64-bit register, so that a single instruction can operated on all eight byte elements simultaneously.

Typical applications of the 64-bit media integer instructions include music synthesis, speech synthesis, speech recognition, audio and video compression (encoding) and decompression (decoding), 2D and 3D graphics (including 3D texture mapping), and streaming video. Typical applications of the 64-bit media floating-point instructions include digital signal processing (DSP) kernels and front-end 3D graphics algorithms, such as geometry, clipping, and lighting.

These types of applications are referred to as *media* applications. Such applications commonly use small data elements in repetitive loops, in which the typical operations are inherently parallel. In 256-color video applications, for example, 8-bit operands in 64-bit MMX registers can be used to compute transformations on eight pixels per instruction.

5.3.1 Parallel Operations

Most of the 64-bit media instructions perform parallel operations on vectors of operands. *Vector* operations are also called *packed* or *SIMD* (single-instruction, multiple-data) operations. They take operands consisting of multiple elements and operate on all elements in parallel. Figure 5-1 on page 241 shows an example of an integer operation on two vectors, each containing 16-bit (word) elements. There are also 64-bit media instructions that operate on vectors of byte or doubleword elements.

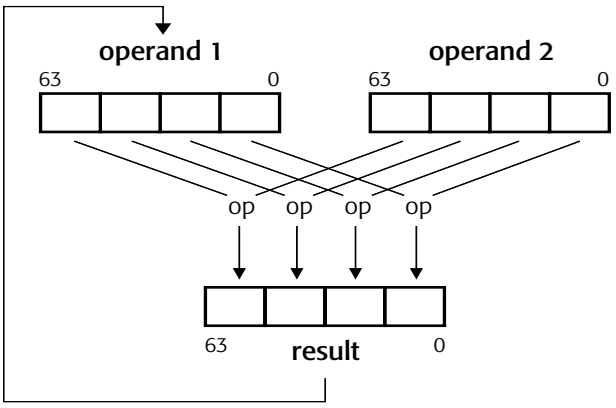


Figure 5-1. Parallel Integer Operations on Elements of Vectors

5.3.2 Data Conversion and Reordering

The 64-bit media instructions support conversions of various integer data types to floating-point data types, and vice versa.

There are also instructions that reorder vector-element ordering or the bit-width of vector elements. For example, the unpack instructions take two vector operands and interleave their low or high elements. Figure 5-2 on page 242 shows an unpack operation (PUNPCKLWD) that interleaves low-order elements of each source operand. If each element of operand 2 has the value zero, the operation zero-extends each element of operand 1 to twice its original width. This may be useful, for example, prior to an arithmetic operation in which the data-conversion result must be paired with another source operand containing vector elements that are twice the width of the pre-conversion (half-size) elements. There are also pack instructions that convert each element of 2x size in a pair of vectors to elements of 1x size, with saturation at maximum and minimum values.

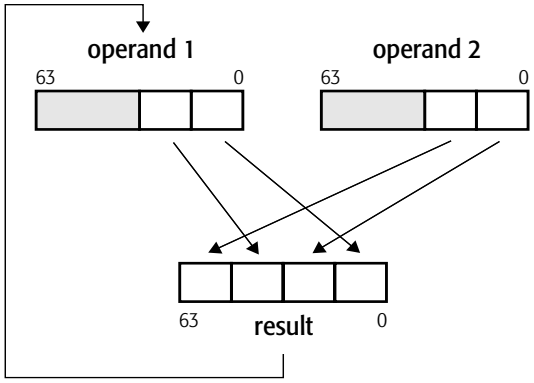


Figure 5-2. Unpack and Interleave Operation

Figure 5-3 shows a shuffle operation (PSHUFW), in which one of the operands provides vector data, and an immediate byte provides shuffle control for up to 256 permutations of the data.

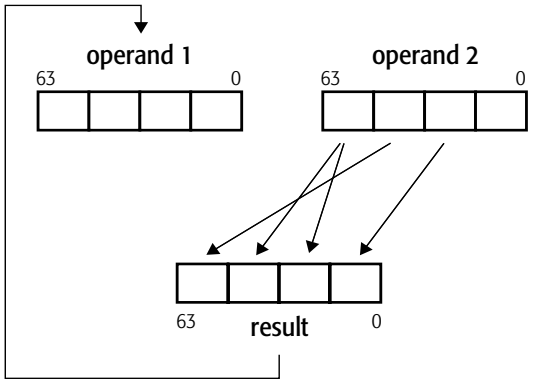


Figure 5-3. Shuffle Operation (1 of 256)

5.3.3 Matrix Operations

Media applications often multiply and accumulate vector and matrix data. In 3D graphics applications, for example, objects are typically represented by triangles, each of whose vertices are located in 3D space by a matrix of coordinate values, and matrix transforms are performed to simulate object movement.

The 64-bit media integer and floating-point instructions can perform several types of matrix-vector or matrix-matrix operations, such as addition, subtraction, multiplication, and accumulation. The integer

instructions can also perform multiply-accumulate operations. Efficient matrix multiplication is further supported with instructions that can first transpose the elements of matrix rows and columns. These transpositions can make subsequent accesses to memory or cache more efficient when performing arithmetic matrix operations.

Figure 5-4 shows a vector multiply-add instruction (PMADDWD) that multiplies vectors of 16-bit integer elements to yield intermediate results of 32-bit elements, which are then summed pair-wise to yield two 32-bit elements.

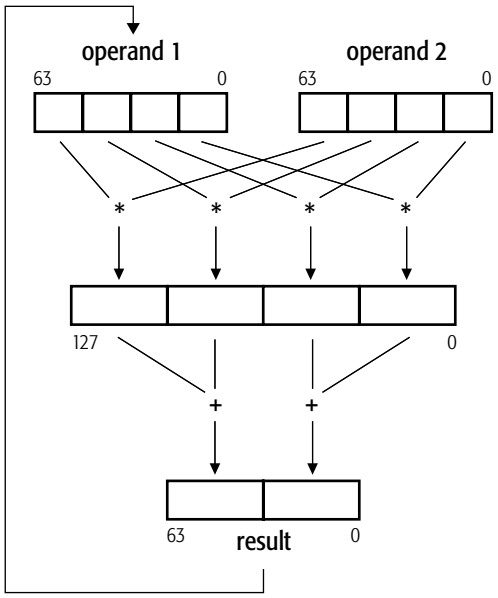


Figure 5-4. Multiply-Add Operation

The operation shown in Figure 5-4 can be used together with transpose and vector-add operations (see "Addition" on page 262) to accumulate *dot product* results (also called *inner* or *scalar products*), which are used in many media algorithms.

5.3.4 Saturation

Several of the 64-bit media integer instructions and most of the 64-bit media floating-point instructions produce vector results in which each element *saturates* independently of the other elements in the result vector. Such results are clamped (limited) to the maximum or minimum value representable by the destination data type when the true result exceeds that maximum or minimum representable value.

Saturation avoids the need for code that tests for potential overflow or underflow. Saturating data is useful for representing physical-world data, such as sound and color. It is used, for example, when combining values for pixel coloring.

5.3.5 Branch Removal

Branching is a time-consuming operation that, unlike most 64-bit media vector operations, does not exhibit parallel behavior (there is only one branch target, not multiple targets, per branch instruction). In many media applications, a branch involves selecting between only a few (often only two) cases. Such branches can be replaced with 64-bit media vector compare and vector logical instructions that simulate predicated execution or conditional moves.

Figure 5-5 shows an example of a non-branching sequence that implements a two-way multiplexer—one that is equivalent to the ternary operator "?:" in C and C++. The comparable code sequence is explained in "Compare and Write Mask" on page 267.

The sequence in Figure 5-5 begins with a vector compare instruction that compares the elements of two source operands in parallel and produces a mask vector containing elements of all 1s or 0s. This mask vector is ANDed with one source operand and ANDed-Not with the other source operand to isolate the desired elements of both operands. These results are then ORed to select the relevant elements from each operand. A similar branch-removal operation can be done using floating-point source operands.

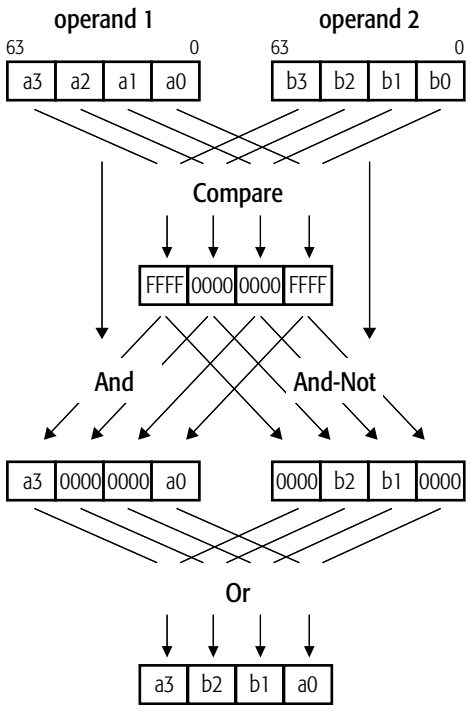


Figure 5-5. Branch-Removal Sequence

5.3.6 Floating-Point (3DNow!™) Vector Operations

Floating-point vector instructions using the MMX registers were introduced by AMD with the 3DNow! technology. These instructions take 64-bit vector operands consisting of two 32-bit single-precision floating-point numbers, shown as *FP single* in Figure 5-6.

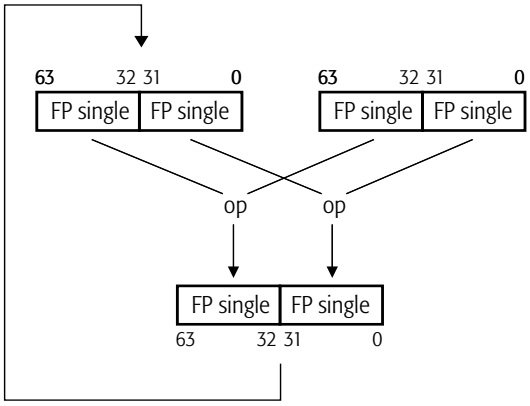


Figure 5-6. Floating-Point (3DNow!™ Instruction) Operations

The AMD64 architecture's 3DNow! floating-point instructions provide a unique advantage over legacy x87 floating-point instructions: They allow integer *and* floating-point instructions to be intermixed in the same procedure, using only the MMX registers. This avoids the need to switch between integer MMX procedures and x87 floating-point procedures—a switch that may involve time-consuming state saves and restores—while at the same time leaving the YMM/XMM register resources free for other applications.

The 3DNow! instructions allow applications such as 3D graphics to accelerate front-end geometry, clipping, and lighting calculations. Picture and pixel data are typically integer data types, although both integer and floating-point instructions are often required to operate completely on the data. For example, software can change the viewing perspective of a 3D scene through transformation matrices by using floating-point instructions in the same procedure that contains integer operations on other aspects of the graphics data.

3DNow! programs typically perform better than x87 floating-point code, because the MMX register file is flat rather than stack-oriented and because 3DNow! instructions can operate on twice as many operands as x87 floating-point instructions. This ability to operate in parallel on twice as many floating-point values in the same register space often makes it possible to remove local temporary variables in 3DNow! code that would otherwise be needed in x87 floating-point code.

5.4 Registers

5.4.1 MMX™ Registers

Eight 64-bit MMX registers, *mmx0–mmx7,* support the 64-bit media instructions. Figure 5-7 shows these registers. They can hold operands for both vector and scalar operations on integer (MMX) and floating-point (3DNow!) data types.

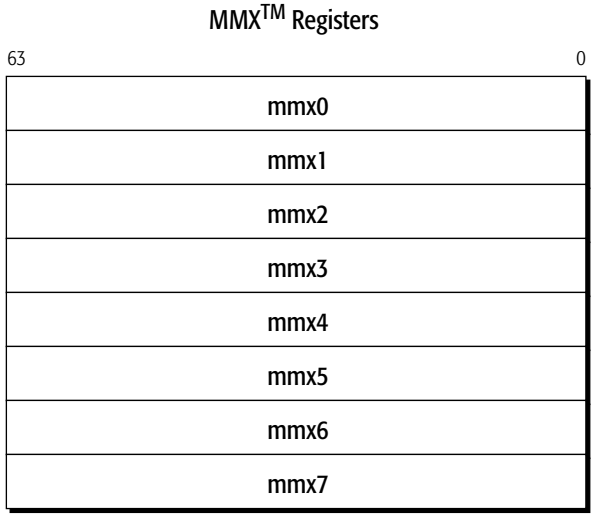


Figure 5-7. 64-Bit Media Registers

The MMX registers are mapped onto the low 64 bits of the 80-bit x87 floating-point physical data registers, FPR0–FPR7, described in Section 6.2. “Registers” on page 286. However, the x87 stack register structure, ST(0)–ST(7), is not used by MMX instructions. The x87 tag bits, top-of-stack pointer (TOP), and high bits of the 80-bit FPR registers are changed when 64-bit media instructions are executed. For details about the x87-related actions performed by hardware during execution of 64-bit media instructions, see “Actions Taken on Executing 64-Bit Media Instructions” on page 278.

5.4.2 Other Registers

Some 64-bit media instructions that perform data transfer, data conversion or data reordering operations (“Data Transfer” on page 256, “Data Conversion” on page 257, and “Data Conversion” on page 271) can access operands in the general-purpose registers (GPRs) or XMM registers. When addressing GPRs or YMM/XMM registers in 64-bit mode, the REX instruction prefix can be used to access the extended GPRs or YMM/XMM registers, as described in “REX Prefixes” on page 79. For a description of the GPR registers, see “Registers” on page 23. For a description of the YMM/XMM registers, see Section 4.2.1. “SSE Registers” on page 113.

5.5 Operands

Operands for a 64-bit media instruction are either referenced by the instruction's opcode or included as an immediate value in the instruction encoding. Depending on the instruction, referenced operands can be located in registers or memory. The data types of these operands include vector and scalar integer, and vector floating-point.

5.5.1 Data Types

Figure 5-8 on page 248 shows the register images of the 64-bit media data types. These data types can be interpreted by instruction syntax and/or the software context as one of the following types of values:

- Vector (packed) single-precision (32-bit) floating-point numbers.
- Vector (packed) signed (two's-complement) integers.
- Vector (packed) unsigned integers.
- Scalar signed (two's-complement) integers.
- Scalar unsigned integers.

Hardware does not check or enforce the data types for instructions. Software is responsible for ensuring that each operand for an instruction is of the correct data type. Software can interpret the data types in ways other than those shown in Figure 5-8 on page 248—such as bit fields or fractional numbers—but the 64-bit media instructions do not directly support such interpretations and software must handle them entirely on its own.

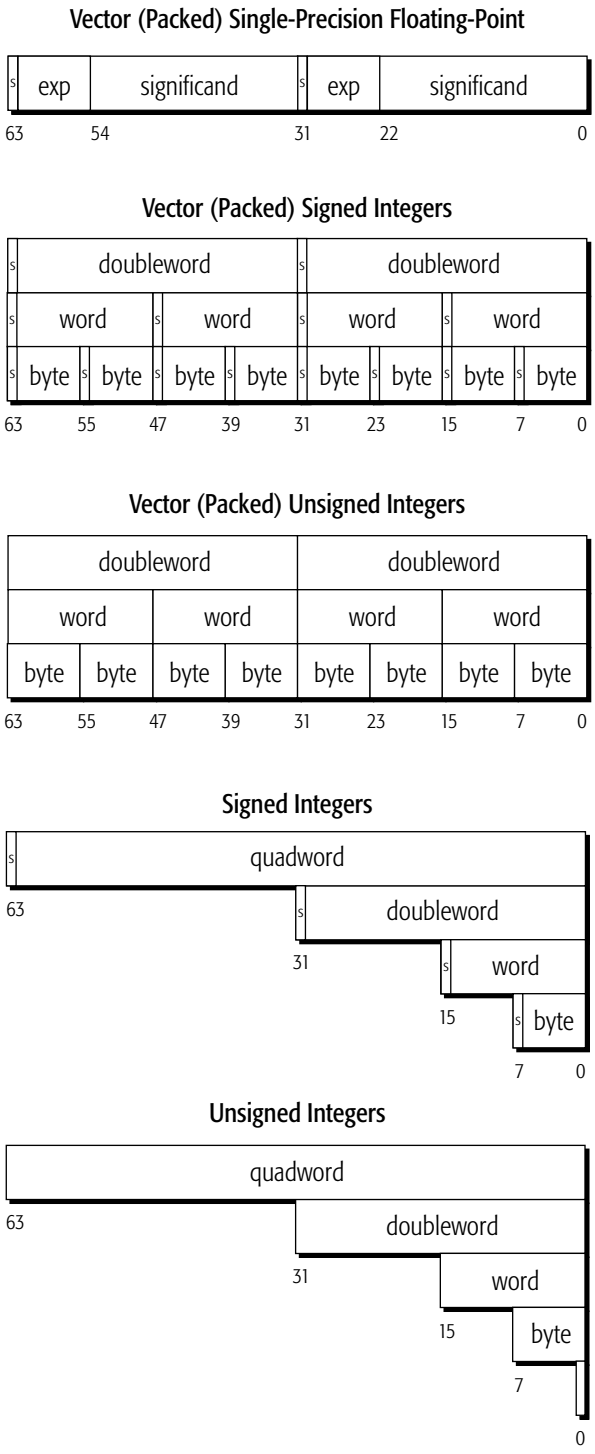


Figure 5-8. 64-Bit Media Data Types

5.5.2 Operand Sizes and Overrides

Operand sizes for 64-bit media instructions are determined by instruction opcodes. Some of these opcodes include an operand-size override prefix, but this prefix acts in a special way to modify the opcode and is considered an integral part of the opcode. The general use of the 66h operand-size override prefix described in “Instruction Prefixes” on page 76 does not apply to 64-bit media instructions.

For details on the use of operand-size override prefixes in 64-bit media instructions, see the opcodes in “64-Bit Media Instruction Reference” in Volume 5.

5.5.3 Operand Addressing

Depending on the 64-bit media instruction, referenced operands may be in registers or memory.

5.5.3.1 Register Operands

Most 64-bit media instructions can access source and destination operands located in MMX registers. A few of these instructions access the XMM or GPR registers. When addressing GPR or XMM registers in 64-bit mode, the REX instruction prefix can be used to access the extended GPR or XMM registers, as described in “Instruction Prefixes” on page 275.

The 64-bit media instructions do not access the rFLAGS register, and none of the bits in that register are affected by execution of the 64-bit media instructions.

5.5.3.2 Memory Operands

Most 64-bit media instructions can read memory for source operands, and a few of the instructions can write results to memory. “Memory Addressing” on page 14, describes the general methods and conditions for addressing memory operands.

5.5.3.3 Immediate Operands

Immediate operands are used in certain data-conversion and vector-shift instructions. Such instructions take 8-bit immediates, which provide control for the operation.

5.5.3.4 I/O Ports

I/O ports in the I/O address space cannot be directly addressed by 64-bit media instructions, and although memory-mapped I/O ports can be addressed by such instructions, doing so may produce unpredictable results, depending on the hardware implementation of the architecture. See the data sheet or software-optimization documentation for particular hardware implementations.

5.5.4 Data Alignment

Those 64-bit media instructions that access a 128-bit operand in memory incur a general-protection exception (#GP) if the operand is not aligned to a 16-byte boundary. These instructions include:

- CVTPD2PI—Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers.

- CVTTPD2PI—Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers, Truncated.
- FXRSTOR—Restore XMM, MMX, and x87 State.
- FXSAVE—Save XMM, MMX, and x87 State.

For other 64-bit media instructions, the architecture does not impose data-alignment requirements for accessing 64-bit media data in memory. Specifically, operands in physical memory do not need to be stored at addresses that are even multiples of the operand size in bytes. However, the consequence of storing operands at unaligned locations is that accesses to those operands may require more processor and bus cycles than for aligned accesses. See "Data Alignment" on page 43 for details.

5.5.5 Integer Data Types

Most of the MMX instructions support operations on the integer data types shown in Figure 5-8 on page 248. These instructions are summarized in "Instruction Summary—Integer Instructions" on page 253. The characteristics of these data types are described below.

5.5.5.1 Sign

Many of the 64-bit media instructions have variants for operating on signed or unsigned integers. For signed integers, the sign bit is the most-significant bit—bit 7 for a byte, bit 15 for a word, bit 31 for a doubleword, or bit 63 for a quadword. Arithmetic instructions that are not specifically named as unsigned perform signed two's-complement arithmetic.

5.5.5.2 Maximum and Minimum Representable Values

Table 5-1 shows the range of representable values for the integer data types.

Table 5-1. Range of Values in 64-Bit Media Integer Data Types

Data-Type Interpretation		Byte	Word	Doubleword	Quadword
Unsigned integers	Base-2 (exact)	0 to $+2^8-1$	0 to $+2^{16}-1$	0 to $+2^{32}-1$	0 to $+2^{64}-1$
	Base-10 (approx.)	0 to 255	0 to 65,535	0 to $4.29 * 10^9$	0 to $1.84 * 10^{19}$
Signed integers[1]	Base-2 (exact)	-2^7 to $+(2^7-1)$	-2^{15} to $+(2^{15}-1)$	-2^{31} to $+(2^{31}-1)$	-2^{63} to $+(2^{63}-1)$
	Base-10 (approx.)	–128 to +127	–32,768 to +32,767	$-2.14 * 10^9$ to $+2.14 * 10^9$	$-9.22 * 10^{18}$ to $+9.22 * 10^{18}$

5.5.5.3 Saturation

Saturating (also called limiting or clamping) instructions limit the value of a result to the maximum or minimum value representable by the destination data type. Saturating versions of integer vector-arithmetic instructions operate on byte-sized and word-sized elements. These instructions—for example, PADDSx, PADDUSx, PSUBSx, and PSUBUSx—saturate signed or unsigned data independently for each element in a vector when the element reaches its maximum or minimum representable value. Saturation avoids overflow or underflow errors.

The examples in Table 5-2 on page 251 illustrate saturating and non-saturating results with word operands. Saturation for other data-type sizes follows similar rules. Once saturated, the saturated value is treated like any other value of its type. For example, if 0001h is subtracted from the saturated value, 7FFFh, the result is 7FFEh.

Table 5-2. Saturation Examples

Operation	Non-Saturated Infinitely Precise Result	Saturated Signed Result	Saturated Unsigned Result
7000h + 2000h	9000h	7FFFh	9000h
7000h + 7000h	E000h	7FFFh	E000h
F000h + F000h	1E000h	E000h	FFFFh
9000h + 9000h	12000h	8000h	FFFFh
7FFFh + 0100h	80FFh	7FFFh	80FFh
7FFFh + FF00h	17EFFh	7EFFh	FFFFh

Arithmetic instructions not specifically designated as saturating perform non-saturating, two's-complement arithmetic.

5.5.5.4 Rounding

There is a rounding version of the integer vector-multiply instruction, PMULHRW, that multiplies pairs of signed-integer word elements and then adds 8000h to the lower word of the doubleword result, thus rounding the high-order word which is returned as the result.

5.5.5.5 Other Fixed-Point Operands

The architecture provides specific support only for integer fixed-point operands—those in which an implied binary point is located to the right of bit 0. Nevertheless, software may use fixed-point operands in which the implied binary point is located in any position. In such cases, software is responsible for managing the interpretation of such implied binary points, as well as any redundant sign bits that may occur during multiplication.

5.5.6 Floating-Point Data Types

All 64-bit media 3DNow! instructions, except PFRCP and PFRSQRT, take 64-bit vector operands. They operate in parallel on two single-precision (32-bit) floating-point values contained in those vectors.

Figure 5-9 shows the format of the vector operands. The characteristics of the single-precision floating-point data types are described below. The 64-bit floating-point media instructions are summarized in "Instruction Summary—Floating-Point Instructions" on page 270.

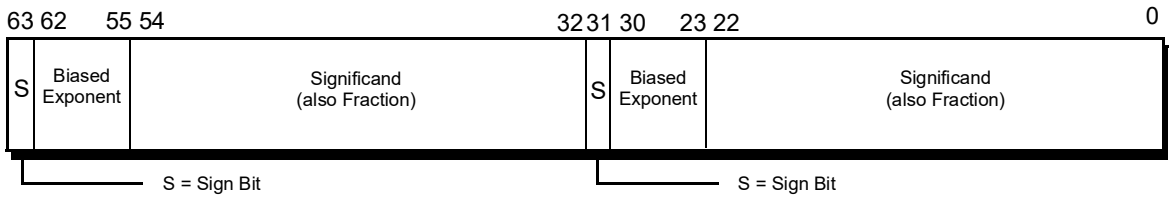


Figure 5-9. 64-Bit Floating-Point (3DNow!™) Vector Operand

5.5.6.1 Single-Precision Format

The single-precision floating-point format supported by 64-bit media instructions is the same format as the normalized IEEE 754 single-precision format. This format includes a sign bit, an 8-bit biased exponent, and a 23-bit significand with one hidden integer bit for a total of 24 bits in the significand. The hidden integer bit is assumed to have a value of 1, and the significand field is also the fraction. The bias of the exponent is 127. However, the 3DNow! format does not support other aspects of the IEEE 754 standard, such as multiple rounding modes, representation of numbers other than normalized numbers, and floating-point exceptions.

5.5.6.2 Range of Representable Values and Saturation

Table 5-3 shows the range of representable values for 64-bit media floating-point data. Table 5-4 shows the exponent ranges. The largest representable positive normal number has an exponent of FEh and a significand of 7FFFFFh, with a numerical value of $2^{127} * (2 - 2^{-23})$. The smallest representable negative normal number has an exponent of 01h and a significand of 000000h, with a numerical value of 2^{-126}.

Table 5-3. Range of Values in 64-Bit Media Floating-Point Data Types

Data-Type Interpretation		Doubleword	Quadword
Floating-point	Base-2 (exact)	2^{-126} to $2^{127} * (2 - 2^{-23})$	Two single-precision floating-point doublewords
	Base-10 (approx.)	$1.17 * 10^{-38}$ to $+3.40 * 10^{38}$	

Table 5-4. 64-Bit Floating-Point Exponent Ranges

Biased Exponent	Description
FFh	Unsupported[1]
00h	Zero

Note:
1. *Unsupported numbers can be used as source operands but produce undefined results.*

Table 5-4. 64-Bit Floating-Point Exponent Ranges (continued)

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

Note:
1. *Unsupported numbers can be used as source operands but produce undefined results.*

Results that, after rounding, overflow above the maximum-representable positive or negative number are saturated (limited or clamped) at the maximum positive or negative number. Results that underflow below the minimum-representable positive or negative number are treated as zero.

5.5.6.3 Floating-Point Rounding

In contrast to the IEEE standard, which requires four rounding modes, the 64-bit media floating-point instructions support only one rounding mode, depending on the instruction. All such instructions use round-to-nearest, except certain floating-point-to-integer conversion instructions (“Data Conversion” on page 271) which use round-to-zero.

5.5.6.4 No Support for Infinities, NaNs, and Denormals

64-bit media floating-point instructions support only normalized numbers. They do not support infinity, NaN, and denormalized number representations. Operations on such numbers produce undefined results, and no exceptions are generated. If all source operands are normalized numbers, these instructions never produce infinities, NaNs, or denormalized numbers as results.

This aspect of 64-bit media floating-point operations does not comply with the IEEE 754 standard. Software must use only normalized operands and ensure that computations remain within valid normalized-number ranges.

5.5.6.5 No Support for Floating-Point Exceptions

The 64-bit media floating-point instructions do not generate floating-point exceptions. Software must ensure that in-range operands are provided to these instructions.

5.6 Instruction Summary—Integer Instructions

This section summarizes the functions of the integer (MMX and a few SSE and SSE2) instructions in the 64-bit media instruction subset. These include integer instructions that use an MMX register for source or destination and data-conversion instructions that convert from integers to floating-point formats. For a summary of the floating-point instructions in the 64-bit media instruction subset, including data-conversion instructions that convert from floating-point to integer formats, see “Instruction Summary—Floating-Point Instructions” on page 270.

The instructions are organized here by functional group—such as data-transfer, vector arithmetic, and so on. Software running at any privilege level can use any of these instructions, if the CPUID instruction reports support for the instructions (see “Feature Detection” on page 276). More detail on individual instructions is given in the alphabetically organized “64-Bit Media Instruction Reference” in Volume 5.

5.6.1 Syntax

Each instruction has a *mnemonic syntax* used by assemblers to specify the operation and the operands to be used for source and destination (result) data. The majority of 64-bit media integer instructions have the following syntax:

```
MNEMONIC mmx1, mmx2/mem64
```

Figure 5-10 on page 254 shows an example of the mnemonic syntax for a packed add bytes (PADDB) instruction.

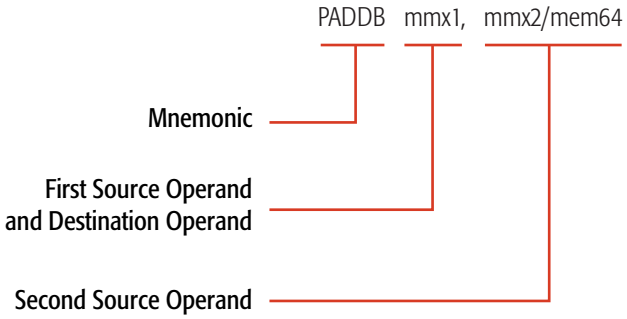


Figure 5-10. Mnemonic Syntax for Typical Instruction

This example shows the PADDB mnemonic followed by two operands, a 64-bit MMX register operand and another 64-bit MMX register or 64-bit memory operand. In most instructions that take two operands, the first (left-most) operand is both a source operand and the destination operand. The second (right-most) operand serves only as a source. Some instructions can have one or more prefixes that modify default properties, as described in “Instruction Prefixes” on page 275.

5.6.1.1 Mnemonics

The following characters are used as prefixes in the mnemonics of integer instructions:

- **CVT**—Convert
- **CVTT**—Convert with truncation
- **P**—Packed (vector)
- **PACK**—Pack elements of 2x data size to 1x data size
- **PUNPCK**—Unpack and interleave elements

In addition to the above prefix characters, the following characters are used elsewhere in the mnemonics of integer instructions:

- **B**—Byte
- **D**—Doubleword
- **DQ**—Double quadword
- **ID**—Integer doubleword
- **IW**—Integer word
- **PD**—Packed double-precision floating-point
- **PI**—Packed integer
- **PS**—Packed single-precision floating-point
- **Q**—Quadword
- **S**—Signed
- **SS**—Signed saturation
- **U**—Unsigned
- **US**—Unsigned saturation
- **W**—Word
- ***x***—One or more variable characters in the mnemonic

For example, the mnemonic for the instruction that packs four words into eight unsigned bytes is PACKUSWB. In this mnemonic, the PACK designates 2x-to-1x conversion of vector elements, the US designates unsigned results with saturation, and the WB designates vector elements of the source as words and those of the result as bytes.

5.6.2 Exit Media State

The exit media state instructions are used to isolate the use of processor resources between 64-bit media instructions and x87 floating-point instructions.

- EMMS—Exit Media State
- FEMMS—Fast Exit Media State

These instructions initialize the contents of the x87 floating-point stack registers—called *clearing the MMX state*. Software should execute one of these instructions before leaving a 64-bit media procedure.

The EMMS and FEMMS instructions both clear the MMX state, as described in “Mixing Media Code with x87 Code” on page 280. The instructions differ in one respect: FEMMS leaves the data in the x87 stack registers undefined. By contrast, EMMS leaves the data in each such register as it was defined by the last x87 or 64-bit media instruction that wrote to the register. The FEMMS instruction is supported for backward-compatibility. Software that must be compatible with both AMD and non-AMD processors should use the EMMS instruction.

5.6.3 Data Transfer

The data-transfer instructions copy operands between a 32-bit or 64-bit memory location, an MMX register, an XMM register, or a GPR. The MOV mnemonic, which stands for *move*, is a misnomer. A copy function is actually performed instead of a move.

Move

- MOVD—Move Doubleword
- MOVQ—Move Quadword
- MOVDQ2Q—Move Double Quadword to Quadword
- MOVQ2DQ—Move Quadword to Double Quadword

The MOVD instruction copies a 32-bit or 64-bit value from a general-purpose register (GPR) or memory location to an MMX register, or from an MMX register to a GPR or memory location. If the source operand is 32 bits and the destination operand is 64 bits, the source is zero-extended to 64 bits in the destination. If the source is 64 bits and the destination is 32 bits, only the low-order 32 bits of the source are copied to the destination.

The MOVQ instruction copies a 64-bit value from an MMX register or 64-bit memory location to another MMX register, or from an MMX register to another MMX register or 64-bit memory location.

The MOVDQ2Q instruction copies the low-order 64-bit value in an XMM register to an MMX register.

The MOVQ2DQ instruction copies a 64-bit value from an MMX register to the low-order 64 bits of an XMM register, with zero-extension to 128 bits.

The MOVD and MOVQ instructions—along with the PUNPCK*x* instructions—are often among the most frequently used instructions in 64-bit media procedures (both integer and floating-point). The move instructions are similar to the assignment operator in high-level languages.

5.6.3.1 Move Non-Temporal

The move non-temporal instructions are called *streaming-store* instructions. They minimize pollution of the cache. The assumption is that the data they reference will be used only once, and is therefore not subject to cache-related overhead such as write-allocation. For further information, see “Memory Optimization” on page 98.

- MOVNTQ—Move Non-temporal Quadword
- MASKMOVQ—Mask Move Quadword

The MOVNTQ instruction stores a 64-bit MMX register value into a 64-bit memory location. The MASKMOVQ instruction stores bytes from the first operand, as selected by the mask value (most-significant bit of each byte) in the second operand, to a memory location specified in the rDI and DS registers. The first operand is an MMX register, and the second operand is another MMX register. The

size of the store is determined by the effective address size. Figure 5-11 on page 257 shows the MASKMOVQ operation.

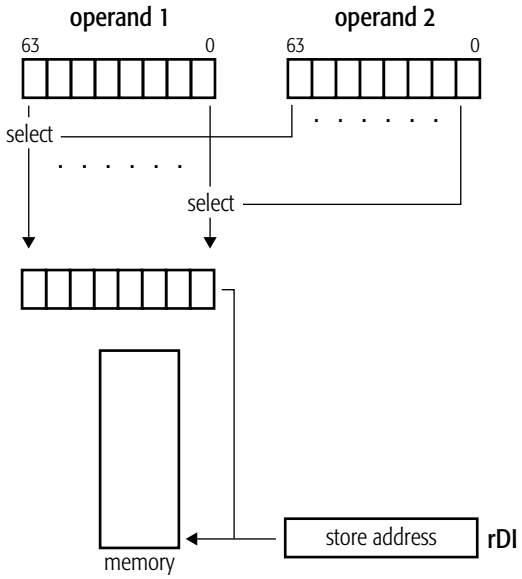


Figure 5-11. MASKMOVQ Move Mask Operation

The MOVNTQ and MASKMOVQ instructions use weakly-ordered, write-combining buffering of write data and they minimize cache pollution. The exact method by which cache pollution is minimized depends on the hardware implementation of the instruction. For further information, see “Memory Optimization” on page 98.

A typical case benefitting from streaming stores occurs when data written by the processor is never read by the processor, such as data written to a graphics frame buffer. MASKMOVQ is useful for the handling of end cases in block copies and block fills based on streaming stores.

Move Mask

- PMOVMSKB—Packed Move Mask Byte

The PMOVMSKB instruction moves the most-significant bit of each byte in an MMX register to the low-order byte of a 32-bit or 64-bit general-purpose register, with zero-extension. It is useful for extracting bits from a mask, or extracting zero-point values from quantized data such as signal samples, resulting in a byte that can be used for data-dependent branching.

5.6.4 Data Conversion

The integer data-conversion instructions convert operands from integer formats to floating-point formats. They take 64-bit integer source operands. For data-conversion instructions that take 32-bit and 64-bit floating-point source operands, see “Data Conversion” on page 271. For data-conversion

instructions that take 128-bit source operands, see “Data Conversion” on page 155 and “Data Conversion” on page 190.

5.6.4.1 Convert Integer to Floating-Point

These instructions convert integer data types into floating-point data types.

- CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point
- CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point
- PI2FW—Packed Integer To Floating-Point Word Conversion
- PI2FD—Packed Integer to Floating-Point Doubleword Conversion

The CVTPI2Px instructions convert two 32-bit signed integer values in the second operand (an MMX register or 64-bit memory location) to two single-precision (CVTPI2PS) or double-precision (CVTPI2PD) floating-point values. The instructions then write the converted values into the low-order 64 bits of an XMM register (CVTPI2PS) or the full 128 bits of an XMM register (CVTPI2PD). The CVTPI2PS instruction does not modify the high-order 64 bits of the XMM register.

The PI2Fx instructions are 3DNow! instructions. They convert two 16-bit (PI2FW) or 32-bit (PI2FD) signed integer values in the second operand to two single-precision floating-point values. The instructions then write the converted values into the destination. If a PI2FD conversion produces an inexact value, the value is truncated (rounded toward zero).

5.6.5 Data Reordering

The integer data-reordering instructions pack, unpack, interleave, extract, insert, shuffle, and swap the elements of vector operands.

5.6.5.1 Pack with Saturation

These instructions pack 2x-sized data types into 1x-sized data types, thus halving the precision of each element in a vector operand.

- PACKSSDW—Pack with Saturation Signed Doubleword to Word
- PACKSSWB—Pack with Saturation Signed Word to Byte
- PACKUSWB—Pack with Saturation Signed Word to Unsigned Byte

The PACKSSDW instruction converts each 32-bit signed integer in its two source operands (an MMX register or 64-bit memory location and another MMX register) into a 16-bit signed integer and packs the converted values into the destination MMX register. The PACKSSWB instruction does the analogous operation between word elements in the source vectors and byte elements in the destination vector. The PACKUSWB instruction does the same as PACKSSWB except that it converts word integers into unsigned (rather than signed) bytes.

Figure 5-12 on page 259 shows an example of a PACKSSDW instruction. The operation merges vector elements of 2x size (doubleword-size) into vector elements of 1x size (word-size), thus reducing the precision of the vector-element data types. Any results that would otherwise overflow or

underflow are saturated (clamped) at the maximum or minimum representable value, respectively, as described in “Saturation” on page 250.

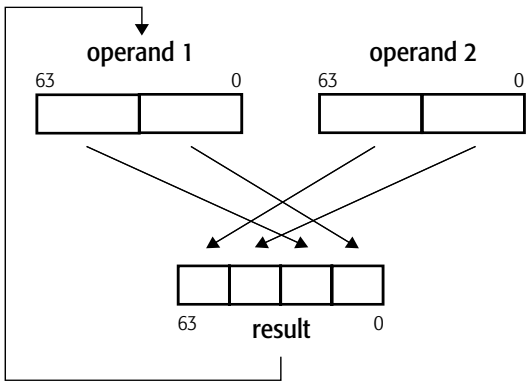


Figure 5-12. PACKSSDW Pack Operation

Conversion from higher-to-lower precision may be needed, for example, after an arithmetic operation which requires the higher-precision format to prevent possible overflow, but which requires the lower-precision format for a subsequent operation.

5.6.5.2 Unpack and Interleave

These instructions interleave vector elements from the high or low half of two source operands. They can be used to double the precision of operands.

- PUNPCKHBW—Unpack and Interleave High Bytes
- PUNPCKHWD—Unpack and Interleave High Words
- PUNPCKHDQ—Unpack and Interleave High Doublewords
- PUNPCKLBW—Unpack and Interleave Low Bytes
- PUNPCKLWD—Unpack and Interleave Low Words
- PUNPCKLDQ—Unpack and Interleave Low Doublewords

The PUNPCKHBW instruction unpacks the four high-order bytes from its two source operands and interleaves them into the bytes in the destination operand. The bytes in the low-order half of the source operand are ignored. The PUNPCKHWD and PUNPCKHDQ instructions perform analogous operations for words and doublewords in the source operands, packing them into interleaved words and interleaved doublewords in the destination operand.

The PUNPCKLBW, PUNPCKLWD, and PUNPCKLDQ instructions are analogous to their high-element counterparts except that they take elements from the low doubleword of each source vector

and ignore elements in the high doubleword. If the source operand for PUNPCKL*x* instructions is in memory, only the low 32 bits of the operand are loaded.

Figure 5-13 on page 260 shows an example of the PUNPCKLWD instruction. The elements are taken from the low half of the source operands. In this register image, elements from *operand2* are placed to the left of elements from *operand1*.

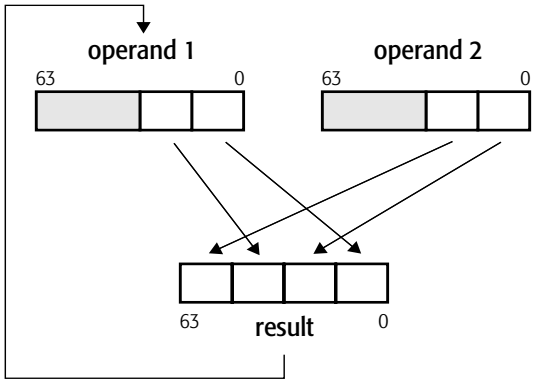


Figure 5-13. PUNPCKLWD Unpack and Interleave Operation

If one of the two source operands is a vector consisting of all zero-valued elements, the unpack instructions perform the function of expanding vector elements of 1x size into vector elements of 2x size (for example, word-size to doubleword-size). If both source operands are of identical value, the unpack instructions can perform the function of duplicating adjacent elements in a vector.

The PUNPCK*x* instructions—along with MOVD and MOVQ—are among the most frequently used instructions in 64-bit media procedures (both integer and floating-point).

5.6.5.3 Extract and Insert

These instructions copy a word element from a vector, in a manner specified by an immediate operand.

- PEXTRW—Packed Extract Word
- PINSRW—Packed Insert Word

The PEXTRW instruction extracts a 16-bit value from an MMX register, as selected by the immediate-byte operand, and writes it to the low-order word of a 32-bit or 64-bit general-purpose register, with zero-extension to 32 or 64 bits. PEXTRW is useful for loading computed values, such as table-lookup indices, into general-purpose registers where the values can be used for addressing tables in memory.

The PINSRW instruction inserts a 16-bit value from a the low-order word of a 32-bit or 64-bit general purpose register or a 16-bit memory location into an MMX register. The location in the destination

register is selected by the immediate-byte operand. The other words in the destination register operand are not modified.

5.6.5.4 Shuffle and Swap

These instructions reorder the elements of a vector.

- PSHUFW—Packed Shuffle Words
- PSWAPD—Packed Swap Doubleword

The PSHUFW instruction moves any one of the four words in its second operand (an MMX register or 64-bit memory location) to specified word locations in its first operand (another MMX register). The ordering of the shuffle can occur in any of 256 possible ways, as specified by the immediate-byte operand. Figure 5-14 shows one of the 256 possible shuffle operations. PSHUFW is useful, for example, in color imaging when computing alpha saturation of RGB values. In this case, PSHUFW can replicate an alpha value in a register so that parallel comparisons with three RGB values can be performed.

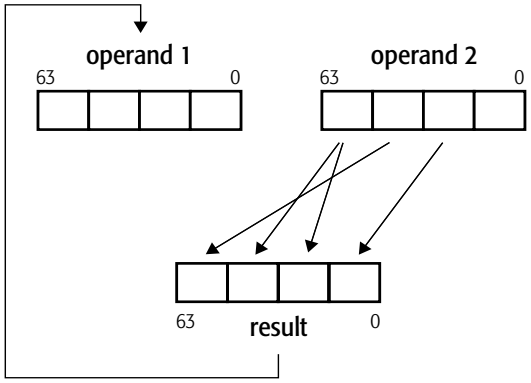


Figure 5-14. PSHUFW Shuffle Operation

The PSWAPD instruction swaps (reverses) the order of two 32-bit values in the second operand and writes each swapped value in the corresponding doubleword of the destination. Figure 5-15 shows a swap operation. PSWAPD is useful, for example, in complex-number multiplication in which the elements of one source operand must be swapped (see “Accumulation” on page 272 for details). PSWAPD supports independent source and result operands so that it can also perform a load function.

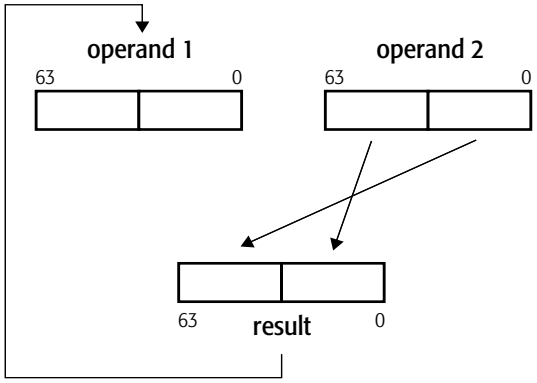


Figure 5-15. PSWAPD Swap Operation

5.6.6 Arithmetic

The integer vector-arithmetic instructions perform an arithmetic operation on the elements of two source vectors. Arithmetic instructions that are not specifically named as unsigned perform signed two's-complement arithmetic.

Addition

- PADDB—Packed Add Bytes
- PADDW—Packed Add Words
- PADDD—Packed Add Doublewords
- PADDQ—Packed Add Quadwords
- PADDSB—Packed Add with Saturation Bytes
- PADDSW—Packed Add with Saturation Words
- PADDUSB—Packed Add Unsigned with Saturation Bytes
- PADDUSW—Packed Add Unsigned with Saturation Words

The PADDB, PADDW, PADDD, and PADDQ instructions add each 8-bit (PADDB), 16-bit (PADDW), 32-bit (PADDD), or 64-bit (PADDQ) integer element in the second operand to the corresponding, same-sized integer element in the first operand. The instructions then write the integer result of each addition to the corresponding, same-sized element of the destination. These instructions operate on both signed and unsigned integers. However, if the result overflows, only the low-order byte, word, doubleword, or quadword of each result is written to the destination. The PADDD instruction can be used together with PMADDWD (page 264) to implement dot products.

The PADDSB and PADDSW instructions perform additions analogous to the PADDB and PADDW instructions, except with saturation. For each result in the destination, if the result is larger than the

largest, or smaller than the smallest, representable 8-bit (PADDSB) or 16-bit (PADDSW) signed integer, the result is saturated to the largest or smallest representable value, respectively.

The PADDUSB and PADDUSW instructions perform saturating additions analogous to the PADDSB and PADDSW instructions, except on unsigned integer elements.

Subtraction

- PSUBB—Packed Subtract Bytes
- PSUBW—Packed Subtract Words
- PSUBD—Packed Subtract Doublewords
- PSUBQ—Packed Subtract Quadword
- PSUBSB—Packed Subtract with Saturation Bytes
- PSUBSW—Packed Subtract with Saturation Words
- PSUBUSB—Packed Subtract Unsigned and Saturate Bytes
- PSUBUSW—Packed Subtract Unsigned and Saturate Words

The subtraction instructions perform operations analogous to the addition instructions.

The PSUBB, PSUBW, PSUBD, and PSUBQ instructions subtract each 8-bit (PSUBB), 16-bit (PSUBW), 32-bit (PSUBD), or 64-bit (PSUBQ) integer element in the second operand from the corresponding, same-sized integer element in the first operand. The instructions then write the integer result of each subtraction to the corresponding, same-sized element of the destination. These instructions operate on both signed and unsigned integers. However, if the result underflows, only the low-order byte, word, doubleword, or quadword of each result is written to the destination.

The PSUBSB and PSUBSW instructions perform subtractions analogous to the PSUBB and PSUBW instructions, except with saturation. For each result in the destination, if the result is larger than the largest, or smaller than the smallest, representable 8-bit (PSUBSB) or 16-bit (PSUBSW) signed integer, the result is saturated to the largest or smallest representable value, respectively.

The PSUBUSB and PSUBUSW instructions perform saturating subtractions analogous to the PSUBSB and PSUBSW instructions, except on unsigned integer elements.

Multiplication

- PMULHW—Packed Multiply High Signed Word
- PMULLW—Packed Multiply Low Signed Word
- PMULHRW—Packed Multiply High Rounded Word
- PMULHUW—Packed Multiply High Unsigned Word
- PMULUDQ—Packed Multiply Unsigned Doubleword and Store Quadword

The PMULHW instruction multiplies each 16-bit signed integer value in first operand by the corresponding 16-bit integer in the second operand, producing a 32-bit intermediate result. The instruction then writes the high-order 16 bits of the 32-bit intermediate result of each multiplication to

the corresponding word of the destination. The PMULLW instruction performs the same multiplication as PMULHW but writes the low-order 16 bits of the 32-bit intermediate result to the corresponding word of the destination.

The PMULHRW instruction performs the same multiplication as PMULHW but with rounding. After the multiplication, PMULHRW adds 8000h to the lower word of the doubleword result, thus rounding the high-order word which is returned as the result.

The PMULHUW instruction performs the same multiplication as PMULHW but on unsigned operands. The instruction is useful in 3D rasterization, which operates on unsigned pixel values.

The PMULUDQ instruction, unlike the other PMUL*x* instructions, preserves the full precision of the result. It multiplies 32-bit unsigned integer values in the first and second operands and writes the full 64-bit result to the destination.

See “Shift” on page 266 for shift instructions that can be used to perform multiplication and division by powers of 2.

Multiply-Add

- PMADDWD—Packed Multiply Words and Add Doublewords

The PMADDWD instruction multiplies each 16-bit signed value in the first operand by the corresponding 16-bit signed value in the second operand. The instruction then adds the adjacent 32-bit intermediate results of each multiplication, and writes the 32-bit result of each addition into the corresponding doubleword of the destination. PMADDWD thus performs two signed (16 × 16 = 32) + (16 × 16 = 32) multiply-adds in parallel. Figure 5-16 shows the PMADDWD operation.

The only case in which overflow can occur is when all four of the 16-bit source operands used to produce a 32-bit multiply-add result have the value 8000h. In this case, the result returned is 8000_0000h, because the maximum negative 16-bit value of 8000h multiplied by itself equals 4000_0000h, and 4000_0000h added to 4000_0000h equals 8000_0000h. The result of multiplying two negative numbers should be a positive number, but 8000_0000h is the maximum possible 32-bit negative number rather than a positive number.

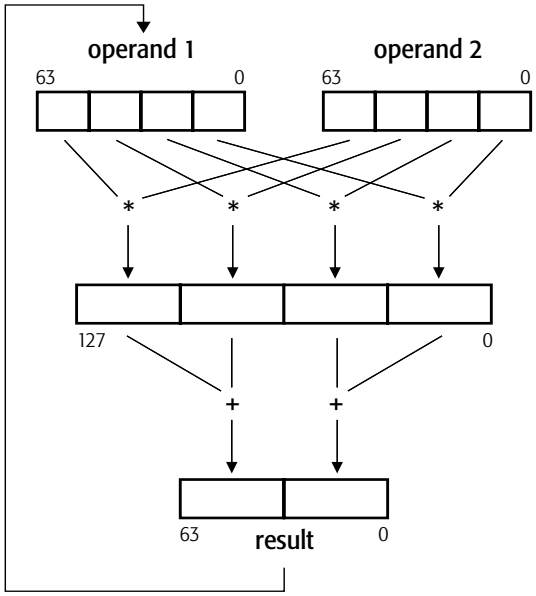


Figure 5-16. PMADDWD Multiply-Add Operation

PMADDWD can be used with one source operand (for example, a coefficient) taken from memory and the other source operand (for example, the data to be multiplied by that coefficient) taken from an MMX register. The instruction can also be used together with the PADDD instruction (page 262) to compute dot products, such as those required for finite impulse response (FIR) filters, one of the commonly used DSP algorithms. Scaling can be done, before or after the multiply, using a vector-shift instruction (page 266).

For floating-point multiplication operations, see the PFMUL instruction on page 272. For floating-point accumulation operations, see the PFACC, PFNACC, and PFPNACC instructions on page 272.

Average

- PAVGB—Packed Average Unsigned Bytes
- PAVGW—Packed Average Unsigned Words
- PAVGUSB—Packed Average Unsigned Packed Bytes

The PAVGx instructions compute the rounded average of each unsigned 8-bit (PAVGB) or 16-bit (PAVGW) integer value in the first operand and the corresponding, same-sized unsigned integer in the second operand. The instructions then write each average in the corresponding, same-sized element of the destination. The rounded average is computed by adding each pair of operands, adding 1 to the temporary sum, and then right-shifting the temporary sum by one bit.

The PAVGB instruction is useful for MPEG decoding, in which motion compensation performs many byte-averaging operations between and within macroblocks. In addition to speeding up these operations, PAVGB can free up registers and make it possible to unroll the averaging loops.

The PAVGUSB instruction (a 3DNow! instruction) performs a function identical to the PAVGB instruction, described on page 265, although the two instructions have different opcodes.

Sum of Absolute Differences

- PSADBW—Packed Sum of Absolute Differences of Bytes into a Word

The PSADBW instruction computes the absolute values of the differences of corresponding 8-bit signed integer values in the first and second operands. The instruction then sums the differences and writes an unsigned 16-bit integer result in the low-order word of the destination. The remaining bytes in the destination are cleared to all 0s.

Sums of absolute differences are used to compute the L1 norm in motion-estimation algorithms for video compression.

5.6.7 Shift

The vector-shift instructions are useful for scaling vector elements to higher or lower precision, packing and unpacking vector elements, and multiplying and dividing vector elements by powers of 2.

Left Logical Shift

- PSLLW—Packed Shift Left Logical Words
- PSLLD—Packed Shift Left Logical Doublewords
- PSLLQ—Packed Shift Left Logical Quadwords

The PSLLx instructions left-shift each of the 16-bit (PSLLW), 32-bit (PSLLD), or 64-bit (PSLLQ) values in the first operand by the number of bits specified in the second operand. The instructions then write each shifted value into the corresponding, same-sized element of the destination. The first and second operands are either an MMX register and another MMX register or 64-bit memory location, or an MMX register and an immediate-byte value. The low-order bits that are emptied by the shift operation are cleared to 0.

In integer arithmetic, left logical shifts effectively multiply unsigned operands by positive powers of 2.

Right Logical Shift

- PSRLW—Packed Shift Right Logical Words
- PSRLD—Packed Shift Right Logical Doublewords
- PSRLQ—Packed Shift Right Logical Quadwords

The PSRLx instructions right-shift each of the 16-bit (PSRLW), 32-bit (PSRLD), or 64-bit (PSRLQ) values in the first operand by the number of bits specified in the second operand. The instructions then write each shifted value into the corresponding, same-sized element of the destination. The first and

second operands are either an MMX register and another MMX register or 64-bit memory location, or an MMX register and an immediate-byte value. The high-order bits that are emptied by the shift operation are cleared to 0. In integer arithmetic, right logical shifts effectively divide unsigned operands or positive signed operands by positive powers of 2.

PSRLQ can be used to move the high 32 bits of an MMX register to the low 32 bits of the register.

Right Arithmetic Shift

- PSRAW—Packed Shift Right Arithmetic Words
- PSRAD—Packed Shift Right Arithmetic Doublewords

The PSRAx instructions right-shifts each of the 16-bit (PSRAW) or 32-bit (PSRAD) values in the first operand by the number of bits specified in the second operand. The instructions then write each shifted value into the corresponding, same-sized element of the destination. The high-order bits that are emptied by the shift operation are filled with the sign bit of the initial value.

In integer arithmetic, right arithmetic shifts effectively divide signed operands by positive powers of 2.

5.6.8 Compare

The integer vector-compare instructions compare two operands, and they either write a mask or they write the maximum or minimum value.

Compare and Write Mask

- PCMPEQB—Packed Compare Equal Bytes
- PCMPEQW—Packed Compare Equal Words
- PCMPEQD—Packed Compare Equal Doublewords
- PCMPGTB—Packed Compare Greater Than Signed Bytes
- PCMPGTW—Packed Compare Greater Than Signed Words
- PCMPGTD—Packed Compare Greater Than Signed Doublewords

The PCMPEQx and PCMPGTx instructions compare corresponding bytes, words, or doubleword in the first and second operands. The instructions then write a mask of all 1s or 0s for each compare into the corresponding, same-sized element of the destination.

For the PCMPEQx instructions, if the compared values are equal, the result mask is all 1s. If the values are not equal, the result mask is all 0s. For the PCMPGTx instructions, if the signed value in the first operand is greater than the signed value in the second operand, the result mask is all 1s. If the value in the first operand is less than or equal to the value in the second operand, the result mask is all 0s. PCMPEQx can be used to set the bits in an MMX register to all 1s by specifying the same register for both operands.

By specifying the same register for both operands, PCMPEQx can be used to set the bits in an MMX register to all 1s.

Figure 5-5 on page 244 shows an example of a non-branching sequence that implements a two-way multiplexer—one that is equivalent to the following sequence of ternary operators in C or C++:

```
r0 = a0 > b0 ? a0 : b0
r1 = a1 > b1 ? a1 : b1
r2 = a2 > b2 ? a2 : b2
r3 = a3 > b3 ? a3 : b3
```

Assuming `mmx0` contains `a`, and `mmx1` contains `b`, the above C sequence can be implemented with the following assembler sequence:

```
MOVQ        mmx3, mmx0
PCMPGTW     mmx3, mmx2  ; a > b ? 0xffff : 0
PAND        mmx0, mmx3  ; a > b ? a: 0
PANDN       mmx3, mmx1  ; a > b > 0 : b
POR         mmx0, mmx3  ; r = a > b ? a: b
```

In the above sequence, PCMPGTW, PAND, PANDN, and POR operate, in parallel, on all four elements of the vectors.

Compare and Write Minimum or Maximum

- PMAXUB—Packed Maximum Unsigned Bytes
- PMINUB—Packed Minimum Unsigned Bytes
- PMAXSW—Packed Maximum Signed Words
- PMINSW—Packed Minimum Signed Words

The PMAXUB and PMINUB instructions compare each of the 8-bit unsigned integer values in the first operand with the corresponding 8-bit unsigned integer values in the second operand. The instructions then write the maximum (PMAXUB) or minimum (PMINUB) of the two values for each comparison into the corresponding byte of the destination.

The PMAXSW and PMINSW instructions perform operations analogous to the PMAXUB and PMINUB instructions, except on 16-bit signed integer values.

5.6.9 Logical

The vector-logic instructions perform Boolean logic operations, including AND, OR, and exclusive OR.

And

- PAND—Packed Logical Bitwise AND
- PANDN—Packed Logical Bitwise AND NOT

The PAND instruction performs a bitwise logical AND of the values in the first and second operands and writes the result to the destination.

The PANDN instruction inverts the first operand (creating a one's complement of the operand), ANDs it with the second operand, and writes the result to the destination, and writes the result to the destination. Table 5-5 shows an example.

Table 5-5. Example PANDN Bit Values

Operand1 Bit	Operand1 Bit (Inverted)	Operand2 Bit	PANDN Result Bit
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0

PAND can be used with the value 7FFFFFFF7FFFFFFFh to compute the absolute value of the elements of a 64-bit media floating-point vector operand. This method is equivalent to the x87 FABS (floating-point absolute value) instruction.

Or

- POR—Packed Logical Bitwise OR

The POR instruction performs a bitwise logical OR of the values in the first and second operands and writes the result to the destination.

Exclusive Or

- PXOR—Packed Logical Bitwise Exclusive OR

The PXOR instruction performs a bitwise logical exclusive OR of the values in the first and second operands and writes the result to the destination. PXOR can be used to clear all bits in an MMX register by specifying the same register for both operands. PXOR can also used with the value 8000000080000000h to change the sign bits of the elements of a 64-bit media floating-point vector operand. This method is equivalent to the x87 floating-point change sign (FCHS) instruction.

5.6.10 Save and Restore State

These instructions save and restore the processor state for 64-bit media instructions.

Save and Restore 64-Bit Media and x87 State

- FSAVE—Save x87 and MMX State
- FNSAVE—Save No-Wait x87 and MMX State
- FRSTOR—Restore x87 and MMX State

These instructions save and restore the entire processor state for x87 floating-point instructions and 64-bit media instructions. The instructions save and restore either 94 or 108 bytes of data, depending on the effective operand size.

Assemblers issue FSAVE as an FWAIT instruction followed by an FNSAVE instruction. Thus, FSAVE (but not FNSAVE) reports pending unmasked x87 floating-point exceptions before saving the state. After saving the state, the processor initializes the x87 state by performing the equivalent of an FINIT instruction.

Save and Restore 128-Bit, 64-Bit, and x87 State

- FXSAVE—Save XMM, MMX, and x87 State
- FXRSTOR—Restore XMM, MMX, and x87 State

The FXSAVE and FXRSTOR instructions save and restore the entire 512-byte processor state for 128-bit media instructions, 64-bit media instructions, and x87 floating-point instructions. The architecture supports two memory formats for FXSAVE and FXRSTOR, a 512-byte 32-bit legacy format and a 512-byte 64-bit format. Selection of the 32-bit or 64-bit format is determined by the effective operand size for the FXSAVE and FXRSTOR instructions. For details on the FXSAVE and FXRSTOR Instructions, see the “64-bit Media Instruction Reference” in Volume 5.

FXSAVE and FXRSTOR execute faster than FSAVE/FNSAVE and FRSTOR. However, unlike FSAVE and FNSAVE, FXSAVE does not initialize the x87 state, and like FNSAVE it does not report pending unmasked x87 floating-point exceptions. For details, see “Saving and Restoring State” on page 280.

5.7 Instruction Summary—Floating-Point Instructions

This section summarizes the functions of the floating-point (3DNow! and a few SSE and SSE2) instructions in the 64-bit media instruction subset. These include floating-point instructions that use an MMX register for source or destination and data-conversion instructions that convert from floating-point to integers formats. For a summary of the integer instructions in the 64-bit media instruction subset, including data-conversion instructions that convert from integer to floating-point formats, see “Instruction Summary—Integer Instructions” on page 253.

For a summary of the 128-bit media floating-point instructions, see “Instruction Summary—Floating-Point Instructions” on page 184. For a summary of the x87 floating-point instructions, see Section 6.4. “Instruction Summary” on page 310.

The instructions are organized here by functional group—such as data-transfer, vector arithmetic, and so on. Software running at any privilege level can use any of these instructions, if the CPUID instruction reports support for the instructions (see “Feature Detection” on page 276). More detail on individual instructions is given in the alphabetically organized “64-Bit Media Instruction Reference” in Volume 5.

5.7.1 Syntax

The 64-bit media floating-point instructions have the same syntax rules as those for the 64-bit media integer instructions, described in “Syntax” on page 254, except that the mnemonics of most floating-point instructions begin with the following prefix:

- PF—Packed floating-point

5.7.2 Data Conversion

These data-conversion instructions convert operands from floating-point to integer formats. The instructions take 32-bit or 64-bit floating-point source operands. For data-conversion instructions that take 64-bit integer source operands, see “Data Conversion” on page 257. For data-conversion instructions that take 128-bit source operands, see “Data Conversion” on page 155 and “Data Conversion” on page 190.

Convert Floating-Point to Integer

- CVTPS2PI—Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers
- CVTTPS2PI—Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers, Truncated
- CVTPD2PI—Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers
- CVTTPD2PI—Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers, Truncated
- PF2IW—Packed Floating-Point to Integer Word Conversion
- PF2ID—Packed Floating-Point to Integer Doubleword Conversion

The CVTPS2PI and CVTTPS2PI instructions convert two single-precision (32-bit) floating-point values in the second operand (the low-order 64 bits of an XMM register or a 64-bit memory location) to two 32-bit signed integers, and write the converted values into the first operand (an MMX register). For the CVTPS2PI instruction, if the conversion result is an inexact value, the value is rounded as specified in the rounding control (RC) field of the MXCSR register (“MXCSR Register” on page 115), but for the CVTTPS2PI instruction such a result is truncated (rounded toward zero).

The CVTPD2PI and CVTTPD2PI instructions perform conversions analogous to CVTPS2PI and CVTTPS2PI but for two double-precision (64-bit) floating-point values.

The 3DNow! PF2IW instruction converts two single-precision floating-point values in the second operand (an MMX register or a 64-bit memory location) to two 16-bit signed integer values, sign-extended to 32-bits, and writes the converted values into the first operand (an MMX register). The 3DNow! PF2ID instruction converts two single-precision floating-point values in the second operand to two 32-bit signed integer values, and writes the converted values into the first operand. If the result of either conversion is an inexact value, the value is truncated (rounded toward zero).

As described in “Floating-Point Data Types” on page 251, PF2IW and PF2ID do not fully comply with the IEEE-754 standard. Conversion of some source operands of the C type *float* (IEEE-754 single-precision)—specifically NaNs, infinities, and denormals—are not supported. Attempts to convert such source operands produce undefined results, and no exceptions are generated.

5.7.3 Arithmetic

The floating-point vector-arithmetic instructions perform an arithmetic operation on two floating-point operands. For a description of 3DNow! instruction saturation on overflow and underflow conditions, see “Floating-Point Data Types” on page 251.

Addition

- PFADD—Packed Floating-Point Add

The PFADD instruction adds each single-precision floating-point value in the first operand (an MMX register) to the corresponding single-precision floating-point value in the second operand (an MMX register or 64-bit memory location). The instruction then writes the result of each addition into the corresponding doubleword of the destination.

Subtraction

- PFSUB—Packed Floating-Point Subtract
- PFSUBR—Packed Floating-Point Subtract Reverse

The PFSUB instruction subtracts each single-precision floating-point value in the second operand from the corresponding single-precision floating-point value in the first operand. The instruction then writes the result of each subtraction into the corresponding quadword of the destination.

The PFSUBR instruction performs a subtraction that is the reverse of the PFSUB instruction. It subtracts each value in the first operand from the corresponding value in the second operand. The provision of both the PFSUB and PFSUBR instructions allows software to choose which source operand to overwrite during a subtraction.

Multiplication

- PFMUL—Packed Floating-Point Multiply

The PFMUL instruction multiplies each of the two single-precision floating-point values in the first operand by the corresponding single-precision floating-point value in the second operand and writes the result of each multiplication into the corresponding doubleword of the destination.

Division

For a description of floating-point division techniques, see “Reciprocal Estimation” on page 273. Division is equivalent to multiplication of the dividend by the reciprocal of the divisor.

Accumulation

- PFACC—Packed Floating-Point Accumulate
- PFNACC—Packed Floating-Point Negative Accumulate
- PFPNACC—Packed Floating-Point Positive-Negative Accumulate

The PFACC instruction adds the two single-precision floating-point values in the first operand and writes the result into the low-order word of the destination, and it adds the two single-precision values

in the second operand and writes the result into the high-order word of the destination. Figure 5-17 illustrates the operation.

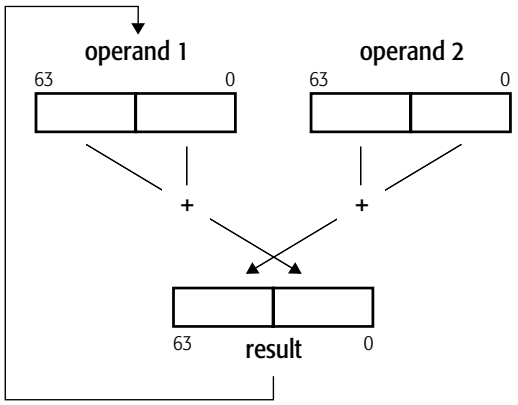


Figure 5-17. PFACC Accumulate Operation

The PFNACC instruction subtracts the first operand's high-order single-precision floating-point value from its low-order single-precision floating-point value and writes the result into the low-order doubleword of the destination, and it subtracts the second operand's high-order single-precision floating-point value from its low-order single-precision floating-point value and writes the result into the high-order doubleword of the destination.

The PFPNACC instruction *subtracts* the first operand's high-order single-precision floating-point value from its low-order single-precision floating-point value and writes the result into the low-order doubleword of the destination, and it *adds* the two single-precision values in the second operand and writes the result into the high-order doubleword of the destination.

PFPNACC is useful in complex-number multiplication, in which mixed positive-negative accumulation must be performed. Assuming that complex numbers are represented as two-element vectors (one element is the real part, the other element is the imaginary part), there is a need to swap the elements of one source operand to perform the multiplication, and there is a need for mixed positive-negative accumulation to complete the parallel computation of real and imaginary results. The PSWAPD instruction can swap elements of one source operand and the PFPNACC instruction can perform the mixed positive-negative accumulation to complete the computation.

Reciprocal Estimation

- PFRCP—Packed Floating-Point Reciprocal Approximation
- PFRCPIT1—Packed Floating-Point Reciprocal, Iteration 1
- PFRCPIT2—Packed Floating-Point Reciprocal or Reciprocal Square Root, Iteration 2

The PFRCP instruction computes the approximate reciprocal of the single-precision floating-point value in the low-order 32 bits of the second operand and writes the result into both doublewords of the first operand.

The PFRCPIT1 instruction performs the first intermediate step in the Newton-Raphson iteration to refine the reciprocal approximation produced by the PFRCP instruction. The first operand contains the input to a previous PFRCP instruction, and the second operand contains the result of the same PFRCP instruction.

The PFRCPIT2 instruction performs the second and final step in the Newton-Raphson iteration to refine the reciprocal approximation produced by the PFRCP instruction or the reciprocal square-root approximation produced by the PFSQRT instructions. The first operand contains the result of a previous PFRCPIT1 or PFRSQIT1 instruction, and the second operand contains the result of a PFRCP or PFRSQRT instruction.

The PFRCP instruction can be used together with the PFRCPIT1 and PFRCPIT2 instructions to increase the accuracy of a single-precision significand.

Reciprocal Square Root

- PFRSQRT—Packed Floating-Point Reciprocal Square Root Approximation
- PFRSQIT1—Packed Floating-Point Reciprocal Square Root, Iteration 1

The PFRSQRT instruction computes the approximate reciprocal square root of the single-precision floating-point value in the low-order 32 bits of the second operand and writes the result into each doubleword of the first operand. The second operand is a single-precision floating-point value with a 24-bit significand. The result written to the first operand is accurate to 15 bits. Negative operands are treated as positive operands for purposes of reciprocal square-root computation, with the sign of the result the same as the sign of the source operand.

The PFRSQIT1 instruction performs the first step in the Newton-Raphson iteration to refine the reciprocal square-root approximation produced by the PFSQRT instruction. The first operand contains the input to a previous PFRSQRT instruction, and the second operand contains the square of the result of the same PFRSQRT instruction.

The PFRSQRT instruction can be used together with the PFRSQIT1 instruction and the PFRCPIT2 instruction (described in “Reciprocal Estimation” on page 273) to increase the accuracy of a single-precision significand.

5.7.4 Compare

The floating-point vector-compare instructions compare two operands, and they either write a mask or they write the maximum or minimum value.

Compare and Write Mask

- PFCMPEQ—Packed Floating-Point Compare Equal
- PFCMPGT—Packed Floating-Point Compare Greater Than

- PFCMPGE—Packed Floating-Point Compare Greater or Equal

The PFCMP*x* instructions compare each of the two single-precision floating-point values in the first operand with the corresponding single-precision floating-point value in the second operand. The instructions then write the result of each comparison into the corresponding doubleword of the destination. If the comparison test (equal, greater than, greater or equal) is true, the result is a mask of all 1s. If the comparison test is false, the result is a mask of all 0s.

Compare and Write Minimum or Maximum

- PFMAX—Packed Floating-Point Maximum
- PFMIN—Packed Floating-Point Minimum

The PFMAX and PFMIN instructions compare each of the two single-precision floating-point values in the first operand with the corresponding single-precision floating-point value in the second operand. The instructions then write the maximum (PFMAX) or minimum (PFMIN) of the two values for each comparison into the corresponding doubleword of the destination.

The PFMIN and PFMAX instructions are useful for clamping, such as color clamping in 3D geometry and rasterization. They can also be used to avoid branching.

5.8 Instruction Effects on Flags

The 64-bit media instructions do not read or write any flags in the rFLAGS register, nor do they write any exception-status flags in the x87 status-word register, nor is their execution dependent on any mask bits in the x87 control-word register. The only x87 state affected by the 64-bit media instructions is described in “Actions Taken on Executing 64-Bit Media Instructions” on page 278.

5.9 Instruction Prefixes

Instruction prefixes, in general, are described in “Instruction Prefixes” on page 76. The following restrictions apply to the use of instruction prefixes with 64-bit media instructions.

5.9.1 Supported Prefixes

The following prefixes can be used with 64-bit media instructions:

- *Address-Size Override*—The 67h prefix affects only operands in memory. The prefix is ignored by all other 64-bit media instructions.
- *Operand-Size Override*—The 66h prefix is used to form the opcodes of certain 64-bit media instructions. The prefix is ignored by all other 64-bit media instructions.
- *Segment Overrides*—The 2Eh (CS), 36h (SS), 3Eh (DS), 26h (ES), 64h (FS), and 65h (GS) prefixes affect only operands in memory. In 64-bit mode, the contents of the CS, DS, ES, SS segment registers are ignored.

- *REP*—The F2 and F3h prefixes do not function as repeat prefixes for 64-bit media instructions. Instead, they are used to form the opcodes of certain 64-bit media instructions. The prefixes are ignored by all other 64-bit media instructions.
- *REX*—The REX prefixes affect operands that reference a GPR or XMM register when running in 64-bit mode. It allows access to the full 64-bit width of any of the 16 extended GPRs and to any of the 16 extended XMM registers. The REX prefix also affects the FXSAVE and FXRSTOR instructions, in which it selects between two types of 512-byte memory-image format, as described in "Media and x87 Processor State" in Volume 2. The prefix is ignored by all other 64-bit media instructions.

5.9.2 Special-Use and Reserved Prefixes

The following prefixes are used as opcode bytes in some 64-bit media instructions and are reserved in all other 64-bit media instructions:

- *Operand-Size Override*—The 66h prefix.
- *REP*—The F2 and F3h prefixes.

5.9.3 Prefixes That Cause Exceptions

The following prefixes cause an exception:

- *LOCK*—The F0h prefix causes an invalid-opcode exception when used with 64-bit media instructions.

5.10 Feature Detection

Before executing 64-bit media instructions, software should determine whether the processor supports the technology by executing the CPUID instruction. "Feature Detection" on page 79 describes how software uses the CPUID instruction to detect feature support. For full support of the 64-bit media instructions documented here, the following features require detection:

- MMX instructions, indicated by bit 23 of CPUID function 1 and function 8000_0001h.
- 3DNow! instructions, indicated by bit 31 of CPUID function 8000_0001h.
- MMX extensions, indicated by bit 22 of CPUID function 8000_0001h.
- 3DNow! extensions, indicated by bit 30 of CPUID function 8000_0001h.
- SSE instructions, indicated by bit 25 of CPUID function 8000_0001h.
- SSE2 instruction extensions, indicated by bit 26 of CPUID function 8000_0001h.
- SSE3 instruction extensions, indicated by bit 0 of CPUID function 0000_0001h.
- SSE4A instruction extensions, indicated by bit 6 of CPUID function 8000_0001h.

Software may also wish to check for the following support, because the FXSAVE and FXRSTOR instructions execute faster than FSAVE and FRSTOR:

- FXSAVE and FXRSTOR, indicated by bit 24 of CPUID function 1 and function 8000_0001h.

Software that runs in long mode should also check for the following support:

- Long Mode, indicated by bit 29 of CPUID function 8000_0001h.

See “CPUID” in Volume 3 for details on the CPUID instruction and Appendix D of that volume for information on detemining support for specific instruction subsets.

If the FXSAVE and FXRSTOR instructions are to be used, the operating system must support these instructions by having set CR4.OSFXSR = 1. If the MMX floating-point-to-integer data-conversion instructions (CVTPS2PI, CVTTPS2PI, CVTPD2PI, or CVTTPD2PI) are used, the operating system must support the FXSAVE and FXRSTOR instructions and SIMD floating-point exceptions (by having set CR4.OSXMMEXCPT = 1). For details, see “System-Control Registers” in Volume 2.

5.11 Exceptions

64-bit media instructions can generate two types of exceptions:

- *General-Purpose Exceptions*, described below in “General-Purpose Exceptions”
- *x87 Floating-Point Exceptions (#MF)*, described in “x87 Floating-Point Exceptions (#MF)” on page 278

All exceptions that occur while executing 64-bit media instructions can be handled by legacy exception handlers used for general-purpose instructions and x87 floating-point instructions.

5.11.1 General-Purpose Exceptions

The sections below list exceptions generated and not generated by general-purpose instructions. For a summary of the general-purpose exception mechanism, see “Interrupts and Exceptions” on page 91. For details about each exception and its potential causes, see “Exceptions and Interrupts” in Volume 2.

5.11.1.1 Exceptions Generated

The 64-bit media instructions can generate the following general-purpose exceptions:

- #DB—Debug Exception (Vector 1)
- #UD—Invalid-Opcode Exception (Vector 6)
- #DF—Double-Fault Exception (Vector 8)
- #SS—Stack Exception (Vector 12)
- #GP—General-Protection Exception (Vector 13)
- #PF—Page-Fault Exception (Vector 14)
- #MF—x87 Floating-Point Exception-Pending (Vector 16)
- #AC—Alignment-Check Exception (Vector 17)
- #MC—Machine-Check Exception (Vector 18)
- #XF—SIMD Floating-Point Exception (Vector 19)—Only by the CVTPS2PI, CVTTPS2PI, CVTPD2PI, and CVTTPD2PI instructions.

An invalid-opcode exception (#UD) can occur if a required CPUID feature flag is not set (see "Feature Detection" on page 276), or if an attempt is made to execute a 64-bit media instruction and the operating system has set the floating-point software-emulation (EM) bit in control register 0 to 1 (CR0.EM = 1).

For details on the system control-register bits, see "System-Control Registers" in Volume 2. For details on the machine-check mechanism, see "Machine Check Mechanism" in Volume 2.

For details on #MF exceptions, see "x87 Floating-Point Exceptions (#MF)" on page 278.

5.11.1.2 Exceptions Not Generated

The 64-bit media instructions do not generate the following general-purpose exceptions:

- #DE—Divide-By-Zero-Error Exception (Vector 0)
- Non-Maskable-Interrupt Exception (Vector 2)
- #BP—Breakpoint Exception (Vector 3)
- #OF—Overflow Exception (Vector 4)
- #BR—Bound-Range Exception (Vector 5)
- #NM—Device-Not-Available Exception (Vector 7)
- Coprocessor-Segment-Overrun Exception (Vector 9)
- #TS—Invalid-TSS Exception (Vector 10)
- #NP—Segment-Not-Present Exception (Vector 11)

For details on all general-purpose exceptions, see "Exceptions and Interrupts" in Volume 2.

5.11.2 x87 Floating-Point Exceptions (#MF)

The 64-bit media instructions do not generate x87 floating-point (#MF) exceptions as a consequence of their own computations. However, an #MF exception can occur during the execution of a 64-bit media instruction, due to a prior x87 floating-point instruction. Specifically, if an unmasked x87 floating-point exception is pending at the instruction boundary of the next 64-bit media instruction, the processor asserts the FERR# output signal. For details about the x87 floating-point exceptions and the FERR# output signal, see Section 6.8.2. "x87 Floating-Point Exception Causes" on page 329.

5.12 Actions Taken on Executing 64-Bit Media Instructions

The MMX registers are mapped onto the low 64 bits of the 80-bit x87 floating-point physical registers, FPR0–FPR7, described in Section 6.2. "Registers" on page 286. The MMX instructions do not use the x87 stack-addressing mechanism. However, 64-bit media instructions write certain values in the x87 top-of-stack pointer, tag bits, and high bits of the FPR0–FPR7 data registers.

Specifically, the processor performs the following x87-related actions atomically with the execution of 64-bit media instructions:

- *Top-Of-Stack Pointer (TOP)*—The processor clears the x87 top-of-stack pointer (bits 13–11 in the x87 status word register) to all 0s during the execution of every 64-bit media instruction, causing it to point to the *mmx0* register.
- *Tag Bits*—During the execution of every 64-bit media instruction, except the EMMS and FEMMS instructions, the processor changes the tag state for all eight MMX registers to *full*, as described below. In the case of EMMS and FEMMS, the processor changes the tag state for all eight MMX registers to *empty*, thus initializing the stack for an x87 floating-point procedure.
- *Bits 79:64*—During the execution of every 64-bit media instruction that writes a result to an MMX register, the processor writes the result data to a 64-bit MMX register (the low 64 bits of the associated 80-bit x87 floating-point physical register) and sets the exponent and sign bits (the high 16 bits of the associated 80-bit x87 floating-point physical register) to all 1s. In the x87 environment, the effect of setting the high 16 bits to all 1s indicates that the contents of the low 64 bits are not finite numbers. Such a designation prevents an x87 floating-point instruction from interpreting the data as a finite x87 floating-point number.

The rest of the x87 floating-point processor state—the entire x87 control-word register, the remaining fields of the status-word register, and the error pointers (instruction pointer, data pointer, and last opcode register)—is not affected by the execution of 64-bit media instructions.

The 2-bit tag fields defined by the x87 architecture for each x87 data register, and stored in the x87 tag-word register (also called the floating-point tag word, or FTW), characterize the contents of the MMX registers. The tag bits are visible to software only after an FSAVE or FNSAVE (but not FXSAVE) instruction, as described in “Media and x87 Processor State” in Volume 2. Internally, however, the processor maintains only a one-bit representation of each 2-bit tag field. This single bit indicates whether the associated register is *empty* or *full*. Table 5-6 on page 279 shows the mapping between the 1-bit internal tag—which is referred to in this chapter by its *empty* or *full* state—and the 2-bit architectural tag.

Table 5-6. Mapping Between Internal and Software-Visible Tag Bits

Architectural State		Internal State[1]
State	Binary Value	
Valid	00	Full (0)
Zero	01	
Special (NaN, infinity, denormal)[2]	10	
Empty	11	Empty (1)

Note:
1. *For a more detailed description of this mapping, see “Deriving FSAVE Tag Field from FXSAVE Tag Field” in Volume 2.*
2. *The 64-bit media floating point (3DNow!™) instructions do not support NaNs, infinities, and denormals.*

When the processor executes an FSAVE or FNSAVE (but not FXSAVE) instruction, it changes the internal 1-bit tag state to its 2-bit architectural tag by reading the data in all 80 bits of the physical data

registers and using the mapping in Table 5-6. For example, if the value in the high 16 bits of the 80-bit physical register indicate a NaN, the two tag bits for that register are changed to a binary value of 10 before the x87 status word is written to memory.

The tag bits have no effect on the execution of 64-bit media instructions or their interpretation of the contents of the MMX registers. However, the converse is not true: execution of 64-bit media instructions that write to an MMX register alter the tag bits and thus may affect execution of subsequent x87 floating-point instructions.

For a more detailed description of the mapping shown in Table 5-6, see “Deriving FSAVE Tag Field from FXSAVE Tag Field” in Volume 2 and its accompanying text.

5.13 Mixing Media Code with x87 Code

5.13.1 Mixing Code

Software may freely mix 64-bit media instructions (integer or floating-point) with 128-bit media instructions (integer or floating-point) and general-purpose instructions in a single procedure. However, before transitioning from a 64-bit media procedure—or a 128-bit media procedure that accesses an MMX™ register—to an x87 procedure, or to software that may eventually branch to an x87 procedure, software should clear the MMX state, as described immediately below.

5.13.2 Clearing MMX™ State

Software should separate 64-bit media procedures, 128-bit media procedures, or dynamic link libraries (DLLs) that access MMX registers from x87 floating-point procedures or DLLs by clearing the MMX state with the EMMS or FEMMS instruction before leaving a 64-bit media procedure, as described in “Exit Media State” on page 255.

The 64-bit media instructions and x87 floating-point instructions interpret the contents of their aliased MMX and x87 registers differently. Because of this, software should not exchange register data between 64-bit media and x87 floating-point procedures, or use conditional branches at the end of loops that might jump to code of the other type. Software must not rely on the contents of the aliased MMX and x87 registers across such code-type transitions. If a transition to an x87 procedure occurs from a 64-bit media procedure that does not clear the MMX state, the x87 stack may overflow.

5.14 State-Saving

5.14.1 Saving and Restoring State

In general, system software should save and restore MMX™ and x87 state between task switches or other interventions in the execution of 64-bit media procedures. Virtually all modern operating systems running on x86 processors implement preemptive multitasking that handle saving and restoring of state across task switches, independent of hardware task-switch support.

No changes are needed to the x87 register-saving performed by 32-bit operating systems, exception handlers, or device drivers. The same support provided in a 32-bit operating system's device-not-available (#NM) exception handler by any of the x87-register save/restore instructions described below also supports saving and restoring the MMX registers.

However, application procedures are also free to save and restore MMX and x87 state at any time they deem useful.

5.14.2 State-Saving Instructions

Software running at any privilege level may save and restore 64-bit media and x87 state by executing the FSAVE, FNSAVE, or FXSAVE instruction. Alternatively, software may use move instructions for saving only the contents of the MMX registers, rather than the complete 64-bit media and x87 state. For example, when saving MMX register values, use eight MOVQ instructions.

5.14.2.1 FSAVE/FNSAVE and FRSTOR Instructions

The FSAVE, FNSAVE, and FRSTOR instructions are described in "Save and Restore 64-Bit Media and x87 State" on page 269. After saving state with FSAVE or FNSAVE, the tag bits for all MMX and x87 registers are changed to *empty* and thus available for a new procedure. Thus, FSAVE and FNSAVE also perform the state-clearing function of EMMS or FEMMS.

5.14.2.2 FXSAVE and FXRSTOR Instructions

The FSAVE, FNSAVE, and FRSTOR instructions are described in "Save and Restore 128-Bit, 64-Bit, and x87 State" on page 270. The FXSAVE and FXRSTOR instructions execute faster than FSAVE/FNSAVE and FRSTOR because they do not save and restore the x87 error pointers (described in Section 6.2.5. "Pointers and Opcode State" on page 295) except in the relatively rare cases in which the exception-summary (ES) bit in the x87 status word (register image for FXSAVE, memory image for FXRSTOR) is set to 1, indicating that an unmasked x87 exception has occurred.

Unlike FSAVE and FNSAVE, however, FXSAVE does not alter the tag bits (thus, it does not perform the state-clearing function of EMMS or FEMMS). The state of the saved MMX and x87 registers is retained, thus indicating that the registers may still be valid (or whatever other value the tag bits indicated prior to the save). To invalidate the contents of the MMX and x87 registers after FXSAVE, software must explicitly execute an FINIT instruction. Also, FXSAVE (like FNSAVE) and FXRSTOR do not check for pending unmasked x87 floating-point exceptions. An FWAIT instruction can be used for this purpose.

For details about the FXSAVE and FXRSTOR memory formats, see "Media and x87 Processor State" in Volume 2.

5.15 Performance Considerations

In addition to typical code optimization techniques, such as those affecting loops and the inlining of function calls, the following considerations may help improve the performance of application programs written with 64-bit media instructions.

These are implementation-independent performance considerations. Other considerations depend on the hardware implementation. For information about such implementation-dependent considerations and for more information about application performance in general, see the data sheets and the software-optimization guides relating to particular hardware implementations.

5.15.1 Use Small Operand Sizes

The performance advantages available with 64-bit media operations is to some extent a function of the data sizes operated upon. The smaller the data size, the more data elements that can be packed into single 64-bit vectors. The parallelism of computation increases as the number of elements per vector increases.

5.15.2 Reorganize Data for Parallel Operations

Much of the performance benefit from the 64-bit media instructions comes from the parallelism inherent in vector operations. It can be advantageous to reorganize data before performing arithmetic operations so that its layout after reorganization maximizes the parallelism of the arithmetic operations.

The speed of memory access is particularly important for certain types of computation, such as graphics rendering, that depend on the regularity and locality of data-memory accesses. For example, in matrix operations, performance is high when operating on the rows of the matrix, because row bytes are contiguous in memory, but lower when operating on the columns of the matrix, because column bytes are not contiguous in memory and accessing them can result in cache misses. To improve performance for operations on such columns, the matrix should first be transposed. Such transpositions can, for example, be done using a sequence of unpacking or shuffle instructions.

5.15.3 Remove Branches

Branch can be replaced with 64-bit media instructions that simulate predicated execution or conditional moves, as described in “Branch Removal” on page 244. Where possible, break long dependency chains into several shorter dependency chains which can be executed in parallel. This is especially important for floating-point instructions because of their longer latencies.

5.15.4 Align Data

Data alignment is particularly important for performance when data written by one instruction is read by a subsequent instruction soon after the write, or when accessing streaming (non-temporal) data—data that will not be reused and therefore should not be cached. These cases may occur frequently in 64-bit media procedures.

Accesses to data stored at unaligned locations may benefit from on-the-fly software alignment or from repetition of data at different alignment boundaries, as required by different loops that process the data.

5.15.5 Organize Data for Cacheability

Pack small data structures into cache-line-size blocks. Organize frequently accessed constants and coefficients into cache-line-size blocks and prefetch them. Procedures that access data arranged in memory-bus-sized blocks, or memory-burst-sized blocks, can make optimum use of the available memory bandwidth.

For data that will be used only once in a procedure, consider using non-cacheable memory. Accesses to such memory are not burdened by the overhead of cache protocols.

5.15.6 Prefetch Data

Media applications typically operate on large data sets. Because of this, they make intensive use of the memory bus. Memory latency can be substantially reduced—especially for data that will be used only once—by prefetching such data into various levels of the cache hierarchy. Software can use the PREFETCH*x* instructions very effectively in such cases, as described in “Cache and Memory Management” on page 71.

Some of the best places to use prefetch instructions are inside loops that process large amounts of data. If the loop goes through less than one cache line of data per iteration, partially unroll the loop to obtain multiple iterations of the loop within a cache line. Try to use virtually all of the prefetched data. This usually requires unit-stride memory accesses—those in which all accesses are to contiguous memory locations.

5.15.7 Retain Intermediate Results in MMX™ Registers

Keep intermediate results in the MMX registers as much as possible, especially if the intermediate results are used shortly after they have been produced. Avoid spilling intermediate results to memory and reusing them shortly thereafter.

6 x87 Floating-Point Programming

This chapter describes the x87 floating-point programming model. This model supports all aspects of the legacy x87 floating-point model and complies with the IEEE 754 and 854 standards for binary floating-point arithmetic. In hardware implementations of the AMD64 architecture, support for specific features of the x87 programming model are indicated by the CPUID feature bits, as described in “Feature Detection” on page 327.

6.1 Overview

Floating-point software is typically written to manipulate numbers that are very large or very small, that require a high degree of precision, or that result from complex mathematical operations, such as transcendentals. Applications that take advantage of floating-point operations include geometric calculations for graphics acceleration, scientific, statistical, and engineering applications, and process control.

6.1.1 Capabilities

The advantages of using x87 floating-point instructions include:

- Representation of all numbers in common IEEE-754/854 formats, ensuring replicability of results across all platforms that conform to IEEE-754/854 standards.
- Availability of separate floating-point registers. Depending on the hardware implementation of the architecture, this may allow execution of x87 floating-point instructions in parallel with execution of general-purpose and 128-bit media instructions.
- Availability of instructions that compute absolute value, change-of-sign, round-to-integer, partial remainder, and square root.
- Availability of instructions that compute transcendental values, including 2^x-1, cosine, partial arc tangent, partial tangent, sine, sine with cosine, y*$\log_2$x, and y*$\log_2$(x+1). The cosine, partial arc tangent, sine, and sine with cosine instructions use angular values expressed in radians for operands and results.
- Availability of instructions that load common constants, such as $\log_2$e, $\log_2$10, $\log_{10}$2, $\log_e$2, Pi, 1, and 0.

x87 instructions operate on data in three floating-point formats—32-bit single-precision, 64-bit double-precision, and 80-bit double-extended-precision (sometimes called extended precision)—as well as integer, and 80-bit packed-BCD formats.

x87 instructions carry out all computations using the 80-bit double-extended-precision format. When an x87 instruction reads a number from memory in 80-bit double-extended-precision format, the number can be used directly in computations, without conversion. When an x87 instruction reads a number in a format other than double-extended-precision format, the processor first converts the

number into double-extended-precision format. The processor can convert numbers back to specific formats, or leave them in double-extended-precision format when writing them to memory.

Most x87 operations for addition, subtraction, multiplication, and division specify two source operands, the first of which is replaced by the result. Instructions for subtraction and division have reverse forms which swap the ordering of operands.

6.1.2 Origins

In 1979, AMD introduced the first floating-point coprocessor for microprocessors—the AM9511 arithmetic circuit. This coprocessor performed 32-bit floating-point operations under microprocessor control. In 1980, AMD introduced the AM9512, which performed 64-bit floating-point operations. These coprocessors were second-sourced as the 8231 and 8232 coprocessors. Before then, programmers working with general-purpose microprocessors had to use much slower, vendor-supplied software libraries for their floating-point needs.

In 1985, the Institute of Electrical and Electronics Engineers published the *IEEE Standard for Binary Floating-Point Arithmetic*, also referred to as the *ANSI/IEEE Std 754-1985* standard, or *IEEE 754*. This standard defines the data types, operations, and exception-handling methods that are the basis for the x87 floating-point technology implemented in the legacy x86 architecture. In 1987, the IEEE published a more general radix-independent version of that standard, called the ANSI/IEEE Std 854-1987 standard, or *IEEE 854* for short. The AMD64 architecture complies with both the IEEE 754 and IEEE 854 standards.

6.1.3 Compatibility

x87 floating-point instructions can be executed in any of the architecture's operating modes. Existing x87 binary programs run in legacy and compatibility modes without modification. The support provided by the AMD64 architecture for such binaries is identical to that provided by legacy x86 architectures.

To run in 64-bit mode, x87 floating-point programs must be recompiled. The recompilation has no side effects on such programs, other than to make available the extended general-purpose registers and 64-bit virtual address space.

6.2 Registers

Operands for the x87 instructions are located in x87 registers or memory. Figure 6-1 on page 287 shows an overview of the x87 registers.

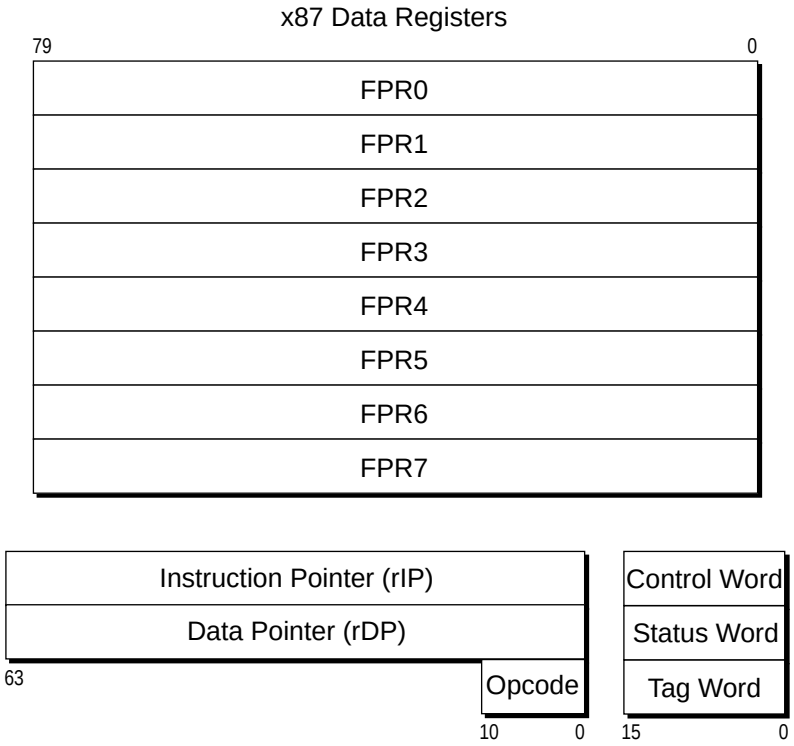


Figure 6-1. x87 Registers

These registers include eight 80-bit data registers, three 16-bit registers that hold the x87 control word, status word, and tag word, two 64-bit registers that hold instruction and data pointers, and an 11-bit register that holds a permutation of an x87 opcode.

6.2.1 x87 Data Registers

Figure 6-2 on page 288 shows the eight 80-bit data registers in more detail. Typically, x87 instructions reference these registers as a stack. x87 instructions store operands only in these 80-bit registers or in memory. They do not (with two exceptions) access the GPR registers, and they do not access the YMM/XMM registers.

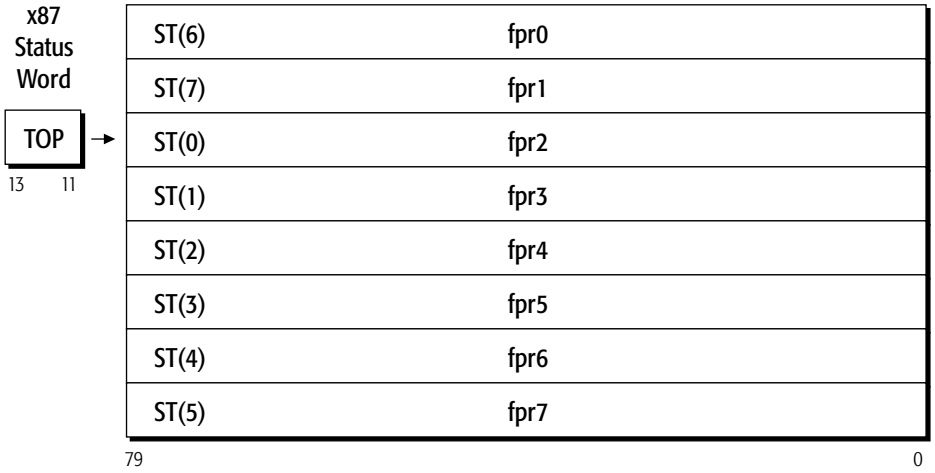


Figure 6-2. x87 Physical and Stack Registers

6.2.1.1 Stack Organization

The bank of eight physical data registers, FPR0–FPR7, are organized internally as a stack, ST(0)–ST(7). The stack functions like a circular modulo-8 buffer. The stack top can be set by software to start at any register position in the bank. Many instructions access the top of stack as well as individual registers relative to the top of stack.

6.2.1.2 Stack Pointer

Bits 13:11 of the x87 status word (“x87 Status Word Register (FSW)” on page 289) are the top-of-stack pointer (TOP). The TOP specifies the mapping of the stack registers onto the physical registers. The TOP contains the physical-register index of the location of the top of stack, ST(0). Instructions that load operands from memory into an x87 register first decrement the stack pointer and then copy the operand (often with conversion to the double-extended-precision format) from memory into the decremented top-of-stack register. Instructions that store operands from an x87 register to memory copy the operand (often with conversion from the double-extended-precision format) in the top-of-stack register to memory and then increment the stack pointer.

Figure 6-2 shows the mapping between stack registers and physical registers when the TOP has the value 2. Modulo-8 wraparound addressing is used. Pushing a new element onto this stack—for example with the FLDZ (floating-point load +0.0) instruction—decrements the TOP to 1, so that ST(0) refers to FPR1, and the new top-of-stack is loaded with +0.0.

The architecture provides alternative versions of many instructions that either modify or do not modify the TOP as a side effect. For example, FADDP (floating-point add and pop) behaves exactly like FADD (floating-point add), except that it pops the stack after completion. Programs that use the x87 registers as a flat register file rather than as a stack would use non-popping versions of instructions to

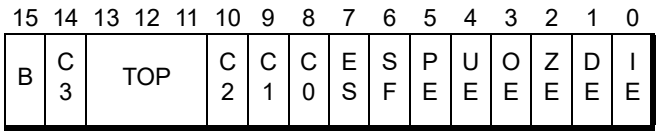
ensure that the TOP remains unchanged. However, loads (pushes) without corresponding pops can cause the stack to overflow, which occurs when a value is pushed or loaded into an x87 register that is not empty (as indicated by the register's tag bits). To prevent overflow, the FXCH (floating-point exchange) instruction can be used to access stack registers, giving the appearance of a flat register file, but all x87 programs must be aware of the register file's stack organization.

The FINCSTP and FDECSTP instructions can be used to increment and decrement, respectively, the TOP, modulo-8, allowing the stack top to wrap around to the bottom of the eight-register file when incremented beyond the top of the file, or to wrap around to the top of the register file when decremented beyond the bottom of the file. Neither the x87 tag word nor the contents of the floating-point stack itself is updated when these instructions are used.

6.2.2 x87 Status Word Register (FSW)

The 16-bit x87 status word register contains information about the state of the floating-point unit, including the top-of-stack pointer (TOP), four condition-code bits, exception-summary flag, stack-fault flag, and six x87 floating-point exception flags. Figure 6-3 on page 290 shows the format of this register. All bits can be read and written, however values written to the B and ES bits (bits 15 and 7) are ignored.

The FRSTOR and FXRSTOR instructions load the status word from memory. The FSTSW, FNSTSW, FSAVE, FNSAVE, FXSAVE, FSTENV, and FNSTENV instructions store the status word to memory. The FCLEX and FNCLEX instructions clear the exception flags. The FINIT and FNINIT instructions clear all bits in the status-word.



Bits	Mnemonic	Description
15	B	x87 Floating-Point Unit Busy
14	C3	Condition Code
13:11	TOP	Top of Stack Pointer 000 = FPR0 111 = FPR7
10	C2	Condition Code
9	C1	Condition Code
8	C0	Condition Code
7	ES	Exception Status
6	SF	Stack Fault
	Exception Flags	
5	PE	Precision Exception
4	UE	Underflow Exception
3	OE	Overflow Exception
2	ZE	Zero-Divide Exception
1	DE	Denormalized-Operand Exception
0	IE	Invalid-Operation Exception

Figure 6-3. x87 Status Word Register (FSW)

The bits in the x87 status word are defined immediately below, starting with bit 0. The six exception flags (IE, DE, ZE, OE, UE, PE) plus the stack fault (SF) flag are sticky bits. Once set by the processor, such a bit remains set until software clears it. For details about the causes of x87 exceptions indicated by bits 6:0, see “x87 Floating-Point Exception Causes” on page 329. For details about the masking of x87 exceptions, see “x87 Floating-Point Exception Masking” on page 333.

6.2.2.1 Invalid-Operation Exception (IE)

Bit 0. The processor sets this bit to 1 when an invalid-operation exception occurs. These exceptions are caused by many types of errors, such as an invalid operand or by stack faults. When a stack fault causes an IE exception, the stack fault (SF) exception bit is also set.

6.2.2.2 Denormalized-Operand Exception (DE)

Bit 1. The processor sets this bit to 1 when one of the source operands of an instruction is in denormalized form. (See “Denormalized (Tiny) Numbers” on page 303.)

6.2.2.3 Zero-Divide Exception (ZE)

Bit 2. The processor sets this bit to 1 when a non-zero number is divided by zero.

6.2.2.4 Overflow Exception (OE)

Bit 3. The processor sets this bit to 1 when the absolute value of a rounded result is larger than the largest representable normalized floating-point number for the destination format. (See “Normalized Numbers” on page 303.)

6.2.2.5 Underflow Exception (UE)

Bit 4. The processor sets this bit to 1 when the absolute value of a rounded non-zero result is too small to be represented as a normalized floating-point number for the destination format. (See “Normalized Numbers” on page 303.)

The underflow exception has an unusual behavior. When masked by the UM bit (bit 4 of the x87 control word), the processor only reports a UE exception if the UE occurs *together with* a precision exception (PE).

6.2.2.6 Precision Exception (PE)

Bit 5. The processor sets this bit to 1 when a floating-point result, after rounding, differs from the infinitely precise result and thus cannot be represented exactly in the specified destination format. The PE exception is also called the *inexact-result* exception.

6.2.2.7 Stack Fault (SF)

Bit 6. The processor sets this bit to 1 when a stack overflow (due to a push or load into a non-empty stack register) or stack underflow (due to referencing an empty stack register) occurs in the x87 stack-register file. When either of these conditions occur, the processor also sets the invalid-operation exception (IE) flag, and the processor distinguishes overflow from underflow by writing the condition-code 1 (C1) bit (C1 = 1 for overflow, C1 = 0 for underflow). Unlike the flags for the other x87 exceptions, the SF flag does not have a corresponding mask bit in the x87 control word.

If, subsequent to the instruction that caused the SF bit to be set, a second invalid-operation exception (IE) occurs due to an invalid operand in an arithmetic instruction (i.e., not a stack fault), and if software has not cleared the SF bit between the two instructions, the SF bit will remain set.

6.2.2.8 Exception Status (ES)

Bit 7. The processor calculates the value of this bit at each instruction boundary and sets the bit to 1 when one or more unmasked floating-point exceptions occur. If the ES bit has already been set by the action of some prior instruction, the processor invokes the #MF exception handler when the next non-control x87 or 64-bit media instruction is executed. (See “Control” on page 323 for a definition of control instructions).

The ES bit can be written, but the written value is ignored. Like the SF bit, the ES bit does not have a corresponding mask bit in the x87 control word.

6.2.2.9 Top-of-Stack Pointer (TOP)

Bits 13:11. The TOP contains the physical register index of the location of the top of stack, ST(0). It thus specifies the mapping of the x87 stack registers, ST(0)–ST(7), onto the x87 physical registers, FPR0–FPR7. The processor changes the TOP during any instructions that pushes or pops the stack. For details on how the stack works, see “Stack Organization” on page 288.

6.2.2.10 Condition Codes (C3–C0)

Bits 14 and 10:8. The processor sets these bits according to the result of arithmetic, compare, and other instructions. In certain cases, other status-word flags can be used together with the condition codes to determine the result of an operation, including stack overflow, stack underflow, sign, least-significant quotient bits, last-rounding direction, and out-of-range operand. For details on how each instruction sets the condition codes, see “x87 Floating-Point Instruction Reference” in Volume 5.

6.2.2.11 x87 Floating-Point Unit Busy (B)

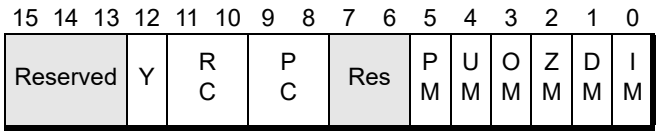
Bit 15. The processor sets the value of this bit equal to the calculated value of the ES bit, bit 7. This bit can be written, but the written value is ignored. The bit is included only for backward-compatibility with the 8087 coprocessor, in which it indicates that the coprocessor is busy.

For further details about the x87 floating-point exceptions, see “x87 Floating-Point Exception Causes” on page 329.

6.2.3 x87 Control Word Register (FCW)

The 16-bit x87 control word register allows software to manage certain x87 processing options, including rounding, precision, and masking of the six x87 floating-point exceptions (any of which is reported as an #MF exception). Figure 6-4 shows the format of the control word. All bits, except reserved bits, can be read and written.

The FLDCW, FRSTOR, and FXRSTOR instructions load the control word from memory. The FSTCW, FNSTCW, FSAVE, FNSAVE, and FXSAVE instructions store the control word to memory. The FINIT and FNINIT instructions initialize the control word with the value 037Fh, which specifies round-to-nearest, all exceptions masked, and double-extended precision (64-bit).



Bits	Mnemonic	Description
12	Y	Infinity Bit (80287 compatibility)
11:10	RC	Rounding Control
9:8	PC	Precision Control
		#MF Exception Masks
5	PM	Precision Exception Mask
4	UM	Underflow Exception Mask
3	OM	Overflow Exception Mask
2	ZM	Zero-Divide Exception Mask
1	DM	Denormalized-Operand Exception Mask
0	IM	Invalid-Operation Exception Mask

Figure 6-4. x87 Control Word Register (FCW)

Starting from bit 0, the bits are:

6.2.3.1 Exception Masks (PM, UM, OM, ZM, DM, IM)

Bits 5:0. Software can set these bits to mask, or clear these bits to unmask, the corresponding six types of x87 floating-point exceptions (PE, UE, OE, ZE, DE, IE), which are reported in the x87 status word as described in “x87 Status Word Register (FSW)” on page 289. A bit masks its exception type when set to 1, and unmasks it when cleared to 0.

Masking a type of exception causes the processor to handle all subsequent instances of the exception type in a default way. Unmasking the exception type causes the processor to branch to the #MF exception service routine when an exception occurs. For details about the processor’s responses to masked and unmasked exceptions, see “x87 Floating-Point Exception Causes” on page 329.

6.2.3.2 Precision Control (PC)

Bits 9:8. Software can set this field to specify the precision of x87 floating-point calculations, as shown in Table 6-1. Details on each precision are given in “Data Types” on page 299. The default precision is double-extended-precision. Precision control affects only the F(I)ADDx, F(I)SUB*x*, F(I)MULx, F(I)DIV*x*, and FSQRT instructions. For further details on precision, see “Precision” on page 309.

Table 6-1. Precision Control (PC) Summary

PC Value (binary)	Data Type
00	Single precision
01	*reserved*
10	Double precision
11	Double-extended precision (default)

6.2.3.3 Rounding Control (RC)

Bits 11:10. Software can set this field to specify how the results of x87 instructions are to be rounded. Table 6-2 lists the four rounding modes, which are defined by the IEEE 754 standard.

Table 6-2. Types of Rounding

RC Value	Mode	Type of Rounding
00 (default)	Round to nearest	The rounded result is the representable value closest to the infinitely precise result. If equally close, the even value (with least-significant bit 0) is taken.
01	Round down	The rounded result is closest to, but no greater than, the infinitely precise result.
10	Round up	The rounded result is closest to, but no less than, the infinitely precise result.
11	Round toward zero	The rounded result is closest to, but no greater in absolute value than, the infinitely precise result.

Round-to-nearest is the default rounding mode. It provides a statistically unbiased estimate of the true result, and is suitable for most applications. Rounding modes apply to all arithmetic operations except comparison and remainder. They have no effect on operations that produce not-a-number (NaN) results. For further details on rounding, see “Rounding” on page 309.

6.2.3.4 Infinity Bit (Y)

Bit 12. This bit is obsolete. It can be read and written, but the value has no meaning. On pre-386 processor implementations, the bit specified the affine (Y = 1) or projective (Y = 0) infinity. The AMD64 architecture uses only the affine infinity, which specifies distinct positive and negative infinity values.

6.2.4 x87 Tag Word Register (FTW)

The x87 tag word register contains a 2-bit tag field for each x87 physical data register. These tag fields characterize the register’s data. Figure 6-5 shows the format of the tag word.

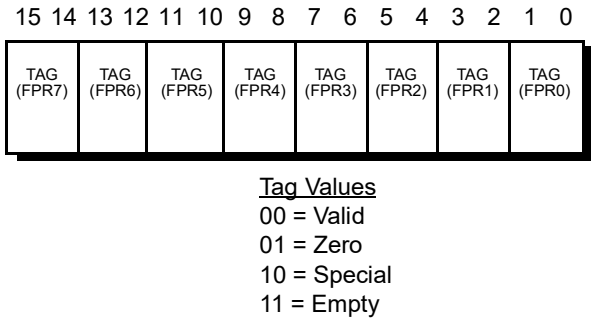


Figure 6-5. x87 Tag Word Register (FTW)

In the memory image saved by the instructions described in "x87 Environment" on page 297, each x87 physical data register has two tag bits which are encoded according to the *Tag Values* shown in Figure 6-5. Internally, the hardware may maintain only a single bit that indicates whether the associated register is *empty* or *full*. The mapping between such a 1-bit internal tag and the 2-bit software-visible architectural representation saved in memory is shown in Table 6-3 on page 295. In such a mapping, whenever software saves the tag word, the processor expands the internal 1-bit tag state to the 2-bit architectural representation by examining the contents of the x87 registers, as described in "SSE, MMX, and x87 Programming" in Volume 2.

Table 6-3. Mapping Between Internal and Software-Visible Tag Bits

Architectural State (Software-Visible)		Hardware State
State	Bit Value	
Valid	00	Full
Zero	01	
Special (NaN, infinity, denormal, or unsupported)	10	
Empty	11	Empty

The FINIT and FNINIT instructions write the tag word so that it specifies all floating-point registers as *empty*. Execution of 64-bit media instructions that write to an MMX™ register alter the tag bits by setting all the registers to *full*, and thus they may affect execution of subsequent x87 floating-point instructions. For details, see "Mixing Media Code with x87 Code" on page 280.

6.2.5 Pointers and Opcode State

The x87 instruction pointer, instruction opcode, and data pointer are part of the x87 environment (non-data processor state) that is loaded and stored by the instructions described in "x87 Environment" on page 297. Figure 6-6 illustrates the pointer and opcode state. Execution of all x87 instructions—except control instructions (see "Control" on page 323)—causes the processor to store this state in hardware.

For convenience, the pointer and opcode state is illustrated here as registers. However, the manner of storing this state in hardware depends on the hardware implementation. The AMD64 architecture specifies only the software-visible state that is saved in memory. (See “Media and x87 Processor State” in Volume 2 for details of the memory images.)

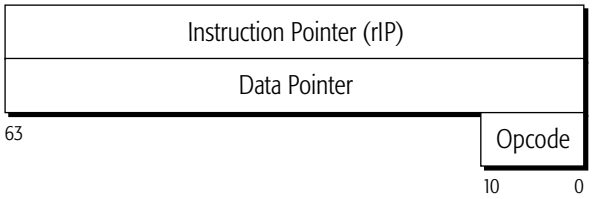


Figure 6-6. x87 Pointers and Opcode State

6.2.5.1 Last x87 Instruction Pointer

The contents of the 64-bit last-instruction pointer depends on the operating mode, as follows:

- *64-Bit Mode*—The pointer contains the 64-bit RIP offset of the last *non-control* x87 instruction executed (see “Control” on page 323 for a definition of control instructions). The 16-bit code-segment (CS) selector is not saved. (It is the operating system’s responsibility to ensure that the 64-bit state-restoration is executed in the same code segment as the preceding 64-bit state-store.)
- *Legacy Protected Mode and Compatibility Mode*—The pointer contains the 16-bit code-segment (CS) selector and the 16-bit or 32-bit eIP of the last non-control x87 instruction executed.
- *Legacy Real Mode and Virtual-8086 Mode*—The pointer contains the 20-bit or 32-bit linear address (CS base + eIP) of the last non-control x87 instruction executed.

The FINIT and FNINIT instructions clear all bits in this pointer.

6.2.5.2 Last x87 Opcode

The 11-bit instruction opcode holds a permutation of the two-byte instruction opcode from the last non-control x87 floating-point instruction executed by the processor. The opcode field is formed as follows:

- Opcode Field[10:8] = First x87-opcode byte[2:0].
- Opcode Field[7:0] = Second x87-opcode byte[7:0].

For example, the x87 opcode D9 F8 (floating-point partial remainder) is stored as 001_1111_1000b. The low-order three bits of the first opcode byte, D9 (1101_1001b), are stored in bits 10:8. The second opcode byte, F8 (1111_1000b), is stored in bits 7:0. The high-order five bits of the first opcode byte (1101_1b) are not needed because they are identical for all x87 instructions.

6.2.5.3 Last x87 Data Pointer

The operating mode determines the value of the 64-bit data pointer, as follows:

- *64-Bit Mode*—The pointer contains the 64-bit offset of the last memory operand accessed by the last non-control x87 instruction executed.
- *Legacy Protected Mode and Compatibility Mode*—The pointer contains the 16-bit data-segment selector and the 16-bit or 32-bit offset of the last memory operand accessed by an executed non-control x87 instruction.
- *Legacy Real Mode and Virtual-8086 Mode*—The pointer contains the 20-bit or 32-bit linear address (segment base + offset) of the last memory operand accessed by an executed non-control x87 instruction.

The FINIT and FNINIT instructions clear all bits in this pointer.

6.2.6 x87 Environment

The x87 environment—or non-data processor state—includes the following processor state:

- x87 control word register (FCW)
- x87 status word register (FSW)
- x87 tag word (FTW)
- last x87 instruction pointer
- last x87 data pointer
- last x87 opcode

Table 6-4 lists the x87 instructions can access this x87 processor state.

Table 6-4. Instructions that Access the x87 Environment

Instruction	Description	State Accessed
FINIT	Floating-Point Initialize	Entire Environment
FNINIT	Floating-Point No-Wait Initialize	Entire Environment
FNSAVE	Floating-Point No-Wait Save State	Entire Environment
FRSTOR	Floating-Point Restore State	Entire Environment
FSAVE	Floating-Point Save State	Entire Environment
FLDCW	Floating-Point Load x87 Control Word	x87 Control Word
FNSTCW	Floating-Point No-Wait Store Control Word	x87 Control Word
FSTCW	Floating-Point Store Control Word	x87 Control Word
FNSTSW	Floating-Point No-Wait Store Status Word	x87 Status Word
FSTSW	Floating-Point Store Status Word	x87 Status Word

Table 6-4. Instructions that Access the x87 Environment (continued)

Instruction	Description	State Accessed
FLDENV	Floating-Point Load x87 Environment	Environment, Not Including x87 Data Registers
FNSTENV	Floating-Point No-Wait Store Environment	Environment, Not Including x87 Data Registers
FSTENV	Floating-Point Store Environment	Environment, Not Including x87 Data Registers

For details on how the x87 environment is stored in memory, see “Media and x87 Processor State” in Volume 2.

6.2.7 Floating-Point Emulation (CR0.EM)

The operating system can set the floating-point software-emulation (EM) bit in control register 0 (CR0) to 1 to allow software emulation of x87 instructions. If the operating system has set CR0.EM = 1, the processor does not execute x87 instructions. Instead, a device-not-available exception (#NM) occurs whenever an attempt is made to execute such an instruction, except that setting CR0.EM to 1 does not cause an #NM exception when the WAIT or FWAIT instruction is executed. For details, see “System-Control Registers” in Volume 2.

6.3 Operands

6.3.1 Operand Addressing

Operands for x87 instructions are referenced by the opcodes. Operands can be located either in x87 registers or memory. Immediate operands are not used in x87 floating-point instructions, and I/O ports cannot be directly addressed by x87 floating-point instructions.

6.3.1.1 Memory Operands

Most x87 floating-point instructions can take source operands from memory, and a few of the instructions can write results to memory. The following sections describe the methods and conditions for addressing memory operands:

- “Memory Addressing” on page 14 describes the general methods and conditions for addressing memory operands.
- “Instruction Prefixes” on page 326 describes the use of address-size instruction overrides by 64-bit media instructions.

6.3.1.2 Register Operands

Most x87 floating-point instructions can read source operands from and write results to x87 registers. Most instructions access the ST(0)–ST(7) register stack. For a few instructions, the register types also include the x87 control word register, the x87 status word register, and (for FSTSW and FNSTSW) the AX general-purpose register.

6.3.2 Data Types

Figure 6-7 shows register images of the x87 data types. These include three scalar floating-point formats (80-bit double-extended-precision, 64-bit double-precision, and 32-bit single-precision), three scalar signed-integer formats (quadword, doubleword, and word), and an 80-bit packed binary-coded decimal (BCD) format. Although Figure 6-7 shows register images of the data types, the three signed-integer data types can exist only in memory. All data types are converted into an 80-bit format when they are loaded into an x87 register.

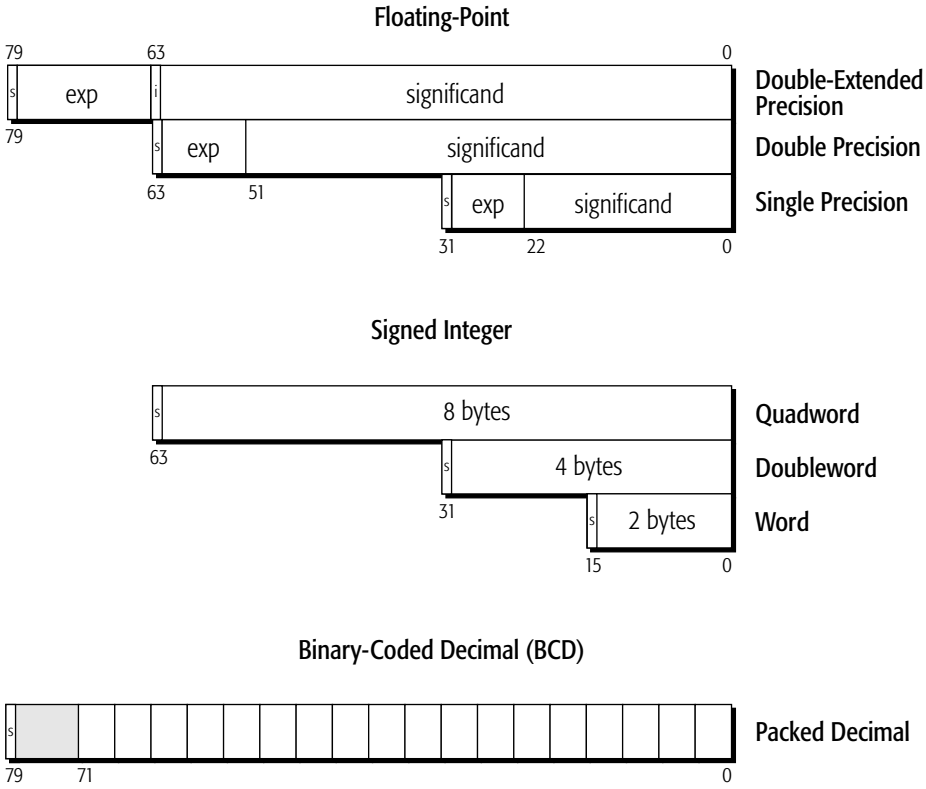


Figure 6-7. x87 Data Types

6.3.2.1 Floating-Point Data Types

The floating-point data types, shown in Figure 6-8 on page 300, include 32-bit single precision, 64-bit double precision, and 80-bit double-extended precision. The default precision is double-extended precision, and all operands loaded into registers are converted into double-extended precision format.

All three floating-point formats are compatible with the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754 and 854).

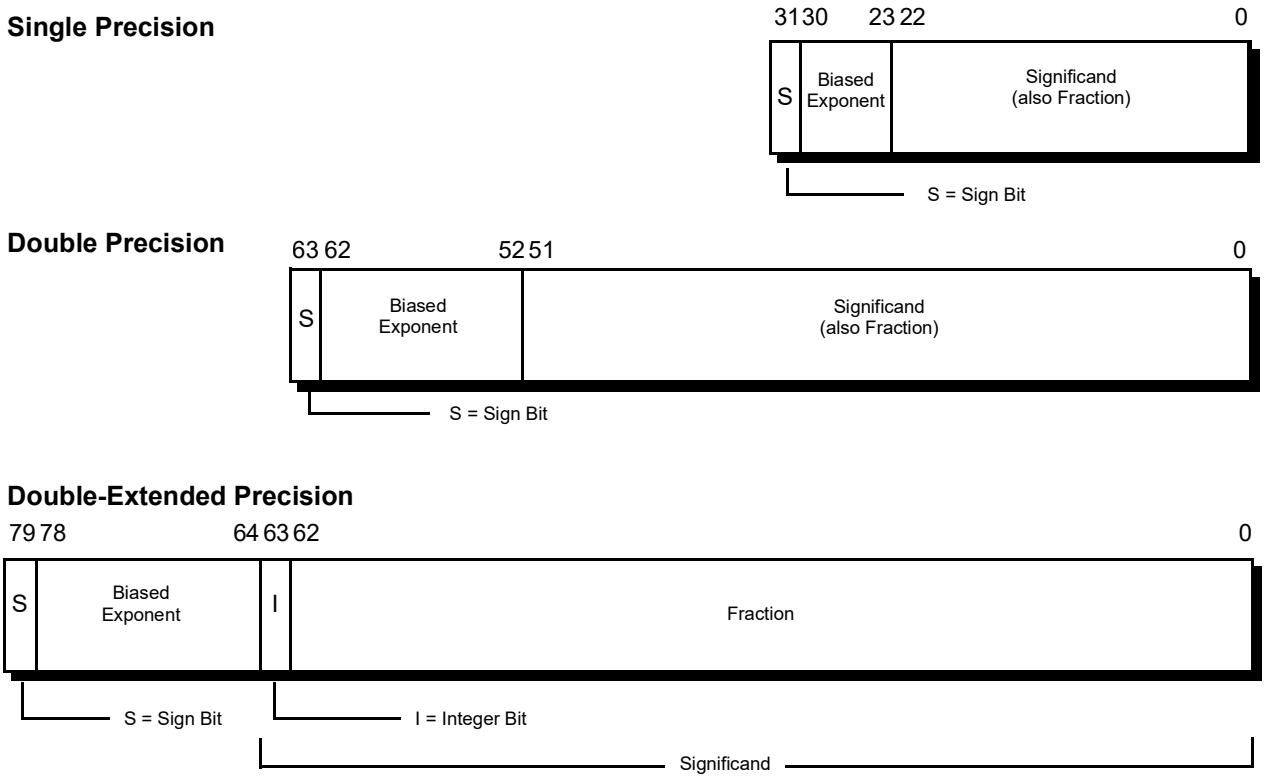


Figure 6-8. x87 Floating-Point Data Types

All of the floating-point data types consist of a sign (0 = positive, 1 = negative), a biased exponent (base-2), and a significand, which represents the integer and fractional parts of the number. The integer bit (also called the *J bit)* is either implied (called a *hidden integer bit*) or explicit, depending on the data type. The value of an implied integer bit can be inferred from number encodings, as described in "Number Encodings" on page 305. The bias of the exponent is a constant which makes the exponent always positive and allows reciprocation, without overflow, of the smallest normalized number representable by that data type.

Specifically, the data types are formatted as follows:

- *Single-Precision Format*—This format includes a 1-bit sign, an 8-bit biased exponent whose value is 127, and a 23-bit significand. The integer bit is implied, making a total of 24 bits in the significand.
- *Double-Precision Format*—This format includes a 1-bit sign, an 11-bit biased exponent whose value is 1023, and a 52-bit significand. The integer bit is implied, making a total of 53 bits in the significand.
- *Double-Extended-Precision Format*—This format includes a 1-bit sign, a 15-bit biased exponent whose value is 16,383, and a 64-bit significand, which includes one explicit integer bit.

Table 6-5 shows the range of finite values representable by the three x87 floating-point data types.

Table 6-5. Range of Finite Floating-Point Values

Data Type	Range of Finite Values[1]		Precision
	Base 2	Base 10	
Single Precision	2^{-126} to $2^{127} * (2 - 2^{-23})$	$1.17 * 10^{-38}$ to $+3.40 * 10^{38}$	24 bits
Double Precision	2^{-1022} to $2^{1023} * (2 - 2^{-52})$	$2.23 * 10^{-308}$ to $+1.79 * 10^{308}$	53 bits
Double-Extended Precision	2^{-16382} to $2^{16383} * (2 - 2^{-63})$	$3.37 * 10^{-4932}$ to $+1.18 * 10^{4932}$	64 bits

Note:
1. See “Number Representation” on page 302.

For example, in the single-precision format, the largest normal number representable has an exponent of FEh and a significand of 7FFFFFh, with a numerical value of $2^{127} * (2 - 2^{-23})$. Results that overflow above the maximum representable value return either the maximum representable normalized number (see “Normalized Numbers” on page 303) or infinity, with the sign of the true result, depending on the rounding mode specified in the rounding control (RC) field of the x87 control word. Results that underflow below the minimum representable value return either the minimum representable normalized number or a denormalized number (see “Denormalized (Tiny) Numbers” on page 303), with the sign of the true result, or a result determined by the x87 exception handler, depending on the rounding mode, precision mode, and underflow-exception mask (UM) in the x87 control word (see “Unmasked Responses” on page 337).

6.3.2.2 Integer Data Type

The integer data types, shown in Figure 6-7 on page 299, include two’s-complement 16-bit word, 32-bit doubleword, and 64-bit quadword. These data types are used in x87 instructions that convert signed integer operands into floating-point values. The integers can be loaded from memory into x87 registers and stored from x87 registers into memory. The data types cannot be moved between x87 registers and other registers.

For details on the format and number-representation of the integer data types, see “Fundamental Data Types” on page 36.

6.3.2.3 Packed-Decimal Data Type

The 80-bit packed-decimal data type, shown in Figure 6-9 on page 302, represents an 18-digit decimal integer using the binary-coded decimal (BCD) format. Each of the 18 digits is a 4-bit representation of an integer. The 18 digits use a total of 72 bits. The next-higher seven bits in the 80-bit format are reserved (ignored on loads, zeros on stores). The high bit (bit 79) is a sign bit.

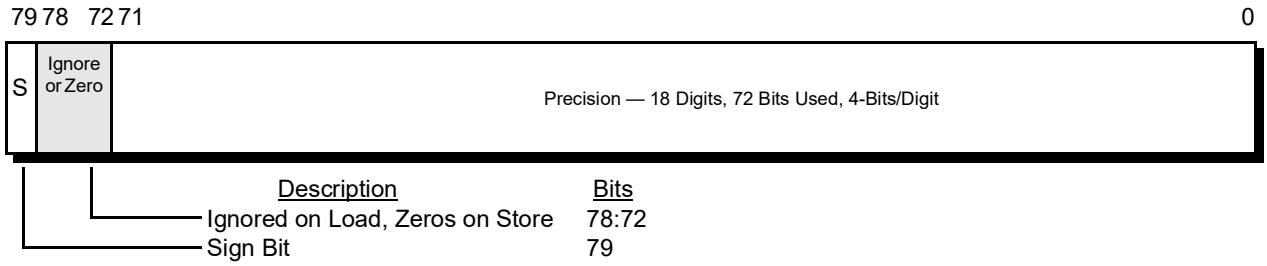


Figure 6-9. x87 Packed Decimal Data Type

Two x87 instructions operate on the packed-decimal data type. The FBLD (floating-point load binary-coded decimal) and FBSTP (floating-point store binary-coded decimal integer and pop) instructions push and pop, respectively, a packed-decimal memory operand between the floating-point stack and memory. FBLD converts the value being pushed to a double-extended-precision floating-point value. FBSTP rounds the value being popped to an integer.

For details on the format and use of 4-bit BCD integers, see “Binary-Coded-Decimal (BCD) Digits” on page 40.

6.3.3 Number Representation

Of the following types of floating-point values, six are supported by the architecture and three are not supported:

- *Supported Values*
  - Normal
  - Denormal (Tiny)
  - Pseudo-Denormal
  - Zero
  - Infinity
  - Not a Number (NaN)
- *Unsupported Values*
  - Unnormal
  - Pseudo-Infinity
  - Pseudo-NaN

The supported values can be used as operands in x87 floating-point instructions. The unsupported values cause an invalid-operation exception (IE) when used as operands.

In common engineering and scientific usage, floating-point numbers—also called *real numbers*—are represented in base (radix) 10. A non-zero number consists of a *sign*, a normalized *significand*, and a signed *exponent*, as in:

```
+2.71828 e0
```

Both large and small numbers are representable in this notation, subject to the limits of data-type precision. For example, a million in base-10 notation appears as +1.00000 e6 and -0.0000383 is represented as -3.83000 e-5. A non-zero number can always be written in *normalized form*—that is, with a leading non-zero digit immediately before the decimal point. Thus, a normalized significand in base-10 notation is a number in the range [1,10). The signed exponent specifies the number of positions that the decimal point is shifted.

Unlike the common engineering and scientific usage described above, x87 floating-point numbers are represented in base (radix) 2. Like its base-10 counterpart, a normalized base-2 significand is written with its leading non-zero digit immediately to the left of the radix point. In base-2 arithmetic, a non-zero digit is always a one, so the range of a binary significand is [1,2):

```
+1.fraction ±exponent
```

The leading non-zero digit is called the *integer bit*, and in the x87 double-extended-precision floating-point format this integer bit is explicit, as shown in Figure 6-8. In the x87 single-precision and the double-precision floating-point formats, the integer bit is simply omitted (and called the *hidden integer bit)*, because its implied value is always 1 in a normalized significand (0 in a denormalized significand), and the omission allows an extra bit of precision.

The following sections describe the supported number representations.

6.3.3.1 Normalized Numbers

Normalized floating-point numbers are the most frequent operands for x87 instructions. These are finite, non-zero, positive or negative numbers in which the integer bit is 1, the biased exponent is non-zero and non-maximum, and the fraction is any representable value. Thus, the significand is within the range of [1, 2). Whenever possible, the processor represents a floating-point result as a normalized number.

6.3.3.2 Denormalized (Tiny) Numbers

Denormalized numbers (also called *tiny* numbers) are smaller than the smallest representable normalized numbers. They arise through an underflow condition, when the exponent of a result lies below the representable minimum exponent. These are finite, non-zero, positive or negative numbers in which the integer bit is 0, the biased exponent is 0, and the fraction is non-zero.

The processor generates a denormalized-operand exception (DE) when an instruction uses a denormalized *source operand*. The processor may generate an underflow exception (UE) when an instruction produces a rounded, non-zero *result* that is too small to be represented as a normalized

floating-point number in the destination format, and thus is represented as a denormalized number. If a result, after rounding, is too small to be represented as the minimum denormalized number, it is represented as zero. (See “Exceptions” on page 327 for specific details.)

Denormalization may correct the exponent by placing leading zeros in the significand. This may cause a loss of precision, because the number of significant bits in the fraction is reduced by the leading zeros. In the single-precision floating-point format, for example, normalized numbers have biased exponents ranging from 1 to 254 (the unbiased exponent range is from –126 to +127). A true result with an exponent of, say, –130, undergoes denormalization by right-shifting the significand by the difference between the normalized exponent and the minimum exponent, as shown in Table 6-6.

Table 6-6. Example of Denormalization

Significand (base 2)	Exponent	Result Type
1.0011010000000000	–130	True result
0.0001001101000000	–126	Denormalized result

6.3.3.3 Pseudo-Denormalized Numbers

Pseudo-denormalized numbers are positive or negative numbers in which the integer bit is 1, the biased exponent is 0, and the fraction is any value. The processor accepts pseudo-denormal source operands but it does not produce pseudo-denormal results. When a pseudo-denormal number is used as a source operand, the processor treats the arithmetic value of its biased exponent as 1 rather than 0, and the processor generates a denormalized-operand exception (DE).

6.3.3.4 Zero

The floating-point zero is a finite, positive or negative number in which the integer bit is 0, the biased exponent is 0, and the fraction is 0. The sign of a zero result depends on the operation being performed and the selected rounding mode. It may indicate the direction from which an underflow occurred, or it may reflect the result of a division by +∞ or –∞.

6.3.3.5 Infinity

Infinity is a positive or negative number, +∞ and –∞, in which the integer bit is 1, the biased exponent is maximum, and the fraction is 0. The infinities are the maximum numbers that can be represented in floating-point format. Negative infinity is less than any finite number and positive infinity is greater than any finite number (i.e., the affine sense).

An infinite result is produced when a non-zero, non-infinite number is divided by 0 or multiplied by infinity, or when infinity is added to infinity or to 0. Arithmetic on infinities is exact. For example, adding any floating-point number to +∞ gives a result of +∞. Arithmetic comparisons work correctly on infinities. Exceptions occur only when the use of an infinity as a source operand constitutes an invalid operation.

6.3.3.6 Not a Number (NaN)

NaNs are non-numbers, lying outside the range of representable floating-point values. The integer bit is 1, the biased exponent is maximum, and the fraction is non-zero. NaNs are of two types:

- *Signaling NaN (SNaN)*
- *Quiet NaN (QNaN)*

A QNaN is a NaN with the most-significant fraction bit set to 1, and an SNaN is a NaN with the most-significant fraction bit cleared to 0. When the processor encounters an SNaN as a source operand for an instruction, an invalid-operation exception (IE) occurs and a QNaN is produced as the result, if the exception is masked. In general, when the processor encounters a QNaN as a source operand for an instruction—in an instruction other than FxCOM*x*, FISTx, or FSTx—the processor does not generate an exception but generates a QNaN as the result.

The processor never generates an SNaN as a result of a floating-point operation. When an invalid-operation exception (IE) occurs due to an SNaN operand, the invalid-operation exception mask (IM) bit determines the processor's response, as described in "x87 Floating-Point Exception Masking" on page 333.

When a floating-point operation or exception produces a QNaN result, its value is derived from the source operands according to the rules shown in Table 6-7.

6.3.4 Number Encodings

6.3.4.1 Supported Encodings

Table 6-8 on page 307 shows the floating-point encodings of supported numbers and non-numbers. The number categories are ordered from large to small. In this affine ordering, positive infinity is larger than any positive normalized number, which in turn is larger than any positive denormalized number, which is larger than positive zero, and so forth. Thus, the ordinary rules of comparison apply between categories as well as within categories, so that comparison of any two numbers is well-defined.

The actual exponent field length is 8, 11, or 15 bits, and the fraction field length is 23, 52, or 63 bits, depending on operand precision.

Table 6-7. NaN Results from NaN Source Operands

Source Operand (in either order)[1]		NaN Result[2]
QNaN	Any non-NaN floating-point value (or single-operand instruction)	Value of QNaN
SNaN	Any non-NaN floating-point value (or single-operand instruction)	Value of SNaN, converted to a QNaN[3]
QNaN	QNaN	Value of QNaN with the larger significand[4]
QNaN	SNaN	Value of QNaN
SNaN	QNaN	Value of QNaN
SNaN	SNaN	Value of SNaN with the larger significand[4]

Note:

1. *This table does not include NaN source operands used in FxCOMx, FISTx, or FSTx instructions.*
2. *A NaN result is produced when the floating-point invalid-operation exception is masked.*
3. *The conversion is done by changing the most-significant fraction bit to 1.*
4. *If the significands of the source operands are equal but their signs are different, the NaN result is undefined.*

The single-precision and double-precision formats do not include the integer bit in the significand (the value of the integer bit can be inferred from number encodings). The double-extended-precision format explicitly includes the integer in bit 63 and places the most-significant fraction bit in bit 62. Exponents of all three types are encoded in biased format, with respective biasing constants of 127, 1023, and 16,383.

Table 6-8. Supported Floating-Point Encodings

Classification		Sign	Biased Exponent[1]	Significand[2]
Positive Non-Numbers	SNaN	0	111 ... 111	1.011 ... 111 to 1.000 ... 001
	QNaN	0	111 ... 111	1.111 ... 111 to 1.100 ... 000
Positive Floating-Point Numbers	Positive Infinity (+∞)	0	111 ... 111	1.000 ... 000
	Positive Normal	0	111 ... 110 to 000 ... 001	1.111 ... 111 to 1.000 ... 000
	Positive Pseudo-Denormal[3]	0	000 ... 000	1.111 ... 111 to 1.000 ... 001
	Positive Denormal	0	000 ... 000	0.111 ... 111 to 0.000 ... 001
	Positive Zero	0	000 ... 000	0.000 ... 000
Negative Floating-Point Numbers	Negative Zero	1	000 ... 000	0.000 ... 000
	Negative Denormal	1	000 ... 000	0.000 ... 001 to 0.111 ... 111
	Negative Pseudo-Denormal[3]	1	000 ... 000	1.000 ... 001 to 1.111 ... 111
	Negative Normal	1	000 ... 001 to 111 ... 110	1.000 ... 000 to 1.111 ... 111
	Negative Infinity (-∞)	1	111 ... 111	1.000 ... 000
Negative Non-Numbers	SNaN	1	111 ... 111	1.000 ... 001 to 1.011 ... 111
	QNaN[4]	1	111 ... 111	1.100 ... 000 to 1.111 ... 111

Note:

1. *The actual exponent field length is 8, 11, or 15 bits, depending on operand precision.*
2. *The "1." and "0." prefixes represent the implicit or explicit integer bit. The actual fraction field length is 23, 52, or 63 bits, depending on operand precision.*
3. *Pseudo-denormals can only occur in double-extended-precision format, because they require an explicit integer bit.*
4. *The floating-point indefinite value is a QNaN with a negative sign and a significand whose value is 1.100 ... 000.*

6.3.4.2 Unsupported Encodings

Table 6-9 on page 308 shows the encodings of unsupported values. These values can exist only in the double-extended-precision format, because they require an explicit integer bit. The processor does not generate them as results, and they cause an invalid-operation exception (IE) when used as source operands.

6.3.4.3 Indefinite Values

Floating-point, integer, and packed-decimal data types each have a unique encoding that represents an *indefinite value*. The processor returns an indefinite value when a masked invalid-operation exception (IE) occurs. The indefinite values for various data types are provided in Table 4-7 on page 129.

For example, if a floating-point arithmetic operation is attempted using a source operand which is in an unsupported format, and IE exceptions are masked, the floating-point indefinite value is returned as the result. Or, if an integer store instruction overflows its destination data type, and IE exceptions are masked, the integer indefinite value is returned as the result.

Table 6-9. Unsupported Floating-Point Encodings

Classification	Sign	Biased Exponent[1]	Significand[2]
Positive Pseudo-NaN	0	111 ... 111	0.111 ... 111 to 0.000 ... 001
Positive Pseudo-Infinity	0	111 ... 111	0.000 ... 000
Positive Unnormal	0	111 ... 110 to 000 ... 001	0.111 ... 111 to 0.000 ... 000
Negative Unnormal	1	000 ... 001 to 111 ... 110	0.000 ... 000 to 0.111 ... 111
Negative Pseudo-Infinity	1	111 ... 111	0.000 ... 000
Negative Pseudo-NaN	1	111 ... 111	0.000 ... 001 to 0.111 ... 111

Note:
1. *The actual exponent field length is 15 bits.*
2. *The "0." prefix represent the explicit integer bit. The actual fraction field length is 63 bits.*

Table 6-10 shows the encodings of the indefinite values for each data type. For floating-point numbers, the indefinite value is a special form of QNaN. For integers, the indefinite value is the largest representable negative two's-complement number, 80...00h. (This value is interpreted as the largest representable negative number, except when a masked IE exception occurs, in which case it is interpreted as an indefinite value.) For packed-decimal numbers, the indefinite value has no other meaning than indefinite.

Table 6-10. Indefinite-Value Encodings

Data Type	Indefinite Encoding
Single-Precision Floating-Point	FFC0_0000h
Double-Precision Floating-Point	FFF8_0000_0000_0000h
Extended-Precision Floating-Point	FFFF_C000_0000_0000_0000h
16-Bit Integer	8000h
32-Bit Integer	8000_0000h
64-Bit Integer	8000_0000_0000_0000h
80-Bit BCD	FFFF_C000_0000_0000_0000h

6.3.5 Precision

The Precision control (PC) field comprises bits [9:8] of the x87 control word ("x87 Control Word Register (FCW)" on page 292). This field specifies the precision of floating-point calculations for the FADDx, FSUB*x*, FMUL*x*, FDIV*x*, and FSQRT instructions, as shown in Table 6-11.

Table 6-11. Precision Control Field (PC) Values and Bit Precision

PC Field	Data Type	Precision (bits)
00	Single precision	24[1]
01	*reserved*	
10	Double precision	53[1]
11	Double-extended precision	64

Note:
1. The single-precision and double-precision bit counts include the implied integer bit.

The default precision is double-extended-precision. Selecting double-precision or single-precision reduces the size of the significand to 53 bits or 24 bits, but keeps the exponent in double extended range. The reduced precision is provided to support the IEEE 754 standard. When using reduced precision, rounding clears the unused bits on the right of the significand to 0s.

6.3.6 Rounding

The rounding control (RC) field comprises bits [11:10] of the x87 control word ("x87 Control Word Register (FCW)" on page 292). This field specifies how the results of x87 floating-point computations are rounded. Rounding modes apply to most arithmetic operations but not to comparison or remainder. They have no effect on operations that produce NaN results.

The IEEE 754 standard defines the four rounding modes as shown in Table 6-12.

Table 6-12. Types of Rounding

RC Value	Mode	Type of Rounding
00 (default)	Round to nearest	The rounded result is the representable value closest to the infinitely precise result. If equally close, the even value (with least-significant bit 0) is taken.
01	Round down	The rounded result is closest to, but no greater than, the infinitely precise result.
10	Round up	The rounded result is closest to, but no less than, the infinitely precise result.
11	Round toward zero	The rounded result is closest to, but no greater in absolute value than, the infinitely precise result.

Round to nearest is the default (reset) rounding mode. It provides a statistically unbiased estimate of the true result, and is suitable for most applications. The other rounding modes are directed roundings: round up (toward +∞), round down (toward –∞), and round toward zero. Round up and round down are used in interval arithmetic, in which upper and lower bounds bracket the true result of a computation. Round toward zero takes the smaller in magnitude, that is, always truncates.

The processor produces a floating-point result defined by the IEEE standard to be infinitely precise. This result may not be representable exactly in the destination format, because only a subset of the continuum of real numbers finds exact representation in any particular floating-point format. Rounding modifies such a result to conform to the destination format, thereby making the result inexact and also generating a precision exception (PE), as described in “x87 Floating-Point Exception Causes” on page 329.

Suppose, for example, the following 24-bit result is to be represented in single-precision format, where “E_2 1010” represents the biased exponent:

$$1.0011\ 0101\ 0000\ 0001\ 0010\ 0111\ E_2\ 1010$$

This result has no exact representation, because the least-significant 1 does not fit into the single-precision format, which allows for only 23 bits of fraction. The rounding control field determines the direction of rounding. Rounding introduces an error in a result that is less than one *unit in the last place (ulp)*, that is, the least-significant bit position of the floating-point representation.

6.4 Instruction Summary

This section summarizes the functions of the x87 floating-point instructions. The instructions are organized here by functional group—such as data-transfer, arithmetic, and so on. More detail on individual instructions is given in the alphabetically organized “x87 Floating-Point Instruction Reference” in Volume 5.

Software running at any privilege level can use any of these instructions, if the CPUID instruction reports support for the instructions (see “Feature Detection” on page 327). Most x87 instructions take

floating-point data types for both their source and destination operands, although some x87 data-conversion instructions take integer formats for their source or destination operands.

6.4.1 Syntax

Each instruction has a *mnemonic syntax* used by assemblers to specify the operation and the operands to be used for source and destination (result) data. Many of x87 instructions have the following syntax:

```
MNEMONIC st(j), st(i)
```

Figure 6-10 on page 311 shows an example of the mnemonic syntax for a floating-point add (FADD) instruction.

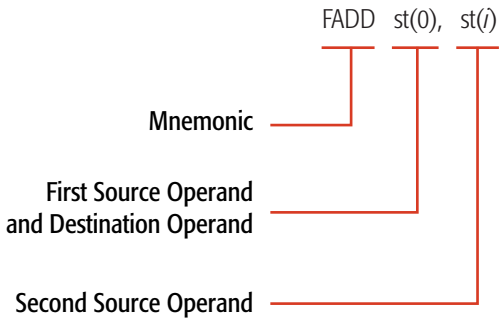


Figure 6-10. Mnemonic Syntax for Typical Instruction

This example shows the FADD mnemonic followed by two operands, both of which are 80-bit stack-register operands. Most instructions take source operands from an x87 stack register and/or memory and write their results to a stack register or memory. Only two of the instructions (FSTSW and FNSTSW) can access a general-purpose registers (GPR), and none access the 128-bit media (XMM) registers. Although the MMX registers map to the x87 registers, the contents of the MMX registers cannot be accessed meaningfully using x87 instructions.

Instructions can have one or more prefixes that modify default operand properties. These prefixes are summarized in “Instruction Prefixes” on page 76.

6.4.1.1 Mnemonics

The following characters are used as prefixes in the mnemonics of integer instructions:

- **F**—x87 Floating-point

In addition to the above prefix characters, the following characters are used elsewhere in the mnemonics of x87 instructions:

- **B**—Below, or BCD
- **BE**—Below or Equal

- **CMOV**—Conditional Move
- **c**—Variable condition
- **E**—Equal
- **I**—Integer
- **LD**—Load
- **N**—No Wait
- **NB**—Not Below
- **NBE**—Not Below or Equal
- **NE**—Not Equal
- **NU**—Not Unordered
- **P**—Pop
- **PP**—Pop Twice
- **R**—Reverse
- **ST**—Store
- **U**—Unordered
- ***x***—One or more variable characters in the mnemonic

For example, the mnemonic for the store instruction that stores the top-of-stack and pops the stack is FSTP. In this mnemonic, the F means a floating-point instruction, the ST means a store, and the P means pop the stack.

6.4.2 Data Transfer and Conversion

The data transfer and conversion instructions copy data—in some cases with data conversion—between x87 stack registers and memory or between stack positions.

Load or Store Floating-Point

- FLD—Floating-Point Load
- FST—Floating-Point Store Stack Top
- FSTP—Floating-Point Store Stack Top and Pop

The FLD instruction pushes the source operand onto the top-of-stack, ST(0). The source operand may be a single-precision, double-precision, or double-extended-precision floating-point value in memory or the contents of a specified stack position, ST(i).

The FST instruction copies the value at the top-of-stack, ST(0), to a specified stack position, ST(i), or to a 32-bit or 64-bit memory location. If the destination is a memory location, the value copied is converted to the precision allowed by the destination and rounded, as specified by the rounding control (RC) field of the x87 control word. If the top-of-stack value is a NaN or an infinity, FST truncates the stack-top exponent and significand to fit the destination size. (For details, see “FST FSTP” in *AMD64*

Architecture Programmer's Manual Volume 5: 64-bit Media and x87 Floating-Point Instructions, order# 26569.

The FSTP instruction is similar to FST, except that FSTP can also store to an 80-bit memory location and it pops the stack after the store. FSTP can be used to clean up the x87 stack at the end of an x87 procedure by removing one register of preloaded data from the stack.

Convert and Load or Store Integer

- FILD—Floating-Point Load Integer
- FIST—Floating-Point Integer Store
- FISTP—Floating-Point Integer Store and Pop
- FISTTP—Floating-Point Integer Truncate and Store

The FILD instruction converts the 16-bit, 32-bit, or 64-bit source signed integer in memory into a double-extended-precision floating-point value and pushes the result onto the top-of-stack, ST(0).

The FIST instruction converts and rounds the source value in the top-of-stack, ST(0), to a signed integer and copies it to the specified 16-bit or 32-bit memory location. The type of rounding is determined by the rounding control (RC) field of the x87 control word.

The FISTP instruction is similar to FIST, except that FISTP can also store the result to a 64-bit memory location and it pops ST(0) after the store.

The FISTTP instruction converts a floating-point value in ST(0) to an integer by truncating the fractional part of the number and storing the integer result to the memory address specified by the destination operand. FISTTP then pops the floating point register stack. The FISTTP instruction ignores the rounding mode specified by the x87 control word.

Convert and Load or Store BCD

- FBLD—Floating-Point Load Binary-Coded Decimal
- FBSTP—Floating-Point Store Binary-Coded Decimal Integer and Pop

The FBLD and FBSTP instructions, respectively, push and pop an 80-bit packed BCD memory value on and off the top-of-stack, ST(0). FBLD first converts the value being pushed to a double-extended-precision floating-point value. FBSTP rounds the value being popped to an integer, using the rounding mode specified by the RC field, and converts the value to an 80-bit packed BCD value. Thus, no FRNDIT (round-to-integer) instruction is needed prior to FBSTP.

Conditional Move

- FCMOVB—Floating-Point Conditional Move If Below
- FCMOVBE—Floating-Point Conditional Move If Below or Equal
- FCMOVE—Floating-Point Conditional Move If Equal
- FCMOVNB—Floating-Point Conditional Move If Not Below
- FCMOVNBE—Floating-Point Conditional Move If Not Below or Equal

- FCMOVNE—Floating-Point Conditional Move If Not Equal
- FCMOVNU—Floating-Point Conditional Move If Not Unordered
- FCMOVU—Floating-Point Conditional Move If Unordered

The FCMOV*cc* instructions copy the contents of a specified stack position, ST(i), to the top-of-stack, ST(0), if the specified rFLAGS condition is met. Table 6-13 on page 314 specifies the flag combinations for each conditional move.

Table 6-13. rFLAGS Conditions for FCMOV*cc*

Condition	Mnemonic	rFLAGS Register State
Below	B	Carry flag is set (CF = 1)
Below or Equal	BE	Either carry flag or zero flag is set (CF = 1 or ZF = 1)
Equal	E	Zero flag is set (ZF = 1)
Not Below	NB	Carry flag is not set (CF = 0)
Not Below or Equal	NBE	Neither carry flag nor zero flag is set (CF = 0, ZF = 0)
Not Equal	NE	Zero flag is not set (ZF = 0)
Not Unordered	NU	Parity flag is not set (PF = 0)
Unordered	U	Parity flag is set (PF = 1)

Exchange

- FXCH—Floating-Point Exchange

The FXCH instruction exchanges the contents of a specified stack position, ST(i), with the top-of-stack, ST(0). The top-of-stack pointer is left unchanged. In the form of the instruction that specifies no operand, the contents of ST(1) and ST(0) are exchanged.

Extract

- FXTRACT—Floating-Point Extract Exponent and Significand

The FXTRACT instruction copies the unbiased exponent of the original value in the top-of-stack, ST(0), and writes it as a floating-point value to ST(1), then copies the significand and sign of the original value in the top-of-stack and writes it as a floating-point value with an exponent of zero to the top-of-stack, ST(0).

6.4.3 Load Constants

Load 0, 1, or Pi

- FLDZ—Floating-Point Load +0.0
- FLD1—Floating-Point Load +1.0
- FLDPI—Floating-Point Load Pi

The FLDZ, FLD1, and FLDPI instructions, respectively, push the floating-point constant value, +0.0, +1.0, and Pi (3.141592653...), onto the top-of-stack, ST(0).

Load Logarithm

- FLDL2E—Floating-Point Load Log2 e
- FLDL2T—Floating-Point Load Log2 10
- FLDLG2—Floating-Point Load Log10 2
- FLDLN2—Floating-Point Load Ln 2

The FLDL2E, FLDL2T, FLDLG2, and FLDLN2 instructions, respectively, push the floating-point constant value, $\log_2 e$, $\log_2 10$, $\log_{10} 2$, and $\log_e 2$, onto the top-of-stack, ST(0).

6.4.4 Arithmetic

The arithmetic instructions support addition, subtraction, multiplication, division, change-sign, round, round to integer, partial remainder, and square root. In most arithmetic operations, one of the source operands is the top-of-stack, ST(0). The other source operand can be another stack entry, ST(*i*), or a floating-point or integer operand in memory.

The non-commutative operations of subtraction and division have two forms, the direct FSUB and FDIV, and the reverse FSUBR and FDIVR. FSUB, for example, subtracts the right operand from the left operand, and writes the result to the left operand. FSUBR subtracts the left operand from the right operand, and writes the result to the left operand. The FADD and FMUL operations have no reverse counterparts.

Addition

- FADD—Floating-Point Add
- FADDP—Floating-Point Add and Pop
- FIADD—Floating-Point Add Integer to Stack Top

The FADD instruction syntax has forms that include one or two explicit source operands. In the one-operand form, the instruction reads a 32-bit or 64-bit floating-point value from memory, converts it to the double-extended-precision format, adds it to ST(0), and writes the result to ST(0). In the two-operand form, the instruction adds both source operands from stack registers and writes the result to the first operand.

The FADDP instruction syntax has forms that include zero or two explicit source operands. In the zero-operand form, the instruction adds ST(0) to ST(1), writes the result to ST(1), and pops the stack. In the two-operand form, the instruction adds both source operands from stack registers, writes the result to the first operand, and pops the stack.

The FIADD instruction reads a 16-bit or 32-bit integer value from memory, converts it to the double-extended-precision format, adds it to ST(0), and writes the result to ST(0).

Subtraction

- FSUB—Floating-Point Subtract
- FSUBP—Floating-Point Subtract and Pop
- FISUB—Floating-Point Integer Subtract
- FSUBR—Floating-Point Subtract Reverse
- FSUBRP—Floating-Point Subtract Reverse and Pop
- FISUBR—Floating-Point Integer Subtract Reverse

The FSUB instruction syntax has forms that include one or two explicit source operands. In the one-operand form, the instruction reads a 32-bit or 64-bit floating-point value from memory, converts it to the double-extended-precision format, subtracts it from ST(0), and writes the result to ST(0). In the two-operand form, both source operands are located in stack registers. The instruction subtracts the second operand from the first operand and writes the result to the first operand.

The FSUBP instruction syntax has forms that include zero or two explicit source operands. In the zero-operand form, the instruction subtracts ST(0) from ST(1), writes the result to ST(1), and pops the stack. In the two-operand form, both source operands are located in stack registers. The instruction subtracts the second operand from the first operand, writes the result to the first operand, and pops the stack.

The FISUB instruction reads a 16-bit or 32-bit integer value from memory, converts it to the double-extended-precision format, subtracts it from ST(0), and writes the result to ST(0).

The FSUBR and FSUBRP instructions perform the same operations as FSUB and FSUBP, respectively, except that the source operands are reversed. Instead of subtracting the second operand from the first operand, FSUBR and FSUBRP subtract the first operand from the second operand.

Multiplication

- FMUL—Floating-Point Multiply
- FMULP—Floating-Point Multiply and Pop
- FIMUL—Floating-Point Integer Multiply

The FMUL instruction has three forms. One form of the instruction multiplies two double-extended precision floating-point values located in ST(0) and another floating-point stack register and leaves the product in ST(0). The second form multiplies two double-extended precision floating-point values located in ST(0) and another floating-point stack destination register and leaves the product in the destination register. The third form converts a floating-point value in a specified memory location to double-extended-precision format, multiplies the result by the value in ST(0) and writes the product to ST(0).

The FMULP instruction syntax is similar to the form of FMUL described in the previous paragraph. This instruction pops the floating-point register stack after performing the multiplication operation. This instruction cannot take a memory operand.

The FIMUL instruction reads a 16-bit or 32-bit integer value from memory, converts it to the double-extended-precision format, multiplies ST(0) by the memory operand, and writes the result to ST(0).

Division

- FDIV—Floating-Point Divide
- FDIVP—Floating-Point Divide and Pop
- FIDIV—Floating-Point Integer Divide
- FDIVR—Floating-Point Divide Reverse
- FDIVRP—Floating-Point Divide Reverse and Pop
- FIDIVR—Floating-Point Integer Divide Reverse

The FDIV instruction syntax has forms that include one or two source explicit operands that may be single-precision or double-precision floating-point values or 16-bit or 32-bit integer values. In the one-operand form, the instruction reads a value from memory, divides ST(0) by the memory operand, and writes the result to ST(0). In the two-operand form, both source operands are located in stack registers. The instruction divides the first operand by the second operand and writes the result to the first operand.

The FDIVP instruction syntax has forms that include zero or two explicit source operands. In the zero-operand form, the instruction divides ST(1) by ST(0), writes the result to ST(1), and pops the stack. In the two-operand form, both source operands are located in stack registers. The instruction divides the first operand by the second operand, writes the result to the first operand, and pops the stack.

The FIDIV instruction reads a 16-bit or 32-bit integer value from memory, converts it to the double-extended-precision format, divides ST(0) by the memory operand, and writes the result to ST(0).

The FDIVR and FDIVRP instructions perform the same operations as FDIV and FDIVP, respectively, except that the source operands are reversed. Instead of dividing the first operand by the second operand, FDIVR and FDIVRP divide the second operand by the first operand.

Change Sign

- FABS—Floating-Point Absolute Value
- FCHS—Floating-Point Change Sign

The FABS instruction changes the top-of-stack value, ST(0), to its absolute value by clearing its sign bit to 0. The top-of-stack value is always positive following execution of the FABS instruction. The FCHS instruction complements the sign bit of ST(0). For example, if ST(0) was +0.0 before the execution of FCHS, it is changed to -0.0.

Round

- FRNDINT—Floating-Point Round to Integer

The FRNDINT instruction rounds the top-of-stack value, ST(0), to an integer value, although the value remains in double-extended-precision floating-point format. Rounding takes place according to the setting of the rounding control (RC) field in the x87 control word.

Partial Remainder

- FPREM—Floating-Point Partial Remainder
- FPREM1—Floating-Point Partial Remainder

The FPREM instruction returns the remainder obtained by dividing ST(0) by ST(1) and stores it in ST(0). If the exponent difference between ST(0) and ST(1) is less than 64, all integer bits of the quotient are calculated, guaranteeing that the remainder returned is less in magnitude than the divisor in ST(1). If the exponent difference is equal to or greater than 64, only a subset of the integer quotient bits, numbering between 32 and 63, are calculated and a partial remainder is returned. FPREM can be repeated on a partial remainder until reduction is complete. It can be used to bring the operands of transcendental functions into their proper range. FPREM is supported for software written for early x87 coprocessors. Unlike the FPREM1 instruction, FPREM does not calculate the partial remainder as specified in IEEE Standard 754.

The FPREM1 instruction works like FPREM, except that the FPREM1 quotient is rounded using round-to-nearest mode, whereas FPREM truncates the quotient.

Square Root

- FSQRT—Floating-Point Square Root

The FSQRT instruction replaces the contents of the top-of-stack, ST(0), with its square root.

6.4.5 Transcendental Functions

The transcendental instructions compute trigonometric functions, inverse trigonometric functions, logarithmic functions, and exponential functions.

Trigonometric Functions

- FSIN—Floating-Point Sine
- FCOS—Floating-Point Cosine
- FSINCOS—Floating-Point Sine and Cosine
- FPTAN—Floating-Point Partial Tangent
- FPATAN—Floating-Point Partial Arctangent

The FSIN instruction replaces the contents of ST(0) (in radians) with its sine.

The FCOS instruction replaces the contents of ST(0) (in radians) with its cosine.

The FSINCOS instruction computes both the sine and cosine of the contents of ST(0) (in radians) and writes the sine to ST(0) and pushes the cosine onto the stack. Frequently, a piece of code that needs to compute the sine of an argument also needs to compute the cosine of that same argument. In such

cases, use the FSINCOS instruction to compute both functions concurrently, which is faster than using separate FSIN and FCOS instructions.

The FPTAN instruction replaces the contents of the ST(0) (in radians), with its tangent, in radians, and pushes the value 1.0 onto the stack.

The FPATAN instruction computes θ = arctan (Y/X), in which X is located in ST(0) and Y in ST(1). The result, θ, is written over Y in ST(1), and the stack is popped.

FSIN, FCOS, FSINCOS, and FPTAN are architecturally restricted in their argument range. Only arguments with a magnitude of less than 2^{63} can be evaluated. If the argument is out of range, the C2 condition-code bit in the x87 status word is set to 1, and the argument is returned as the result. If software detects an out-of-range argument, the FPREM or FPREM1 instruction can be used to reduce the magnitude of the argument before using the FSIN, FCOS, FSINCOS, or FPTAN instruction again.

Logarithmic Functions

- F2XM1—Floating-Point Compute 2^x-1
- FSCALE—Floating-Point Scale
- FYL2X—Floating-Point y * log2x
- FYL2XP1—Floating-Point y * log2(x +1)

The F2XM1 instruction computes $Y = 2^X - 1$. X is located in ST(0) and must fall between –1 and +1. Y replaces X in ST(0). If ST(0) is out of range, the instruction returns an undefined result but no x87 status-word exception bits are affected.

The FSCALE instruction replaces ST(0) with ST(0) times 2^n, where n is the value in ST(1) truncated to an integer. This provides a fast method of multiplying by integral powers of 2.

The FYL2X instruction computes $Z = Y * \log_2 X$. X is located in ST(0) and Y is located in ST(1). X must be greater than 0. The result, Z, replaces Y in ST(1), which becomes the new top-of-stack because X is popped off the stack.

The FYL2XP1 instruction computes $Z = Y * \log_2(X + 1)$. X located in ST(0) and must be in the range $0 < |X| < (1 - 2^{½} / 2)$. Y is taken from ST(1). The result, Z, replaces Y in ST(1), which becomes the new top-of-stack because X is popped off the stack.

6.4.5.1 Accuracy of Transcendental Results

x87 computations are carried out in double-extended-precision format, so that the transcendental functions provide results accurate to within one *unit in the last place (ulp)* for each of the floating-point data types.

6.4.5.2 Argument Reduction Using Pi

The FPREM and FPREM1 instructions can be used to reduce an argument of a trigonometric function by a multiple of Pi. The following example shows a reduction by 2π:

```
sin(n*2π+x) = sin(x) for all integral n
```

In this example, the range is $0 \le x < 2\pi$ in the case of FPREM or $-\pi \le x \le \pi$ in the case of FPREM1. Negative arguments are reduced by repeatedly subtracting -2π. See “Partial Remainder” on page 318 for details of the instructions.

6.4.6 Compare and Test

The compare-and-test instructions set and clear flags in the rFLAGS register to indicate the relationship between two operands (less, equal, greater, or unordered).

Floating-Point Ordered Compare

- FCOM—Floating-Point Compare
- FCOMP—Floating-Point Compare and Pop
- FCOMPP—Floating-Point Compare and Pop Twice
- FCOMI—Floating-Point Compare and Set Flags
- FCOMIP—Floating-Point Compare and Set Flags and Pop

The FCOM instruction syntax has forms that include zero or one explicit source operands. In the zero-operand form, the instruction compares ST(1) with ST(0) and writes the x87 status-word condition codes accordingly. In the one-operand form, the instruction reads a 32-bit or 64-bit value from memory, compares it with ST(0), and writes the x87 condition codes accordingly.

The FCOMP instruction performs the same operation as FCOM but also pops ST(0) after writing the condition codes.

The FCOMPP instruction performs the same operation as FCOM but also pops both ST(0) and ST(1). FCOMPP can be used to initialize the x87 stack at the end of an x87 procedure by removing two registers of preloaded data from the stack.

The FCOMI instruction compares the contents of ST(0) with the contents of another stack register and writes the ZF, PF, and CF flags in the rFLAGS register as shown in Table 6-14. If no source is specified, ST(0) is compared to ST(1). If ST(0) or the source operand is a NaN or in an unsupported format, the flags are set to indicate an unordered condition.

The FCOMIP instruction performs the same comparison as FCOMI but also pops ST(0) after writing the rFLAGS bits.

Table 6-14. rFLAGS Values for FCOMI Instruction

Flag	ST(0) > ST(i)	ST(0) < ST(i)	ST(0) = ST(i)	Unordered
ZF	0	0	1	1
PF	0	0	0	1
CF	0	1	0	1

For comparison-based branches, the combination of FCOMI and FCMOV*cc* is faster than the classical method of using F*x*STSW AX to copy condition codes through the AX register to the rFLAGS register, where they can provide branch direction for conditional operations.

The FCOM*x* instructions perform ordered compares, as opposed to the FUCOM*x* instructions. See the description of ordered vs. unordered compares immediately below.

Floating-Point Unordered Compare

- FUCOM—Floating-Point Unordered Compare
- FUCOMP—Floating-Point Unordered Compare and Pop
- FUCOMPP—Floating-Point Unordered Compare and Pop Twice
- FUCOMI—Floating-Point Unordered Compare and Set Flags
- FUCOMIP—Floating-Point Unordered Compare and Set Flags and Pop

The FUCOM*x* instructions perform the same operations as the FCOM*x* instructions, except that the FUCOM*x* instructions generate an invalid-operation exception (IE) only if any operand is an unsupported data type or a signaling NaN (SNaN), whereas the ordered-compare FCOM*x* instructions generate an invalid-operation exception if any operand is an unsupported data type or any type of NaN. For a description of NaNs, see “Number Representation” on page 302.

Integer Compare

- FICOM—Floating-Point Integer Compare
- FICOMP—Floating-Point Integer Compare and Pop

The FICOM instruction reads a 16-bit or 32-bit integer value from memory, compares it with ST(0), and writes the condition codes in the same way as the FCOM instruction.

The FICOMP instruction performs the same operations as FICOM but also pops ST(0).

Test

- FTST—Floating-Point Test with Zero

The FTST instruction compares ST(0) with zero and writes the condition codes in the same way as the FCOM instruction.

Classify

- FXAM—Floating-Point Examine

The FXAM instruction determines the type of value in ST(0) and sets the condition codes accordingly, as shown in Table 6-15.

Table 6-15. Condition-Code Settings for FXAM

C3	C2	C0	C1[1]	Meaning
0	0	0	0	+unsupported
0	0	0	1	-unsupported
0	0	1	0	+NAN
0	0	1	1	-NAN
0	1	0	0	+normal
0	1	0	1	-normal
0	1	1	0	+infinity
0	1	1	1	-infinity
1	0	0	0	+0
1	0	0	1	-0
1	0	1	0	+empty
1	0	1	1	-empty
1	1	0	0	+denormal
1	1	0	1	-denormal

Note:
1. C1 is the sign of ST(0).

6.4.7 Stack Management

The stack management instructions move the x87 top-of-stack pointer (TOP) and clear the contents of stack registers.

Stack Control

- FDECSTP—Floating-Point Decrement Stack-Top Pointer
- FINCSTP—Floating-Point Increment Stack-Top Pointer

The FINCSTP and FDECSTP instructions increment and decrement, respectively, the TOP, modulo-8. Neither the x87 tag word nor the contents of the floating-point stack itself is updated.

Clear State

- FFREE—Free Floating-Point Register

The FFREE instruction frees a specified stack register by setting the x87 tag-word bits for the register to all 1s, indicating *empty*. Neither the stack-register contents nor the stack pointer is modified by this instruction.

6.4.8 No Operation

This instruction uses processor cycles but generates no result.

- FNOP—Floating-Point No Operation

The FNOP instruction has no operands and writes no result. Its purpose is simply to delay execution of a sequence of instructions.

6.4.9 Control

The control instructions are used to initialize, save, and restore x87 processor state and to manage x87 exceptions.

Initialize

- FINIT—Floating-Point Initialize
- FNINIT—Floating-Point No-Wait Initialize

The FINIT and FNINIT instructions set all bits in the x87 control-word, status-word, and tag word registers to their default values. Assemblers issue FINIT as an FWAIT instruction followed by an FNINIT instruction. Thus, FINIT (but not FNINIT) reports pending unmasked x87 floating-point exceptions before performing the initialization.

Both FINIT and FNINIT write the control word with its initialization value, 037Fh, which specifies round-to-nearest, all exceptions masked, and double-extended-precision. The tag word indicates that the floating-point registers are *empty*. The status word and the four condition-code bits are cleared to 0. The x87 pointers and opcode state (“Pointers and Opcode State” on page 295) are all cleared to 0.

The FINIT instruction should be used when pending x87 floating-point exceptions are being reported (unmasked). The no-wait instruction, FNINIT, should be used when pending x87 floating-point exceptions are not being reported (masked).

Wait for Exceptions

- FWAIT or WAIT—Wait for Unmasked x87 Floating-Point Exceptions

The FWAIT and WAIT instructions are synonyms. The instruction forces the processor to test for and handle any pending, unmasked x87 floating-point exceptions.

Clear Exceptions

- FCLEX—Floating-Point Clear Flags
- FNCLEX—Floating-Point No-Wait Clear Flags

These instructions clear the status-word exception flags, stack-fault flag, and busy flag. They leave the four condition-code bits undefined.

Assemblers issue FCLEX as an FWAIT instruction followed by an FNCLEX instruction. Thus, FCLEX (but not FNCLEX) reports pending unmasked x87 floating-point exceptions before clearing the exception flags.

The FCLEX instruction should be used when pending x87 floating-point exceptions are being reported (unmasked). The no-wait instruction, FNCLEX, should be used when pending x87 floating-point exceptions are not being reported (masked).

Save and Restore x87 Control Word

- FLDCW—Floating-Point Load x87 Control Word
- FSTCW—Floating-Point Store Control Word
- FNSTCW—Floating-Point No-Wait Store Control Word

These instructions load or store the x87 control-word register as a 2-byte value from or to a memory location.

The FLDCW instruction loads a control word. If the loaded control word unmasks any pending x87 floating-point exceptions, these exceptions are reported when the next non-control x87 or 64-bit media instruction is executed.

Assemblers issue FSTCW as an FWAIT instruction followed by an FNSTCW instruction. Thus, FSTCW (but not FNSTCW) reports pending unmasked x87 floating-point exceptions before storing the control word.

The FSTCW instruction should be used when pending x87 floating-point exceptions are being reported (unmasked). The no-wait instruction, FNSTCW, should be used when pending x87 floating-point exceptions are not being reported (masked).

Save x87 Status Word

- FSTSW—Floating-Point Store Status Word
- FNSTSW—Floating-Point No-Wait Store Status Word

These instructions store the x87 status word either at a specified 2-byte memory location or in the AX register. The second form, F*x*STSW AX, is used in older code to copy condition codes through the AX register to the rFLAGS register, where they can be used for conditional branching using general-purpose instructions. However, the combination of FCOMI and FCMOV*cc* provides a faster method of conditional branching.

Assemblers issue FSTSW as an FWAIT instruction followed by an FNSTSW instruction. Thus, FSTSW (but not FNSTSW) reports pending unmasked x87 floating-point exceptions before storing the status word.

The FSTSW instruction should be used when pending x87 floating-point exceptions are being reported (unmasked). The no-wait instruction, FNSTSW, should be used when pending x87 floating-point exceptions are not being reported (masked).

Save and Restore x87 Environment

- FLDENV—Floating-Point Load x87 Environment
- FNSTENV—Floating-Point No-Wait Store Environment

- FSTENV—Floating-Point Store Environment

These instructions load or store the entire x87 environment (non-data processor state) as a 14-byte or 28-byte block, depending on effective operand size, from or to memory.

When executing FLDENV, any exception flags are set in the new status word, and these exceptions are unmasked in the control word, a floating-point exception occurs when the next non-control x87 or 64-bit media instruction is executed.

Assemblers issue FSTENV as an FWAIT instruction followed by an FNSTENV instruction. Thus, FSTENV (but not FNSTENV) reports pending unmasked x87 floating-point exceptions before storing the status word.

The x87 environment includes the x87 control word register, x87 status word register, x87 tag word, last x87 instruction pointer, last x87 data pointer, and last x87 opcode. See “Media and x87 Processor State” in Volume 2 for details on how the x87 environment is stored in memory.

Save and Restore x87 and 64-Bit Media State

- FSAVE—Save x87 and MMX State.
- FNSAVE—Save No-Wait x87 and MMX State.
- FRSTOR—Restore x87 and MMX State.

These instructions save and restore the entire processor state for x87 floating-point instructions and 64-bit media instructions. The instructions save and restore either 94 or 108 bytes of data, depending on the effective operand size.

Assemblers issue FSAVE as an FWAIT instruction followed by an FNSAVE instruction. Thus, FSAVE (but not FNSAVE) reports pending unmasked x87 floating-point exceptions before saving the state.

After saving the state, the processor initializes the x87 state by performing the equivalent of an FINIT instruction. For details, see “State-Saving” on page 339.

Save and Restore x87, 128-Bit, and 64-Bit State

- FXSAVE—Save XMM, MMX, and x87 State.
- FXRSTOR—Restore XMM, MMX, and x87 State.

The FXSAVE and FXRSTOR instructions save and restore the entire 512-byte processor state for 128-bit media instructions, 64-bit media instructions, and x87 floating-point instructions. The architecture supports two memory formats for FXSAVE and FXRSTOR, a 512-byte 32-bit legacy format and a 512-byte 64-bit format. Selection of the 32-bit or 64-bit format is determined by the effective operand size for the FXSAVE and FXRSTOR instructions. For details, see “Media and x87 Processor State” in Volume 2.

FXSAVE and FXRSTOR execute faster than FSAVE/FNSAVE and FRSTOR. However, unlike FSAVE and FNSAVE, FXSAVE does not initialize the x87 state, and like FNSAVE it does not report pending unmasked x87 floating-point exceptions. For details, see “State-Saving” on page 339.

6.5 Instruction Effects on rFLAGS

The rFLAGS register is described in “Flags Register” on page 34. Table 6-16 on page 326 summarizes the effect that x87 floating-point instructions have on individual flags within the rFLAGS register. Only instructions that access the rFLAGS register are shown—all other x87 instructions have no effect on rFLAGS.

The following codes are used within the table:

- Mod—The flag is modified.
- Tst—The flag is tested.
- Gray shaded cells indicate the flag is not affected by the instruction.

Table 6-16. Instruction Effects on rFLAGS

Instruction Mnemonic	rFLAGS Mnemonic and Bit Number					
	OF 11	SF 7	ZF 6	AF 4	PF 2	CF 0
FCMOV*cc*			Tst		Tst	Tst
FCOMI FCOMIP FUCOMI FUCOMIP			Mod		Mod	Mod

6.6 Instruction Prefixes

Instruction prefixes, in general, are described in “Instruction Prefixes” on page 76. The following restrictions apply to the use of instruction prefixes with x87 instructions.

6.6.0.1 Supported Prefixes

The following prefixes can be used with x87 instructions:

- *Operand-Size Override*—The 66h prefix affects only the FLDENV, FSTENV, FNSTENV, FSAVE, FNSAVE, and FRSTOR instructions, in which it selects between a 16-bit and 32-bit memory-image format. The prefix is ignored by all other x87 instructions.
- *Address-Size Override*—The 67h prefix affects only operands in memory, in which it selects between a 16-bit and 32-bit addresses. The prefix is ignored by all other x87 instructions.
- *Segment Overrides*—The 2Eh (CS), 36h (SS), 3Eh (DS), 26h (ES), 64h (FS), and 65h (GS) prefixes specify a segment. They affect only operands in memory. In 64-bit mode, the CS, DS, ES, SS segment overrides are ignored.
- *REX*—The REX.W bit affects the FXSAVE and FXRSTOR instructions, in which it selects between two types of 512-byte memory-image formats, as described in "Saving Media and x87 Processor State" in Volume 2. The REX.W bit also affects the FLDENV, FSTENV, FSAVE, and

FRSTOR instructions, in which it selects the 32-bit memory-image format. The REX.R, REX.X, and REX.B bits only affect operands in memory, in which they are used to find the memory operand.

6.6.0.2 Ignored Prefixes

The following prefixes are ignored by x87 instructions:

- *REP*—The F3h and F2h prefixes.

6.6.0.3 Prefixes That Cause Exceptions

The following prefixes cause an exception:

- *LOCK*—The F0h prefix causes an invalid-opcode exception when used with x87 instructions.

6.7 Feature Detection

Before executing x87 floating-point instructions, software should determine if the processor supports the technology by executing the CPUID instruction. “Feature Detection” on page 79 describes how software uses the CPUID instruction to detect feature support. For full support of the x87 floating-point features, the following feature must be present:

- On-Chip Floating-Point Unit, indicated by bit 0 of CPUID function 1 and CPUID function 8000_0001h.
- CMOV*cc* (conditional moves), indicated by bit 15 of CPUID function 1 and CPUID function 8000_0001h. This bit indicates support for x87 floating-point conditional moves (FCMOV*cc*) whenever the On-Chip Floating-Point Unit bit (bit 0) is also set.

Software may also wish to check for the following support, because the FXSAVE and FXRSTOR instructions execute faster than FSAVE and FRSTOR:

- FXSAVE and FXRSTOR, indicated by bit 24 of CPUID function 1 and function 8000_0001h.

Software that runs in long mode should also check for the following support:

- Long Mode, indicated by bit 29 of CPUID function 8000_0001h.

See “CPUID” in Volume 3 for details on the CPUID instruction and Appendix D of that volume for information on detemining support for specific instruction subsets.

6.8 Exceptions

6.8.0.1 Types of Exceptions

x87 instructions can generate two types of exceptions:

- *General-Purpose Exceptions*, described below in “General-Purpose Exceptions”

- *x87 Floating-Point Exceptions (#MF)*, described in "x87 Floating-Point Exception Causes" on page 329

6.8.0.2 Relation to 128-Bit Media Exceptions

Although the x87 floating-point instructions and the 128-bit media instructions each have certain exceptions with the same names, the exception-reporting and exception-handling methods used by the two instruction subsets are distinct and independent of each other. If procedures using both types of instructions are run in the same operating environment, separate service routines should be provided for the exceptions of each type of instruction subset.

6.8.1 General-Purpose Exceptions

The sections below list general-purpose exceptions generated and not generated by x87 floating-point instructions. For a summary of the general-purpose exception mechanism, see "Interrupts and Exceptions" on page 91. For details about each exception and its potential causes, see "Exceptions and Interrupts" in Volume 2.

6.8.1.1 Exceptions Generated

x87 instructions can generate the following general-purpose exceptions:

- #DB—Debug Exception (Vector 1)
- #BP—Breakpoint Exception (Vector 3)
- #UD—Invalid-Opcode Exception (Vector 6)
- #NM—Device-Not-Available Exception (Vector 7)
- #DF—Double-Fault Exception (Vector 8)
- #SS—Stack Exception (Vector 12)
- #GP—General-Protection Exception (Vector 13)
- #PF—Page-Fault Exception (Vector 14)
- #MF—x87 Floating-Point Exception-Pending (Vector 16)
- #AC—Alignment-Check Exception (Vector 17)
- #MC—Machine-Check Exception (Vector 18)

For details on #MF exceptions, see "x87 Floating-Point Exception Causes" below.

6.8.1.2 Exceptions Not Generated

x87 instructions do not generate the following general-purpose exceptions:

- #DE—Divide-by-zero-error exception (Vector 0)
- Non-Maskable-Interrupt Exception (Vector 2)
- #OF—Overflow exception (Vector 4)
- #BR—Bound-range exception (Vector 5)

- Coprocessor-segment-overrun exception (Vector 9)
- #TS—Invalid-TSS exception (Vector 10)
- #NP—Segment-not-present exception (Vector 11)
- #MC—Machine-check exception (Vector 18)
- #XF—SIMD floating-point exception (Vector 19)

For details on all general-purpose exceptions, see “Exceptions and Interrupts” in Volume 2.

6.8.2 x87 Floating-Point Exception Causes

The x87 floating-point exception-pending (#MF) exception listed above in “General-Purpose Exceptions” is actually the logical OR of six exceptions that can be caused by x87 floating-point instructions. Each of the six exceptions has a status flag in the x87 status word and a mask bit in the x87 control word. A seventh exception, stack fault (SF), is reported together with one of the six maskable exceptions and does not have a mask bit.

If a #MF exception occurs when its mask bit is set to 1 (*masked*), the processor responds in a default way that does not invoke the #MF exception service routine. If an exception occurs when its mask bit is cleared to 0 (*unmasked*), the processor suspends processing of the faulting instruction (if possible) and, at the boundary of the next non-control x87 or 64-bit media instruction (see “Control” on page 323), determines that an unmasked exception is pending—by checking the exception status (ES) flag in the x87 status word—and invokes the #MF exception service routine.

6.8.2.1 #MF Exception Types and Flags

The #MF exceptions are of six types, five of which are mandated by the IEEE 754 standard. These six types and their bit-flags in the x87 status word are shown in Table 6-17. A stack fault (SF) exception is always accompanied by an invalid-operation exception (IE). A summary of each exception type is given in “x87 Status Word Register (FSW)” on page 289.

Table 6-17. x87 Floating-Point (#MF) Exception Flags

Exception and Mnemonic	x87 Status-Word Bit[1]	Comparable IEEE 754 Exception
Invalid-operation exception (IE)	0	Invalid Operation
Invalid-operation exception (IE) with stack fault (SF) exception	0 and 6	*none*
Denormalized-operand exception (DE)	1	*none*
Zero-divide exception (ZE)	2	Division by Zero
Overflow exception (OE)	3	Overflow
Underflow exception (UE)	4	Underflow
Precision exception (PE)	5	Inexact

Note:
1. *See “x87 Status Word Register (FSW)” on page 289 for a summary of each exception.*

The sections below describe the causes for the #MF exceptions. Masked and unmasked responses to the exceptions are described in "x87 Floating-Point Exception Masking" on page 333. The priority of #MF exceptions are described in "x87 Floating-Point Exception Priority" on page 332.

6.8.2.2 Invalid-Operation Exception (IE)

The IE exception occurs due to one of the attempted operations shown in Table 6-18 on page 330. An IE exception may also be accompanied by a stack fault (SF) exception. See "Stack Fault (SF)" on page 331.

Table 6-18. Invalid-Operation Exception (IE) Causes

Operation		Condition
Arithmetic (IE exception)	Any Arithmetic Operation	• A source operand is an SNaN, or • A source operand is an unsupported data type (pseudo-NaN, pseudo-infinity, or unnormal).
	FADD, FADDP	Source operands are infinities with opposite signs.
	FSUB, FSUBP, FSUBR, FSUBRP	Source operands are infinities with same sign.
	FMUL, FMULP	Source operands are zero and infinity.
	FDIV, FDIVP, FDIVR, FDIVRP	Source operands are both infinities or both zeros.
	FSQRT	Source operand is less than zero (except ±0 which returns ±0).
	FYL2X	Source operand is less than zero (except ±0 which returns ±∞).
	FYL2XP1	Source operand is less than minus one.
	FCOS, FPTAN, FSIN, FSINCOS	Source operand is infinity.
	FCOM, FCOMP, FCOMPP, FCOMI, FCOMIP	A source operand is a QNaN.
	FPREM, FPREM1	Dividend is infinity or divisor is zero.
	FIST, FISTP, FISTTP	Source operand overflows the destination size.
	FBSTP	Source operand overflows packed BCD data size.
Stack (IE and SF exceptions)		Stack overflow or underflow.[1]

Note:
1. *The processor sets condition code C1 = 1 for overflow, C1 = 0 for underflow.*

6.8.2.3 Denormalized-Operand Exception (DE)

The DE exception occurs in any of the following cases:

- *Denormalized Operand (any precision)*—An arithmetic instruction uses an operand of any precision that is in denormalized form, as described in "Denormalized (Tiny) Numbers" on page 303.
- *Denormalized Single-Precision or Double-Precision Load*—An instruction loads a single-precision or double-precision (but not double-extended-precision) operand, which is in denormalized form, into an x87 register.

6.8.2.4 Zero-Divide Exception (ZE)

The ZE exception occurs when:

- *Divisor is Zero*—An FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, or FIDIVR instruction attempts to divide zero into a non-zero finite dividend.
- *Source Operand is Zero*—An FYL2X or FXTRACT instruction uses a source operand that is zero.

6.8.2.5 Overflow Exception (OE)

The OE exception occurs when the value of a rounded floating-point result is larger than the largest representable normalized positive or negative floating-point number in the destination format, as shown in Table 6-5 on page 301. An overflow can occur through computation or through conversion of higher-precision numbers to lower-precision numbers. See "Precision" on page 309. Integer and BCD overflow is reported via the invalid-operation exception.

6.8.2.6 Underflow Exception (UE)

The UE exception occurs when the value of a rounded, non-zero floating-point result is too small to be represented as a normalized positive or negative floating-point number in the destination format, as shown in Table 6-5 on page 301. Integer and BCD underflow is reported via the invalid-operation exception.

6.8.2.7 Precision Exception (PE)

The PE exception, also called the *inexact-result* exception, occurs when a floating-point result, after rounding, differs from the infinitely precise result and thus cannot be represented exactly in the specified destination format. Software that does not require exact results normally masks this exception. See "Precision" on page 309 and "Rounding" on page 309.

6.8.2.8 Stack Fault (SF)

The SF exception occurs when a stack overflow (due to a push or load into a non-empty stack register) or stack underflow (due to referencing an empty stack register) occurs in the x87 stack-register file. The empty and non-empty conditions are shown in Table 6-3 on page 295. When either of these conditions occur, the processor also sets the invalid-operation exception (IE) flag, and it sets or clears the condition-code 1 (C1) bit to indicate the direction of the stack fault (C1 = 1 for overflow, C1 = 0 for underflow). Unlike the flags for the other x87 exceptions, the SF flag does not have a corresponding mask bit in the x87 control word.

6.8.3 x87 Floating-Point Exception Priority

Table 6-19 shows the priority with which the processor recognizes multiple, simultaneous SIMD floating-point exceptions and operations involving QNaN operands. Each exception type is characterized by its timing, as follows:

- *Pre-Computation*—an exception that is recognized before an instruction begins its operation.
- *Post-Computation*—an exception that is recognized after an instruction completes its operation.

For post-computation exceptions, a result may be written to the destination, depending on the type of exception and whether the destination is a register or memory location. Operations involving QNaNs do not necessarily cause exceptions, but the processor handles them with the priority shown in Table 6-19 on page 332 relative to the handling of exceptions.

Table 6-19. Priority of x87 Floating-Point Exceptions

Priority	Exception or Operation	Timing
1	Invalid-operation exception (IE) with stack fault (SF) due to underflow	Pre-Computation
2	Invalid-operation exception (IE) with stack fault (SF) due to overflow	Pre-Computation
3	Invalid-operation exception (IE) when accessing unsupported data type	Pre-Computation
4	Invalid-operation exception (IE) when accessing SNaN operand	Pre-Computation
5	Operation involving a QNaN operand[1]	—
6	Any other type of invalid-operation exception (IE)	Pre-Computation
	Zero-divide exception (ZE)	Pre-Computation
7	Denormalized operation exception (DE)	Pre-Computation
8	Overflow exception (OE)	Post-Computation
	Underflow exception (UE)	Post-Computation
9	Precision (inexact) exception (PE)	Post-Computation

Note:
1. *Operations involving QNaN operands do not, in themselves, cause exceptions but they are handled with this priority relative to the handling of exceptions.*

For exceptions that occur before the associated operation (pre-operation, as shown in Table 6-19), if an unmasked exception occurs, the processor suspends processing of the faulting instruction but it waits until the boundary of the next non-control x87 or 64-bit media instruction to be executed before invoking the associated exception service routine. During this delay, non-x87 instructions may overwrite the faulting x87 instruction's source or destination operands in memory. If that occurs, the x87 service routine may be unable to perform its job.

To prevent such problems, analyze x87 procedures for potential exception-causing situations and insert a WAIT or other safe x87 instruction immediately after any x87 instruction that may cause a problem.

6.8.4 x87 Floating-Point Exception Masking

The six floating-point exception flags in the x87 status word have corresponding exception-flag masks in the x87 control word, as shown in Table 6-20 on page 333.

Table 6-20. x87 Floating-Point (#MF) Exception Masks

Exception Mask and Mnemonic	x87 Control-Word Bit[1]
Invalid-operation exception mask (IM)	0
Denormalized-operand exception mask (DM)	1
Zero-divide exception mask (ZM)	2
Overflow exception mask (OM)	3
Underflow exception mask (UM)	4
Precision exception mask (PM)	5

Note:
1. See "x87 Status Word Register (FSW)" on page 289 for a summary of each exception.

Each mask bit, when set to 1, inhibits invocation of the #MF exception handler and instead continues normal execution using the default response for the exception type. During initialization with FINIT or FNINIT, all exception-mask bits in the x87 control word are set to 1 (masked). At processor reset, all exception-mask bits are cleared to 0 (unmasked).

6.8.4.1 Masked Responses

The occurrence of a masked exception does not invoke its exception handler when the exception condition occurs. Instead, the processor handles masked exceptions in a default way, as shown in Table 6-21 on page 334.

Table 6-21. Masked Responses to x87 Floating-Point Exceptions

Exception and Mnemonic	Type of Operation[1]	Processor Response
Invalid-operation exception (IE)[2]	Any Arithmetic Operation: Source operand is an SNaN.	Set IE flag, and return a QNaN value.
	Any Arithmetic Operation: Source operand is an unsupported data type or FADD, FADDP: Source operands are infinities with opposite signs or FSUB, FSUBP, FSUBR, FSUBRP: Source operands are infinities with same sign or FMUL, FMULP: Source operands are zero and infinity or FDIV, FDIVP, FDIVR, FDIVRP: Source operands are both infinities or both are zeros or FSQRT: Source operand is less than zero (except ±0 which returns ±0) or FYL2X: Source operand is less than zero (except ±0 which returns ±∞) or FYL2XP1: Source operand is less than minus one.	Set IE flag, and return the floating-point indefinite value[3].

Note:
1. *See "Instruction Summary" on page 310 for the types of instructions.*
2. *Includes invalid-operation exception (IE) together with stack fault (SF).*
3. *See "Indefinite Values" on page 308.*

Table 6-21. Masked Responses to x87 Floating-Point Exceptions (continued)

Exception and Mnemonic	Type of Operation[1]	Processor Response
Invalid-operation exception (IE)[2]	FCOS, FPTAN, FSIN, FSINCOS: Source operand is ∞ or FPREM, FPREM1: Dividend is infinity or divisor is 0.	Set IE flag, return the floating-point indefinite value[3], and clear condition code C2 to 0.
	FCOM, FCOMP, or FCOMPP: One or both operands is a NaN or FUCOM, FUCOMP, or FUCOMPP: One or both operands is an SNaN.	Set IE flag, and set C3–C0 condition codes to reflect the result.
	FCOMI or FCOMIP: One or both operands is a NaN or FUCOMI or FUCOMIP: One or both operands is an SNaN.	Sets IE flag, and sets the result in eflags to "unordered."
	FIST, FISTP, FISTTP: Source operand overflows the destination size.	Set IE flag, and return the integer indefinite value[3].
	FXCH: A source register is specified *empty* by its tag bits.	Set IE flag, and perform exchange using floating-point indefinite value[3] as content for empty register(s).
	FBSTP: Source operand overflows packed BCD data size.	Set IE flag, and return the packed-decimal indefinite value[3].
Denormalized-operand exception (DE)		Set DE flag, and return the result using the denormal operand(s).
Zero-divide exception (ZE)	FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, or FIDIVR: Divisor is 0.	Set ZE flag, and return signed ∞ with sign bit = XOR of the operand sign bits.
	FYL2X: ST(0) is 0 and ST(1) is a non-zero floating-point value.	Set ZE flag, and return signed ∞ with sign bit = complement of sign bit for ST(1) operand.
	FXTRACT: Source operand is 0.	Set ZE flag, write ST(0) = 0 with sign of operand, and write ST(1) = –∞.

Note:
1. *See "Instruction Summary" on page 310 for the types of instructions.*
2. *Includes invalid-operation exception (IE) together with stack fault (SF).*
3. *See "Indefinite Values" on page 308.*

Table 6-21. Masked Responses to x87 Floating-Point Exceptions (continued)

Exception and Mnemonic	Type of Operation[1]	Processor Response
Overflow exception (OE)	Round to nearest.	• If sign of result is positive, set OE flag, and return +∞. • If sign of result is negative, set OE flag, and return -∞.
	Round toward +∞.	• If sign of result is positive, set OE flag, and return +∞. • If sign of result is negative, set OE flag, and return finite negative number with largest magnitude.
	Round toward -∞.	• If sign of result is positive, set OE flag, and return finite positive number with largest magnitude. • If sign of result is negative, set OE flag, and return -∞.
	Round toward 0.	• If sign of result is positive, set OE flag and return finite positive number with largest magnitude. • If sign of result is negative, set OE flag and return finite negative number with largest magnitude.
Underflow exception (UE)		• If result is both denormal (tiny) and inexact, set UE flag and return denormalized result. • If result is denormal (tiny) but not inexact, return denormalized result but do not set UE flag.
Precision exception (PE)	Without overflow or underflow	Set PE flag, return rounded result, write C1 condition code to specify round-up (C1 = 1) or not round-up (C1 = 0).
	With masked overflow or underflow	Set PE flag and respond as for the OE or UE exceptions.
	With unmasked overflow or underflow for register destination	Set PE flag, respond to the OE or UE exception by calling the #MF service routine.
	With unmasked overflow or underflow for memory destination	Do not set PE flag, respond to the OE or UE exception by calling the #MF service routine. The destination and the TOP are not changed.

Note:
1. *See "Instruction Summary" on page 310 for the types of instructions.*
2. *Includes invalid-operation exception (IE) together with stack fault (SF).*
3. *See "Indefinite Values" on page 308.*

6.8.4.2 Unmasked Responses

The processor handles unmasked exceptions as shown in Table 6-22 on page 337.

Table 6-22. Unmasked Responses to x87 Floating-Point Exceptions

Exception and Mnemonic	Type of Operation	Processor Response[1]
Invalid-operation exception (IE)		Set IE and ES flags, and call the #MF service routine[2]. The destination and the TOP are not changed.
Invalid-operation exception (IE) with stack fault (SF)		
Denormalized-operand exception (DE)		Set DE and ES flags, and call the #MF service routine[2]. The destination and the TOP are not changed.
Zero-divide exception (ZE)		Set ZE and ES flags, and call the #MF service routine[2]. The destination and the TOP are not changed.
Overflow exception (OE)		• If the destination is memory, set OE and ES flags, and call the #MF service routine[2]. The destination and the TOP are not changed. • If the destination is an x87 register: - divide true result by 2^{24576}, - round significand according to PC precision control and RC rounding control (or round to double-extended precision for instructions not observing PC precision control), - write C1 condition code according to rounding (C1 = 1 for round up, C1 = 0 for round toward zero), - write result to destination, - pop or push stack if specified by the instruction, - set OE and ES flags, and call the #MF service routine[2].

Note:

1. For all unmasked exceptions, the processor's response also includes assertion of the FERR# output signal at the completion of the instruction that caused the exception.

2. When CR0.NE is set to 1, the #MF service routine is taken at the next non-control x87 instruction. If CR0.NE is cleared to zero, x87 floating-point instructions are handled by setting the FERR# input signal to 1, which external logic can use to handle the interrupt.

Table 6-22. Unmasked Responses to x87 Floating-Point Exceptions (continued)

Exception and Mnemonic	Type of Operation	Processor Response[1]
Underflow exception (UE)		• If the destination is memory, set UE and ES flags, and call the #MF service routine[2]. The destination and the TOP are not changed. • If the destination is an x87 register: - multiply true result by 2^{24576}, - round significand according to PC precision control and RC rounding control (or round to double-extended precision for instructions not observing PC precision control), - write C1 condition code according to rounding (C1 = 1 for round up, C1 = 0 for round toward zero), - write result to destination, - pop or push stack if specified by the instruction, - set UE and ES flags, and call the #MF service routine[2].
Precision exception (PE)	Without overflow or underflow	Set PE and ES flags, return rounded result, write C1 condition code to specify round-up (C1 = 1) or not round-up (C1 = 0), and call the #MF service routine[2].
	With masked overflow or underflow	Set PE and ES flags, respond as for the OE or UE exception, and call the #MF service routine[2].
	With unmasked overflow or underflow for register destination	
	With unmasked overflow or underflow for memory destination	Do not set PE flag, respond to the OE or UE exception by calling the #MF service routine. The destination and the TOP are not changed.

Note:
1. *For all unmasked exceptions, the processor's response also includes assertion of the FERR# output signal at the completion of the instruction that caused the exception.*
2. *When CR0.NE is set to 1, the #MF service routine is taken at the next non-control x87 instruction. If CR0.NE is cleared to zero, x87 floating-point instructions are handled by setting the FERR# input signal to 1, which external logic can use to handle the interrupt.*

6.8.4.3 FERR# and IGNNE# Signals

In all unmasked-exception responses, the processor also asserts the FERR# output signal at the completion of the instruction that caused the exception. The exception is serviced at the boundary of the next non-waiting x87 or 64-bit media instruction following the instruction that caused the exception. (See "Control" on page 323 for a definition of *control instructions*.)

System software controls x87 floating-point exception reporting using the numeric error (NE) bit in control register 0 (CR0), as follows:

- If CR0.NE = 1, internal processor control over x87 floating-point exception reporting is enabled. In this case, an #MF exception occurs immediately. The FERR# output signal is asserted, but is not

used externally. It is recommended that system software set NE to 1. This enables optimal performance in handling x87 floating-point exceptions.

- If CR0.NE = 0, internal processor control of x87 floating-point exceptions is disabled and the external IGNNE# input signal controls whether x87 floating-point exceptions are ignored, as follows:
  - When IGNNE# is 0, x87 floating-point exceptions are reported by asserting the FERR# output signal, then stopping program execution until an external interrupt is detected. External logic use the FERR# signal to generate the external interrupt.
  - When IGNNE# is 1, x87 floating-point exceptions do not stop program execution. After FERR# is asserted, instructions continue to execute.

6.8.4.4 Using NaNs in IE Diagnostic Exceptions

Both SNaNs and QNaNs can be encoded with many different values to carry diagnostic information. By means of appropriate masking and unmasking of the invalid-operation exception (IE), software can use signaling NaNs to invoke an exception handler. Within the constraints imposed by the encoding of SNaNs and QNaNs, software may freely assign the bits in the significand of a NaN. See the section "Not a Number (NaN)" on page 305 for format details.

For example, software can pre-load each element of an array with a signaling NaN that encodes the array index. When an application accesses an uninitialized array element, the invalid-operation exception is invoked and the service routine can identify that element. A service routine can store debug information in memory as the exceptions occur. The routine can create a QNaN that references its associated debug area in memory. As the program runs, the service routine can create a different QNaN for each error condition, so that a single test-run can identify a collection of errors.

6.9 State-Saving

In general, system software should save and restore x87 state between task switches or other interventions in the execution of x87 floating-point procedures. Virtually all modern operating systems running on x86 processors implement preemptive multitasking that handle saving and restoring of state across task switches, independent of hardware task-switch support. However, application procedures are also free to save and restore x87 state at any time they deem useful.

6.9.1 State-Saving Instructions

6.9.1.1 FSAVE/FNSAVE and FRSTOR Instructions

Application software can save and restore the x87 state by executing the FSAVE (or FNSAVE) and FRSTOR instructions. Alternatively, software may use multiple F*x*ST*x* (floating-point store stack top) instructions for saving only the contents of the x87 data registers, rather than the complete x87 state.

The FSAVE instruction stores the state, but only after handling any pending unmasked x87 floating-point exceptions, whereas the FNSAVE instruction skips the handling of these exceptions. The state of all x87 data registers is saved, as well as all x87 environment state (the x87 control word register,

status word register, tag word, instruction pointer, data pointer, and last opcode register). After saving this state, the tag bits for all x87 registers are changed to *empty* and thus available for a new procedure.

6.9.1.2 FXSAVE and FXRSTOR Instructions

Application software can save and restore the 128-bit media state, 64-bit media state, and x87 floating-point state by executing the FXSAVE and FXRSTOR instructions. The FXSAVE and FXRSTOR instructions execute faster than FSAVE/FNSAVE and FRSTOR because they do not save and restore the x87 pointers (last instruction pointer, last data pointer, and last opcode, described in “Pointers and Opcode State” on page 295) except in the relatively rare cases in which the exception-summary (ES) bit in the x87 status word (the ES register image for FXSAVE, or the ES memory image for FXRSTOR) is set to 1, indicating that an unmasked x87 exception has occurred.

Unlike FSAVE and FNSAVE, however, FXSAVE does not alter the tag bits. The state of the saved x87 data registers is retained, thus indicating that the registers may still be valid (or whatever other value the tag bits indicated prior to the save). To invalidate the contents of the x87 data registers after FXSAVE, software must explicitly execute an FINIT instruction. Also, FXSAVE (like FNSAVE) and FXRSTOR do not check for pending unmasked x87 floating-point exceptions. An FWAIT instruction can be used for this purpose.

The architecture supports two memory formats for FXSAVE and FXRSTOR, a 512-byte 32-bit legacy format and a 512-byte 64-bit format, used in 64-bit mode. Selection of the 32-bit or 64-bit format is determined by the effective operand size for the FXSAVE and FXRSTOR instructions. For details, see “Media and x87 Processor State” in Volume 2.

6.10 Performance Considerations

In addition to typical code optimization techniques, such as those affecting loops and the inlining of function calls, the following considerations may help improve the performance of application programs written with x87 floating-point instructions.

These are implementation-independent performance considerations. Other considerations depend on the hardware implementation. For information about such implementation-dependent considerations and for more information about application performance in general, see the data sheets and the software-optimization guides relating to particular hardware implementations.

6.10.1 Replace x87 Code with 128-Bit Media Code

Code written with 128-bit media floating-point instructions can operate in parallel on four times as many single-precision floating-point operands as can x87 floating-point code. This achieves potentially four times the computational work of x87 instructions that use single-precision operands. Also, the higher density of 128-bit media floating-point operands may make it possible to remove local temporary variables that would otherwise be needed in x87 floating-point code. 128-bit media code is easier to write than x87 floating-point code, because the XMM register file is flat rather than stack-oriented, and, in 64-bit mode there are twice the number of XMM registers as x87 registers.

6.10.2 Use FCOMI-FCMOV*x* Branching

Depending on the hardware implementation of the architecture, the combination of FCOMI and FCMOV*cc* is often faster than the classical approach using FxSTSW AX instructions for comparison-based branches that depend on the condition codes for branch direction, because FNSTSW AX is often a serializing instruction.

6.10.3 Use FSINCOS Instead of FSIN and FCOS

Frequently, a piece of code that needs to compute the sine of an argument also needs to compute the cosine of that same argument. In such cases, use the FSINCOS instruction to compute both trigonometric functions concurrently, which is faster than using separate FSIN and FCOS instructions to accomplish the same task.

6.10.4 Break Up Dependency Chains

Parallelism can be increased by breaking up dependency chains or by evaluating multiple dependency chains simultaneously (explicitly switching execution between them). Depending on the hardware implementation of the architecture, the FXCH instruction may prove faster than FST/FLD pairs for switching execution between dependency chains.

Index

Symbols

Numerics

A

B

C

D

E

F

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

AMD64 Technology

AMD64 Architecture Programmer's Manual

Volume 2: System Programming

Publication No.	Revision	Date
24593	3.38	November 2021

Advanced Micro Devices

Trademarks

Contents

Figures

Tables

Revision History

Date	Revision	Description
November 2021	3.38	Added corrections/clarifications to: Address generation in Section 4.5.3. PCID behavior in Section 5.5. Machine Check Exception in Section 8.2.18. SEV Status MSR in Section 15.34.10. SEV_SNP in Sections 15.36.7, .10, .17. VMCB and VMSA contents in Appendix B. SVM Intercepts in Appendix C. Added new features: Chapter 5: Upper Address Ignore. Chapter 9, Section 3.2.6 and Appendix A: Machine Check Architecture Extension. Section 15.36: VMSA Register Protection, SEV-SNP VMSA Register Protection and Secure TSC.
March 2021	3.37	Added SPEC_CTRL and PRED_CMD support. Added x2APIC support. Added SMM Page Configuration Lock. Corrected the AMD64 Architecture SMM State-Save Area table. Add new section: APERF Read-only (AperfReadOnly). Updated the MSRs of the AMD64 Architecture table. Updated the Machine-Check MSR Cross-Reference table. Updated the Performance-Monitoring MSR Cross-Reference table. Updated the Secure Virtual Machine MSR Cross-Reference table. Added new section: Speculation Control MSRs.

Date	Revision	Description
August 2020	3.36	Added Shadow Stack support. Chapter 3: Section 3.1: Added content. Section 3.1.3: Added bit 23 and new content. Section 3.2.7: Added Shadow Stack Registers as new section 3.2.7. Chapter 5: Section 5.6: Updated content. Chapter 6: Added content to Table 6-1. Added section 6.7. Chapter 7: Table 7-3. Updated. Chapter 8: Table 8-1 and Table 8-2: Added content. Section 8.4: Updated content. Section 8.4.2: Added content. Added #CP as new section 8.2.20. Table 8-8: Added content. Added new Shadow Stack section 8.7.6. Section 8.9: Added bullet. Added new 8.9.4.1 section. Chapter 10: Updated Table 10-1. Section 10.4: Added bullet. Chapter 11: Section 11.5.2 and Figure 11-8: Updated table and figure. Section 11.5.8: Updated Table 11-3. Chapter 12: Section 12.2.2: Updated. Section 12.2.4: Updated Figure 12-6. Section 12.3.2: Updated content and added bullets. Chapter 15: Section 15.5.1: Updates. Section 15.15.3: Updated Figure 15-4. Section 15.25.6: Added bullet. Section 15.29.1. Updates. Section 15.29.4.1: Added content. Section 15.29.8.2: Added content. Added Chapter 18. Appendix A: Table A-1: Added content. Added new section A.10. Appendix B: Table B-1, Table B-2, and Table 4: Added content.
May 2020	3.35	Sections 5.6.1 and 5.6.6: Minor updates. Figure 5-16: Updated figure.

Date	Revision	Description
April 2020	3.34	Section 3.1.3: Updated register information. Added PCIDE and PKE registers. Updated (TCE) content. Section 5.3.2: Added Process Context Identifier register information and register figure. Section 5.3.3: Updated figure. Section 5.3.4: Updated figure. Section 5.3.5: Updated figure. Section 5.4.1: Added (MPK) register information. Section 5.5.1: Inserted Process Context Identifiers as Section 5.5.1. Section 5.5.3: Added bullets to Implicit Invalidations list. Section 5.6: Updated content. Section 8.2.15: Added bullet. Section 8.4.2: Updated register figure and added PK register information. Section 11.5.2: Updated register figure and table. Section 14.1.3: Updated table. Section 15.9: Updated table. Appendix B: Updated table. Appendix C: Updated table.

Date	Revision	Description
April 2020	3.33	Section 1.1.2: Clarification on address size support. Section 3.2.1: New feature enable bits in SYSCFG MSR. Section 7.6.5: Updated terminology. Section 7.10.6: Clarification to encrypted memory operation. Section 8.1.4: Clarification to IRET and NMI behavior. Tables 8-1 and 8-2: Added #HV exception. Inserted new 8.2.20 section for #HV exception. Section 8.4.2: Changes for SEV-SNP extension. Table 8-8: Added SEV-related exceptions. Figure 10-6: Updated I/O Restart DWORD. Section 15 and 15.1: General updates. Section 15.5.2: Relocated VMLOAD/VMSAVE documentation. Section 15.2.4 and 15.2.6: Updated content. Section 15.6: Added content. Table 15-7: Added content. Section 15.25.6: Clarification. Section 15.25.13: General clarifications. Section 15.33.1 and 15.33.2: General clarifications. Section 15.34.3, 15.34.7, and 15.34.10: Clarifications, and additions for SEV-SNP. Table 15-35: Added content. Section 15.35.8: Corrected terminology. Section 15.36: Added SEV-SNP extension documentation. Table A-1 and Table A-7: Added SEV-SNP related MSRs. Appendix B: Updates for SEV-SNP extension. Table C-1: Added exit code for SEV-SNP extension.
October 2019	3.32	Added UMIP, XSS, GMET, VTE, MCOMMIT, and RDPRU.
July 2019	3.31	Added CLWB and WBNOINVD details. Clarified FP error pointer save/restore behavior. Corrected description of APIC Software Enable functionality. Clarified canonical address checking behavior. Clarified fault generation. November 2021n for instructions that cross page or segment boundaries.

Date	Revision	Description
September 2018	3.30	Modified Section 7.4 Modified Section 7.6.4 Modified Section 8.5.2 Modified Section 9.2 Corrected Figure 9-4 Corrected Table 9-1 Modified Section 9.3.2 Corrected Figure 9-6 Corrected Table 9-4 Modified Section 14.2.3 Modified Section 14.4 Modified Section 15.6 Modified Section 15.7 Modified Section 15.34.9 Modified Section 15.34.10 Modified Section 15.35.2 Corrected Table B-4 in Appendix B
December 2017	3.29	Modified Sections 7.10.1 and 7.10.4. Modified Sections 15.34.1, 15.34.7. Added new Section 15.34.10. Modified Section 15.35.10. Modified Appendix A, Table A-7.
March 2017	3.28	Modified CR4 Register, Section 3.1.3. Removed UD2 in Table 6-1. Added new bullet in Section 7.1.1. Modified Note in Table 7-1. Added new Section 7.4.1. Clarified Self Modifying Code in Section 7.6.1. Added UD0 and UD1 instructions in Section 8.2.7. Added Instructions Retired Performance counter in Section 13.1.1. Modified Table in Section 15.34.9.

Date	Revision	Description
December 2016	3.27	Added Resume Flag (RF) Bit in Section 3.1.6, ”RFLAGS Register,” on page 52. Added Tom2ForceMemTypeWB in Section 3.2.1, ”System Configuration Register (SYSCFG),” on page 61. Clarified SYSCALL and SYSRET in Section 6.1.1, ”SYSCALL and SYSRET,” on page 171. Added Section 7.3.2, ”Access Atomicity,” on page 192. Updated Note b in Table 7-11 on page 222. Modified Table 8-1, “Interrupt Vector Source and Cause”, on page 237. Modified Table 8-2, “Interrupt Vector Classification”, on page 238. Added Section 8.2.22, ”#VC—VMM Communication Exception (Vector 29),” on page 252. Added a Note in Chapter 10, "System-Management Mode," on page 317. Added Section 10.5, ”Multiprocessor Considerations,” on page 334. Updated CPUID 8000_001F[EAX] and added CPUID 8000_001F[EDX] in Section 15.34.1, ”Determining Support for SEV,” on page 572. Added new Section 15.35, ”Encrypted State (SEV-ES),” on page 578. Clarified TSC Ratio MSR in Section 15.30.5 ”TSC Ratio MSR (C000_0104h)” on page 568. Modified Appendix B, ”Layout of VMCB” on page 679. Added Table B-3, “Swap Types”, on page 687. Added Codes 8Fh, 90h-9Fh, and 403h in Table C-1, “SVM Intercept Codes”, on page 693.
April 2016	3.26	Clarification on loading a null selector into FS or GS added in Section 4.5.3, ”Segment Registers in 64-Bit Mode,” on page 80 Translation table diagrams corrected for definition of bit 8 in Section 5.5, ”Translation-Lookaside Buffer (TLB),” on page 154 CR0.CD implementation-dependent behavior noted in Section 7.6.2, ”Cache Control Mechanisms,” on page 202 Added clarification on IST usage in Section 8.9.4, ”Interrupt-Stack Table,” on page 276. Added new Section 7.10, ”Secure Memory Encryption,” on page 228. Added guideline for secure AP startup in Section 15.27.8, ”Secure Multiprocessor Initialization,” on page 544 Added TLB maintenance requirement for multiprocessor VM's in Section 15.29.4, ”VMCB Changes in Support of AVIC,” on page 552. Added new Section 15.34, ”Secure Encrypted Virtualization,” on page 571

Date	Revision	Description
June 2015	3.25	Added new section 15.33 Nested Virtualization for coverage of VMSAVE and VMLOAD Virtualization and Virtual GIF. Various minor edits.
October 2013	3.24	Added description of Supervisor-Mode Execution Prevention. See Section 5.6.5 ”Supervisor-Mode Execution Prevention (CR4.SMEP) Bit” on page 160. Indicated the deprecation of the Processor Feedback Interface. See Section 17.4, ”Processor Feedback Interface,” on page 644. Added Section 17.5, ”Processor Core Power Reporting,” on page 644.
May 2013	3.23	Clarified guidelines for implementing cross-modifying code in the sub-section ”Cross-Modifying Code” on page 201. Added AVIC description. See Section 15.29, ”Advanced Virtual Interrupt Controller,” on page 546. Added L2I PMC architecture definition. See Section 13.2, ”Performance Monitoring Counters,” on page 400.
September 2012	3.22	Clarified processor behavior on write of EFER[LMA] bit in Section 3.1.7 ”Extended Feature Enable Register (EFER)” on page 56. Clarified difference between cold reset and warm reset in Section 9.3, ”Machine Check Architecture MSRs,” on page 291. Added information on FFXSR feature bit to Table 11-1 on page 336. Clarified SMM code responsibility to manage VMCB clean bits. See Section 15.15.2, ”Guidelines for Clearing VMCB Clean Bits,” on page 511. Added a note to Table 15-9 on page 514 to indicate that all encodings of TLB_CONTROL not defined are reserved. Corrected information concerning the assignment of logical APIC IDs in Section 16.6.1, ”Receiving System and IPI Interrupts,” on page 619.
March 2012	3.21	Added definition of processor feedback interface—frequency sensitivity monitor (See Section 17.4, ”Processor Feedback Interface,” on page 644) Added Instruction-Based Sampling in a new section of Chapter 13 (See Section 13.3, ”Instruction-Based Sampling,” on page 409.) Reworked Introduction and first section of Chapter 9, "Machine Check Architecture," on page 287 and added deferred error handling. Added description of CR4[FSGSBASE] bit. (See Section 3.1.3, ”CR4 Register,” on page 47.) Added references to the RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE instructions in discussion of FS and GS segment descriptors. (See ”FS and GS Registers in 64-Bit Mode” on page 80) Added Section 6.3.2, ”Accessing Segment Register Hidden State,” on page 176.

Date	Revision	Description
December 2011	3.20	Clarified description of the Cache Disable (CD) memory type in Section 7.4 ”Memory Types” on page 192. Added caveat: an overflow of either APERF or MPERF can invalidate the effective frequency calculation. See Section 17.3, ”Determining Processor Effective Frequency,” on page 642. Other minor editorial changes.
September 2011	3.19	Added XSAVEOPT to discussions on XSAVE. Corrections to discussion on multiprocessor memory access ordering in Chapter 7. Added discussion of extended core and northbridge performance counters and feature indicators to Chapter 13. Added Lightweight Profiling (LWP) to Chapter 13. Added Global Timestamp Counter, Continuous Mode to LWP description Clarification: Function of pin A20M# is only defined in real mode. Statement added to Section 1.2.4, ”Real Addressing,” on page 10. Eliminated hardware P-state references
May 2011	3.18	Added information for OSXSAVE and XSAVE features. Added Cache Topology, Pause Filter Threshold, and XSETBV information. Updated TSC ratio information. Corrected description of FXSAVE/FXRSTOR exception behavior when CR0.EM=1
June 2010	3.17	Replaced missing figures in Chapter 8, "Exceptions and Interrupts," on page 233.
June 2010	3.16	Updated information on performance monitoring counters in ”Performance-Monitoring Counter Enable (PCE)” on page 50 and 6.2.5, ”Accessing Model-Specific Registers” on page 175. Revised Table 4-1, ”Segment Registers” on page 79. Add flush by ASID information to section 15.16, ”TLB Control” on page 513. Added information on VMCB clean field to Chapter15, ”Secure Virtual Machine” on page 483 and Appendix B, ”Layout of VMCB” on page 679. Added section 15.10, ”IOIO Intercepts” on page 501. Added section 15.30.5, ”TSC Ratio MSR (C000_0104h)” on page 568. Added section 17.2, ”Core Performance Boost” on page 641.

Date	Revision	Description
November 2009	3.15	Added section 7.5, ”Buffering and Combining Memory Writes” on page 197 Added MFENCE to list of ”Serializing Instructions” on page 206. Updated section 7.6.1, ”Cache Organization and Operation” on page 199. Updated Table 7-3, “Memory Access Ordering Rules”, on page 196 and notes. Updated 7.4, ”Memory Types” on page 192. Clarified 5.5.3, ”TLB Management” on page 155. Added ”Invalidation of Table Entry Upgrades.” on page 156. Updated ”Speculative Caching of Address Translations” on page 156. Update ”Handling of D-Bit Updates” on page 157. Revised and updated section 7.2, ”Multiprocessor Memory Access Ordering” on page 186 ff. Added information on long mode segment-limit checks in ”Extended Feature Enable Register (EFER)” on page 57table on page 57 and ”Long Mode Segment Limit Enable (LMSLE) bit” on page 58 on page 58. Added discussion of ”Data Limit Checks in 64-bit Mode” on page 123on page 123. Updated Table 6-1, “System Management Instructions”, on page 167. Updated ”Canonicalization and Consistency Checks” on page 489on page 489. Added information about the next sequential instruction pointer (nRIP) in 15.7.1, ”State Saved on Exit” on page 493. Updated priority definition of PAUSE instruction intercept in Table 15-7, “Instruction Intercepts”, on page 498. Added nRIP field to Table B-1, “VMCB Layout, Control Area”, on page 679. Clarified information on ICEBP event injection, on page 516. Deleted erroneous statement concerning the operation of the General Local Vector Table register Mask bit in section 16.4. Clarified the description of the Interrupt Command Register Delivery Status bit in section ”Interprocessor Interrupts (IPI)” on page 615on page 615.

Date	Revision	Description
September 2007	3.14	Added information on ”Speculative Caching of Address Translations,” ”Caching of Upper Level Translation Table Entries,” ”Use of Cached Entries When Reporting a Page Fault Exception,” ”Use of Cached Entries When Reporting a Page Fault Exception,” ”Handling of D-Bit Updates,” ”Invalidation of Cached Upper-level Entries by INVLPG” on page 157 and ”Handling of PDPT Entries in PAE Mode” on page 157to section 5.5.3, ”TLB Management” on page 155. Added 15.21.7, ”Interrupt Masking in Local APIC” on page 520. Added 16.3.6, ”Extended APIC Control Register” on page 607; clarified the use of the ICR DS bit in 16.5, ”Interprocessor Interrupts (IPI)” on page 615. Added minor clarifications and corrected typographical and formatting errors.
July 2007	3.13	Added 5.3.5, ”1-Gbyte Page Translation” on page 147. Added 7.2, ”Multiprocessor Memory Access Ordering” on page 186 Added divide-by-zero exception to Table 8-9, “Simultaneous Interrupt Priorities”, on page 255. Added information on ”CPU Watchdog Timer Register” on page 295and ”Machine-Check Miscellaneous-Error Information Register 0 (MC*i*_MISC0)” on page 302to Chapter 9. Added SSE4A support to Chapter 11, ”SSE, MMX, and x87 Programming” on page 335. Added Monitor and MWAIT intercept information to section 15.9, ”Instruction Intercepts” on page 498 and reorganized intercept information; clarified 15.16.1, ”TLB Flush” on page 513. Added Monitor and MWAIT intercepts to tables B-1, ”VMCB Layout, Control Area” on page 679 and C-1, ”SVM Intercept Codes” on page 693. Added Chapter 16, ”Advanced Programmable Interrupt Controller (APIC)” on page 601, Chapter 17, ”OS-Visible Workaround Information” on page 515, Chapter 17, ”Hardware Performance Monitoring and Control” on page 639. Added Table A-7, “Secure Virtual Machine MSR Cross-Reference”, on page 675. Added minor clarifications and corrected typographical and formatting errors.
September 2006	3.12	Added numerous minor clarifications.
December 2005	3.11	Added Chapter 15, Secure Virtual Machine. Incorporated numerous factual corrections and updates.
February 2005	3.10	Corrected Table 8-6, “General-Protection Exception Conditions”, on page 246. Added SSE3 information. Clarified and corrected information on the CPUID instruction and feature identification. Added information on the RDTSCP instruction. Clarified information about MTRRs and PATs in multiprocessing systems.

Date	Revision	Description
September 2003	3.09	Corrected numerous minor typographical errors.
April 2003	3.08	Clarified terms in section on FXSAVE/FXSTOR. Corrected several minor errors of omission. Documentation of CR0.NW bit has been corrected. Several register diagrams and figure labels have been corrected. Description of shared cache lines has been clarified in 7.3, ”Memory Coherency and Protocol” on page 189.
September 2002	3.07	Made numerous small grammatical changes and factual clarifications. Added Revision History.

Preface

About This Book

This book is part of a multivolume work entitled the *AMD64 Architecture Programmer's Manual*. This table lists each volume and its order number.

Title	Order No.
Volume 1: Application Programming	24592
Volume 2: System Programming	24593
Volume 3: General-Purpose and System Instructions	24594
Volume 4: 128-Bit and 256-Bit Media Instructions	26568
Volume 5: 64-Bit Media and x87 Floating-Point Instructions	26569

Audience

This volume (Volume 2) is intended for programmers writing operating systems, loaders, linkers, device drivers, or system utilities. It assumes an understanding of AMD64 architecture application-level programming as described in Volume 1.

This volume describes the AMD64 architecture's resources and functions that are managed by system software, including operating-mode control, memory management, interrupts and exceptions, task and state-change management, system-management mode (including power management), multi-processor support, debugging, and processor initialization.

Application-programming topics are described in Volume 1. Details about each instruction are described in Volumes 3, 4, and 5.

Organization

This volume begins with an overview of system programming and differences between the x86 and AMD64 architectures. This is followed by chapters that describe the following details of system programming:

- *System Resources*—The system registers and processor ID (CPUID) functions.
- *Segmented Virtual Memory*—The segmented-memory models supported by the architecture and their associated data structures and protection checks.
- *Page Translation and Protection*—The page-translation functions supported by the architecture and their associated data structures and protection checks.

- *System Instructions*—The instructions used to manage system functions.
- *Memory System*—The memory-system hierarchy and its resources and protocols, including memory-characterization, caching, and buffering functions.
- *Exceptions and Interrupts*—Details about the types and causes of exceptions and interrupts, and the methods of transferring control during these events.
- *Machine-Check Mechanism*—The resources and functions that support detection and handling of machine-check errors.
- *System-Management Mode*—The resources and functions that support system-management mode (SMM), including power-management functions.
- *SSE, MMX, and x87 Programming*—The resources and functions that support use (by application software) and state-saving (by the operation system) of the 256-bit media, 128-bit media, 64-bit media, and x87 floating-point instructions.
- *Multiple-Processor Management*—The features of the instruction set and the system resources and functions that support multiprocessing environments.
- *Debug and Performance Resources*—The system resources and functions that support software debugging and performance monitoring.
- *Legacy Task Management*—Support for the legacy hardware multitasking functions, including register resources and data structures.
- *Processor Initialization and Long-Mode Activation*—The methods by which system software initializes and changes operating modes.
- *Mixing Code Across Operating Modes*—Things to remember when running programs in different operating modes.
- *Secure Virtual Machine*—The system resources that support machine virtualization.
- *Advanced Programmable Interrupt Controller (APIC) operation.*

There are appendices describing details of model-specific registers (MSRs) and machine-check implementations. Definitions assumed throughout this volume are listed below. The index at the end of this volume cross-references topics within the volume. For other topics relating to the AMD64 architecture, see the tables of contents and indexes of the other volumes.

Conventions and Definitions

The section which follows, **Notational Conventions,** describes notational conventions used in this volume. The next section, **Definitions,** lists a number of terms used in this volume along with their technical definitions. Some of these definitions assume knowledge of the legacy x86 architecture. See “Related Documents” on page lvii for further information about the legacy x86 architecture. Finally, the **Registers** section lists the registers which are a part of the system programming model.

Notational Conventions

#GP(0)

An instruction exception—in this example, a general-protection exception with error code of 0.

1011b

A binary value—in this example, a 4-bit value.

F0EA_0B02h

A hexadecimal value. Underscore characters may be inserted to improve readability.

128

Numbers without an alpha suffix are decimal unless the context indicates otherwise.

7:4

A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first. Commas may be inserted to indicate gaps.

CPUID Fn*XXXX_XXXX_RRR*[*FieldName*]

Support for optional features or the value of an implementation-specific parameter of a processor can be discovered by executing the CPUID instruction on that processor. To obtain this value, software must execute the CPUID instruction with the function code *XXXX_XXXX*h in EAX and then examine the field *FieldName* returned in register *RRR*. If the "*_RRR*" notation is followed by "_x*YYY*", register ECX must be set to the value *YYY*h before executing CPUID. When *FieldName* is not given, the entire contents of register *RRR* contains the desired value. When determining optional feature support, if the bit identified by *FieldName* is set to a one, the feature is supported on that processor.

CR0–CR4

A register range, from register CR0 through CR4, inclusive, with the low-order register first.

CR0[PE], CR0.PE

Notation for referring to a field within a register—in this case, the PE field of the CR0 register.

CR0[PE] = 1, CR0.PE = 1

The PE field of the CR0 register is set (contains the value 1).

EFER[LME] = 0, EFER.LME = 0

The LME field of the EFER register is cleared (contains a value of 0).

DS:SI

A far pointer or logical address. The real address or segment descriptor specified by the segment register (DS in this example) is combined with the offset contained in the second register (SI in this example) to form a real or virtual address.

RFLAGS[13:12]

A field within a register identified by its bit range. In this example, corresponding to the IOPL field.

Definitions

16-bit mode

Legacy mode or compatibility mode in which a 16-bit address size is active. See *legacy mode* and *compatibility mode*.

32-bit mode

Legacy mode or compatibility mode in which a 32-bit address size is active. See *legacy mode* and *compatibility mode*.

64-bit mode

A submode of *long mode*. In 64-bit mode, the default address size is 64 bits and new features, such as register extensions, are supported for system and application software.

absolute

Said of a displacement that references the base of a code segment rather than an instruction pointer. Contrast with *relative*.

ASID

Address space identifier.

byte

Eight bits.

clear

To write a bit value of 0. Compare *set*.

compatibility mode

A submode of *long mode*. In compatibility mode, the default address size is 32 bits, and legacy 16-bit and 32-bit applications run without modification.

commit

To irreversibly write, in program order, an instruction's result to software-visible storage, such as a register (including flags), the data cache, an internal write buffer, or memory.

CPL

Current privilege level.

direct

Referencing a memory location whose address is included in the instruction's syntax as an immediate operand. The address may be an absolute or relative address. Compare *indirect*.

dirty data

Data held in the processor's caches or internal buffers that is more recent than the copy held in main memory.

displacement

A signed value that is added to the base of a segment (absolute addressing) or an instruction pointer (relative addressing). Same as *offset*.

doubleword

Two words, or four bytes, or 32 bits.

double quadword

Eight words, or 16 bytes, or 128 bits. Also called *octword*.

effective address size

The address size for the current instruction after accounting for the default address size and any address-size override prefix.

effective operand size

The operand size for the current instruction after accounting for the default operand size and any operand-size override prefix.

exception

An abnormal condition that occurs as the result of executing an instruction. The processor's response to an exception depends on the type of the exception. For all exceptions except 128-bit media SIMD floating-point exceptions and x87 floating-point exceptions, control is transferred to the handler (or service routine) for that exception, as defined by the exception's vector. For floating-point exceptions defined by the IEEE 754 standard, there are both masked and unmasked responses. When unmasked, the exception handler is called, and when masked, a default response is provided instead of calling the handler.

flush

An often ambiguous term meaning (1) writeback, if modified, and invalidate, as in "flush the cache line," or (2) invalidate, as in "flush the pipeline," or (3) change a value, as in "flush to zero."

GDT

Global descriptor table.

GIF

Global interrupt flag.

GPA

Guest physical address. In a virtualized environment, the page tables maintained by the guest operating system provide the translation from the linear (virtual) address to the guest physical

address. Nested page tables define the translation of the GPA to the host physical address (HPA). See *SPA* and *HPA*.

HPA

Host physical address. The address space owned by the virtual machine monitor. In a virtualized environment, nested page translation tables controlled by the VMM provide the translation from the guest physical address to the host physical address. See *GPA*.

IDT

Interrupt descriptor table.

IGN

Ignored. Value written is ignored by hardware. Value returned on a read is indeterminate. See *reserved*.

indirect

Referencing a memory location whose address is in a register or other memory location. The address may be an absolute or relative address. Compare *direct*.

IRB

The virtual-8086 mode interrupt-redirection bitmap.

IST

The long-mode interrupt-stack table.

IVT

The real-address mode interrupt vector table.

LDT

Local descriptor table.

legacy x86

The legacy x86 architecture. See “Related Documents” on page lvii for descriptions of the legacy x86 architecture.

legacy mode

An operating mode of the AMD64 architecture in which existing 16-bit and 32-bit applications and operating systems run without modification. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Legacy mode has three submodes, *real mode*, *protected mode*, and *virtual-8086 mode*.

LIP

Linear Instruction Pointer. LIP = (CS.base + rIP).

long mode

An operating mode unique to the AMD64 architecture. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Long mode has two submodes, *64-bit mode* and *compatibility mode*.

lsb

Least-significant bit.

LSB

Least-significant byte.

main memory

Physical memory, such as RAM and ROM (but not cache memory) that is installed in a particular computer system.

mask

(1) A control bit that prevents the occurrence of a floating-point exception from invoking an exception-handling routine. (2) A field of bits used for a control purpose.

MBZ

Must be zero. If software attempts to set an MBZ bit to 1 in a system register, a general-protection exception (#GP) occurs; if in a translation table entry, a reserved-bit page fault exception (#PF) will occur if the hardware attempts to use the entry for address translation. See *reserved*.

memory

Unless otherwise specified, *main memory*.

ModRM

A byte following an instruction opcode that specifies address calculation based on mode (Mod), register (R), and memory (M) variables.

moffset

A 16, 32, or 64-bit offset that specifies a memory operand directly, without using a ModRM or SIB byte.

msb

Most-significant bit.

MSB

Most-significant byte.

octword

Same as *double quadword*.

offset

Same as *displacement*.

overflow

The condition in which a floating-point number is larger in magnitude than the largest, finite, positive or negative number that can be represented in the data-type format being used.

PAE

Physical-address extensions.

physical memory

Actual memory, consisting of *main memory* and cache.

probe

A check for an address in a processor's caches or internal buffers. *External probes* originate outside the processor, and *internal probes* originate within the processor.

procedure stack

A portion of a stack segment in memory that is used to link procedures. Also known as a *program stack*.

program stack

See *procedure stack*.

protected mode

A submode of *legacy mode*.

quadword

Four words, or eight bytes, or 64 bits.

RAZ

Value returned on a read is always zero (0) regardless of what was previously written. See *reserved*.

real-address mode

See *real mode*.

real mode

A short name for *real-address mode*, a submode of *legacy mode*.

relative

Referencing with a displacement (also called offset) from an instruction pointer rather than the base of a code segment. Contrast with *absolute*.

reserved

Fields marked as reserved may be used at some future time.

To preserve compatibility with future processors, reserved fields require special handling when read or written by software. Software must not depend on the state of a reserved field (unless qualified as RAZ), nor upon the ability of such fields to return a previously written state.

If a field is marked reserved without qualification, software must not change the state of that field; it must reload that field with the same value returned from a prior read.

Reserved fields may be qualified as IGN, MBZ, RAZ, or SBZ (see definitions).

REX

An instruction prefix that specifies a 64-bit operand size and provides access to additional registers.

RIP-relative addressing

Addressing relative to the 64-bit RIP instruction pointer.

SBZ

Should be zero. An attempt by software to set an SBZ bit to 1 results in undefined behavior. See *reserved*.

shadow stack

A shadow stack is a separate, protected stack that is conceptually parallel to the procedure stack and used only by the shadow stack feature.

set

To write a bit value of 1. Compare *clear*.

SIB

A byte following an instruction opcode that specifies address calculation based on scale (S), index (I), and base (B).

SPA

System physical address. The address directly used to address system memory. Under SVM, also known as the host physical address. See *HPA*.

sticky bit

A bit that is set or cleared by hardware and that remains in that state until explicitly changed by software.

SVM

Secure virtual machine. AMD's virtualization architecture. SVM is defined in Chapter 15 on page 483.

System software

Privileged software that owns and manages the hardware resources of a system after initialization by *system firmware* and controls access to these resources. In a non-virtualized environment,

system software is provided by the operating system. In a virtualized environment, system software is largely equivalent to the virtual machine monitor (VMM), also commonly known as the *hypervisor*.

TOP

The x87 top-of-stack pointer.

TSS

Task-state segment.

underflow

The condition in which a floating-point number is smaller in magnitude than the smallest nonzero, positive or negative number that can be represented in the data-type format being used.

vector

(1) A set of integer or floating-point values, called *elements*, that are packed into a single data object. Most of the SSE and 64-bit media instructions use vectors as operands.

(2) An index into an interrupt descriptor table (IDT), used to access exception handlers. Compare *exception*.

virtual-8086 mode

A submode of *legacy mode*.

VMCB

Virtual machine control block.

VMM

Virtual machine monitor.

word

Two bytes, or 16 bits.

x86

See *legacy x86*.

Registers

In the following list of registers, the names are used to refer either to a given register or to the contents of that register:

AH–DH

The high 8-bit AH, BH, CH, and DH registers. Compare *AL–DL*.

AL–DL

The low 8-bit AL, BL, CL, and DL registers. Compare *AH–DH*.

AL–r15B

The low 8-bit AL, BL, CL, DL, SIL, DIL, BPL, SPL, and R8B–R15B registers, available in 64-bit mode.

BP

Base pointer register.

CR*n*

Control register number *n*.

CS

Code segment register.

eAX–eSP

The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers. Compare *rAX–rSP*.

EFER

Extended features enable register.

eFLAGS

16-bit or 32-bit flags register. Compare *rFLAGS*.

EFLAGS

32-bit (extended) flags register.

eIP

16-bit or 32-bit instruction-pointer register. Compare *rIP*.

EIP

32-bit (extended) instruction-pointer register.

FLAGS

16-bit flags register.

GDTR

Global descriptor table register.

GPRs

General-purpose registers. For the 16-bit data size, these are AX, BX, CX, DX, DI, SI, BP, and SP. For the 32-bit data size, these are EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP. For the 64-bit data size, these include RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and R8–R15.

IDTR

Interrupt descriptor table register.

IP

16-bit instruction-pointer register.

LDTR

Local descriptor table register.

MSR

Model-specific register.

r8–r15

The 8-bit R8B–R15B registers, or the 16-bit R8W–R15W registers, or the 32-bit R8D–R15D registers, or the 64-bit R8–R15 registers.

rAX–rSP

The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers, or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers, or the 64-bit RAX, RBX, RCX, RDX, RDI, RSI, RBP, and RSP registers. Replace the placeholder *r* with nothing for 16-bit size, “E” for 32-bit size, or “R” for 64-bit size.

RAX

64-bit version of the EAX register.

RBP

64-bit version of the EBP register.

RBX

64-bit version of the EBX register.

RCX

64-bit version of the ECX register.

RDI

64-bit version of the EDI register.

RDX

64-bit version of the EDX register.

rFLAGS

16-bit, 32-bit, or 64-bit flags register. Compare *RFLAGS*.

RFLAGS

64-bit flags register. Compare *rFLAGS*.

rIP

16-bit, 32-bit, or 64-bit instruction-pointer register. Compare *RIP*.

RIP
: 64-bit instruction-pointer register.

RSI
: 64-bit version of the ESI register.

RSP
: 64-bit version of the ESP register.

SP
: Stack pointer register.

SS
: Stack segment register.

SSP
: Shadow-stack pointer register.

TPR
: Task priority register (CR8), a new register introduced in the AMD64 architecture to speed interrupt management.

TR
: Task register.

YMM/XMM
: Set of sixteen (eight accessible in legacy and compatibility modes) 256-bit wide registers that hold scala and vector operands used by the SSE instructions.

Endian Order

The x86 and AMD64 architectures address memory using little-endian byte-ordering. Multibyte values are stored with their least-significant byte at the lowest byte address, and they are illustrated with their least significant byte at the right side. Strings are illustrated in reverse order, because the addresses of their bytes increase from right to left.

Related Documents

- Peter Abel, *IBM PC Assembly Language and Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Rakesh Agarwal, *80x86 Architecture & Programming: Volume II*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- AMD, *BIOS and Kernel Developer's Guide* (BKDG) for particular hardware implementations of older families of the AMD64 architecture.

- AMD, *Processor Programming Reference (PPR)* for particular hardware implementations of newer families of the AMD64 architecture.
- AMD, *AMD I/O Virtualization Technology (IOMMU) Specification*, Revision 2.2 or later; order number 48882.
- AMD, *Software Optimization Guide for AMD Family 15h Processors,* order number 47414.
- Don Anderson and Tom Shanley, *Pentium Processor System Architecture*, Addison-Wesley, New York, 1995.
- Nabajyoti Barkakati and Randall Hyde, *Microsoft Macro Assembler Bible*, Sams, Carmel, Indiana, 1992.
- Barry B. Brey, *8086/8088, 80286, 80386, and 80486 Assembly Language Programming*, Macmillan Publishing Co., New York, 1994.
- Barry B. Brey, *Programming the 80286, 80386, 80486, and Pentium Based Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Ralf Brown and Jim Kyle, *PC Interrupts,* Addison-Wesley, New York, 1994.
- Penn Brumm and Don Brumm, *80386/80486 Assembly Language Programming*, Windcrest McGraw-Hill, 1993.
- Geoff Chappell, *DOS Internals*, Addison-Wesley, New York, 1994.
- Chips and Technologies, Inc. *Super386 DX Programmer's Reference Manual*, Chips and Technologies, Inc., San Jose, 1992.
- John Crawford and Patrick Gelsinger, *Programming the 80386*, Sybex, San Francisco, 1987.
- Cyrix Corporation, *5x86 Processor BIOS Writer's Guide*, Cyrix Corporation, Richardson, TX, 1995.
- Cyrix Corporation, *M1 Processor Data Book*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor MMX Extension Opcode Table*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor Data Book*, Cyrix Corporation, Richardson, TX, 1997.
- Ray Duncan, *Extending DOS: A Programmer's Guide to Protected-Mode DOS*, Addison Wesley, NY, 1991.
- William B. Giles, *Assembly Language Programming for the Intel 80xxx Family*, Macmillan, New York, 1991.
- Frank van Gilluwe, *The Undocumented PC,* Addison-Wesley, New York, 1994.
- John L. Hennessy and David A. Patterson, *Computer Architecture*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.
- Thom Hogan, *The Programmer's PC Sourcebook*, Microsoft Press, Redmond, WA, 1991.
- Hal Katircioglu, *Inside the 486, Pentium, and Pentium Pro*, Peer-to-Peer Communications, Menlo Park, CA, 1997.
- IBM Corporation, *486SLC Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.

- IBM Corporation, *486SLC2 Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *80486DX2 Processor Floating Point Instructions*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *80486DX2 Processor BIOS Writer's Guide*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *Blue Lightning 486DX2 Data Book*, IBM Corporation, Essex Junction, VT, 1994.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, ANSI/IEEE Std 854-1987.
- Muhammad Ali Mazidi and Janice Gillispie Mazidi, *80X86 IBM PC and Compatible Computers*, Prentice-Hall, Englewood Cliffs, NJ, 1997.
- Hans-Peter Messmer, *The Indispensable Pentium Book*, Addison-Wesley, New York, 1995.
- Karen Miller, *An Assembly Language Introduction to Computer Architecture: Using the Intel Pentium*, Oxford University Press, New York, 1999.
- Stephen Morse, Eric Isaacson, and Douglas Albert, *The 80386/387 Architecture*, John Wiley & Sons, New York, 1987.
- NexGen Inc., *Nx586™ Processor Data Book*, NexGen Inc., Milpitas, CA, 1993.
- NexGen Inc., *Nx686™ Processor Data Book*, NexGen Inc., Milpitas, CA, 1994.
- Bipin Patwardhan, *Introduction to the Streaming SIMD Extensions in the Pentium® III*, www.x86.org/articles/sse_pt1/ simd1.htm, June, 2000.
- Peter Norton, Peter Aitken, and Richard Wilton, *PC Programmer's Bible*, Microsoft Press, Redmond, WA, 1993.
- *PharLap 386|ASM Reference Manual*, Pharlap, Cambridge MA, 1993.
- *PharLap TNT DOS-Extender Reference Manual*, Pharlap, Cambridge MA, 1995.
- Sen-Cuo Ro and Sheau-Chuen Her, *i386/i486 Advanced Programming*, Van Nostrand Reinhold, New York, 1993.
- Jeffrey P. Royer, *Introduction to Protected Mode Programming*, course materials for an onsite class, 1992.
- Tom Shanley, *Protected Mode System Architecture*, Addison Wesley, NY, 1996.
- SGS-Thomson Corporation, *80486DX Processor SMM Programming Manual*, SGS-Thomson Corporation, 1995.
- Walter A. Triebel, *The 80386DX Microprocessor*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- John Wharton, *The Complete x86*, MicroDesign Resources, Sebastopol, California, 1994.

1 System-Programming Overview

This entire volume is intended for system-software developers—programmers writing operating systems, loaders, linkers, device drivers, or utilities that require access to system resources. These system resources are generally available only to software running at the highest-privilege level (CPL=0), also referred to as *privileged software*. Privilege levels and their interactions are fully described in “Segment-Protection Overview” on page 104.

This chapter introduces the basic features and capabilities of the AMD64 architecture that are available to system-software developers. The concepts include:

- The supported address forms and how memory is organized.
- How memory-management hardware makes use of the various address forms to access memory.
- The processor operating modes, and how the memory-management hardware supports each of those modes.
- The system-control registers used to manage system resources.
- The interrupt and exception mechanism, and how it is used to interrupt program execution and to report errors.
- Additional, miscellaneous features available to system software, including support for hardware multitasking, reporting machine-check exceptions, debugging software problems, and optimizing software performance.

Many of the legacy features and capabilities are enhanced by the AMD64 architecture to support 64-bit operating systems and applications, while providing backward-compatibility with existing software.

1.1 Memory Model

The AMD64 architecture memory model is designed to allow system software to manage application software and associated data in a secure fashion. The memory model is backward-compatible with the legacy memory model. Hardware-translation mechanisms are provided to map addresses between virtual-memory space and physical-memory space. The translation mechanisms allow system software to relocate applications and data transparently, either anywhere in physical-memory space, or in areas on the system hard drive managed by the operating system.

In long mode, the AMD64 architecture implements a flat-memory model. In legacy mode, the architecture implements all legacy memory models.

1.1.1 Memory Addressing

The AMD64 architecture supports address relocation. To do this, several types of addresses are needed to completely describe memory organization. Specifically, four types of addresses are defined by the AMD64 architecture:

- Logical addresses
- Effective addresses, or segment offsets, which are a portion of the logical address.
- Linear (virtual) addresses
- Physical addresses

Logical Addresses. A *logical address* is a reference into a segmented-address space. It is comprised of the segment selector and the effective address. Notationally, a logical address is represented as

```
Logical Address = Segment Selector : Offset
```

The segment selector specifies an entry in either the global or local descriptor table. The specified descriptor-table entry describes the segment location in virtual-address space, its size, and other characteristics. The effective address is used as an offset into the segment specified by the selector.

Logical addresses are often referred to as *far pointers*. Far pointers are used in software addressing when the segment reference must be explicit (i.e., a reference to a segment outside the current segment).

Effective Addresses. The offset into a memory segment is referred to as an effective address (see “Segmentation” on page 5 for a description of segmented memory). Effective addresses are formed by adding together elements comprising a base value, a scaled-index value, and a displacement value. The effective-address computation is represented by the equation

```
Effective Address = Base + (Scale x Index) + Displacement
```

The elements of an effective-address computation are defined as follows:

- *Base*—A value stored in any general-purpose register.
- *Scale*—A positive value of 1, 2, 4, or 8.
- *Index*—A two’s-complement value stored in any general-purpose register.
- *Displacement*—An 8-bit, 16-bit, or 32-bit two’s-complement value encoded as part of the instruction.

Effective addresses are often referred to as *near pointers*. A near pointer is used when the segment selector is known implicitly or when the flat-memory model is used.

Long mode defines a 64-bit effective-address length. If a processor implementation does not support the full 64-bit virtual-address space, the effective address must be in *canonical form* (see “Canonical Address Form” on page 4).

Linear (Virtual) Addresses. The segment-selector portion of a logical address specifies a segment-descriptor entry in either the global or local descriptor table. The specified segment-descriptor entry contains the segment-base address, which is the starting location of the segment in linear-address space. A *linear address* is formed by adding the segment-base address to the effective address (segment offset), which creates a reference to any byte location within the supported linear-address space. Linear addresses are often referred to as *virtual addresses*, and both terms are used interchangeably throughout this document.

```
Linear Address = Segment Base Address + Effective Address
```

When the flat-memory model is used—as in 64-bit mode—a segment-base address is treated as 0. In this case, the linear address is identical to the effective address. In long mode, linear addresses must be in canonical address form, as described in “Canonical Address Form” on page 4.

Physical Addresses. A *physical address* is a reference into the physical-address space, typically main memory. Physical addresses are translated from virtual addresses using page-translation mechanisms. See “Paging” on page 7 for information on how the paging mechanism is used for virtual-address to physical-address translation. When the paging mechanism is not enabled, the virtual (linear) address is used as the physical address.

1.1.2 Memory Organization

The AMD64 architecture organizes memory into *virtual memory* and *physical memory*. Virtual-memory and physical-memory spaces can be (and usually are) different in size. Generally, the virtual-address space is much larger than physical-address memory. System software relocates applications and data between physical memory and the system hard disk to make it appear that much more memory is available than really exists. System software then uses the hardware memory-management mechanisms to map the larger virtual-address space into the smaller physical-address space.

Virtual Memory. Software uses virtual addresses to access locations within the virtual-memory space. System software is responsible for managing the relocation of applications and data in virtual-memory space using segment-memory management. System software is also responsible for mapping virtual memory to physical memory through the use of page translation. The AMD64 architecture supports different virtual-memory sizes using the following address-translation modes:

- *Protected Mode*—This mode supports 4 gigabytes of virtual-address space using 32-bit virtual addresses.
- *Long Mode*—This mode supports 16 exabytes of virtual-address space using 64-bit virtual addresses. A given implementation may however support less than this, as reported by the CPUID feature identification facility.

Physical Memory. Physical addresses are used to directly access main memory. For a particular computer system, the size of the *available* physical-address space is equal to the amount of main memory installed in the system. The maximum amount of physical memory accessible depends on the processor implementation and on the address-translation mode. The AMD64 architecture supports varying physical-memory sizes using the following address-translation modes:

- *Real-Address Mode*—This mode, also called *real mode*, supports 1 megabyte of physical-address space using 20-bit physical addresses. This address-translation mode is described in "Real Addressing" on page 10. Real mode is available only from legacy mode (see "Legacy Modes" on page 14).
- *Legacy Protected Mode*—This mode supports several different address-space sizes, depending on the translation mechanism used and whether extensions to those mechanisms are enabled.

  Legacy protected mode supports 4 gigabytes of physical-address space using 32-bit physical addresses. Both segment translation (see "Segmentation" on page 5) and page translation (see "Paging" on page 7) can be used to access the physical address space, when the processor is running in legacy protected mode.

  When the physical-address size extensions are enabled (see "Physical-Address Extensions (PAE) Bit" on page 132), the page-translation mechanism can be extended to support 52-bit physical addresses. 52-bit physical addresses allow up to 4 petabytes of physical-address space to be supported. (Currently, the AMD64 architecture supports 40-bit addresses in this mode, allowing up to 1 terabyte of physical-address space to be supported.
- *Long Mode*—This mode is unique to the AMD64 architecture. This mode supports up to 4 petabytes of physical-address space using 52-bit physical addresses. Long mode requires the use of page-translation and the physical-address size extensions (PAE).

1.1.3 Canonical Address Form

Long mode defines 64 bits of virtual-address space, but processor implementations can support less. Although some processor implementations do not use all 64 bits of the virtual address, they all check bits 63 through the most-significant implemented bit to see if those bits are all zeros or all ones. An address that complies with this property is in *canonical address form*. In most cases, a virtual-memory reference that is not in canonical form (in either the linear or effective form of the address) causes a general-protection exception (#GP) to occur. However, implied stack references where the stack address is not in canonical form causes a stack exception (#SS) to occur. Implied stack references include all push and pop instructions, and any instruction using RSP or RBP as a base register.

By checking canonical-address form, the AMD64 architecture prevents software from exploiting unused high bits of pointers for other purposes. Software complying with canonical-address form on a specific processor implementation can run unchanged on long-mode implementations supporting larger virtual-address spaces.

1.2 Memory Management

Memory management consists of the methods by which addresses generated by software are translated by segmentation and/or paging into addresses in physical memory. Memory management is not visible to application software. It is handled by the system software and processor hardware.

1.2.1 Segmentation

Segmentation was originally created as a method by which system software could isolate software processes (tasks), and the data used by those processes, from one another in an effort to increase the reliability of systems running multiple processes simultaneously.

The AMD64 architecture is designed to support all forms of legacy segmentation. However, most modern system software does not use the segmentation features available in the legacy x86 architecture. Instead, system software typically handles program and data isolation using page-level protection. For this reason, the AMD64 architecture dispenses with multiple segments in 64-bit mode and, instead, uses a flat-memory model. The elimination of segmentation allows new 64-bit system software to be coded more simply, and it supports more efficient management of multi-processing than is possible in the legacy x86 architecture.

Segmentation is, however, used in compatibility mode and legacy mode. Here, segmentation is a form of base memory-addressing that allows software and data to be relocated in virtual-address space off of an arbitrary base address. Software and data can be relocated in virtual-address space using one or more variable-sized *memory segments.* The legacy x86 architecture provides several methods of restricting access to segments from other segments so that software and data can be protected from interfering with each other.

In compatibility and legacy modes, up to 16,383 unique segments can be defined. The base-address value, segment size (called a *limit*), protection, and other attributes for each segment are contained in a data structure called a *segment descriptor*. Collections of segment descriptors are held in *descriptor tables*. Specific segment descriptors are referenced or selected from the descriptor table using a *segment selector register*. Six segment-selector registers are available, providing access to as many as six segments at a time.

Figure 1-1 on page 6 shows an example of segmented memory. Segmentation is described in Chapter 4, “Segmented Virtual Memory.”

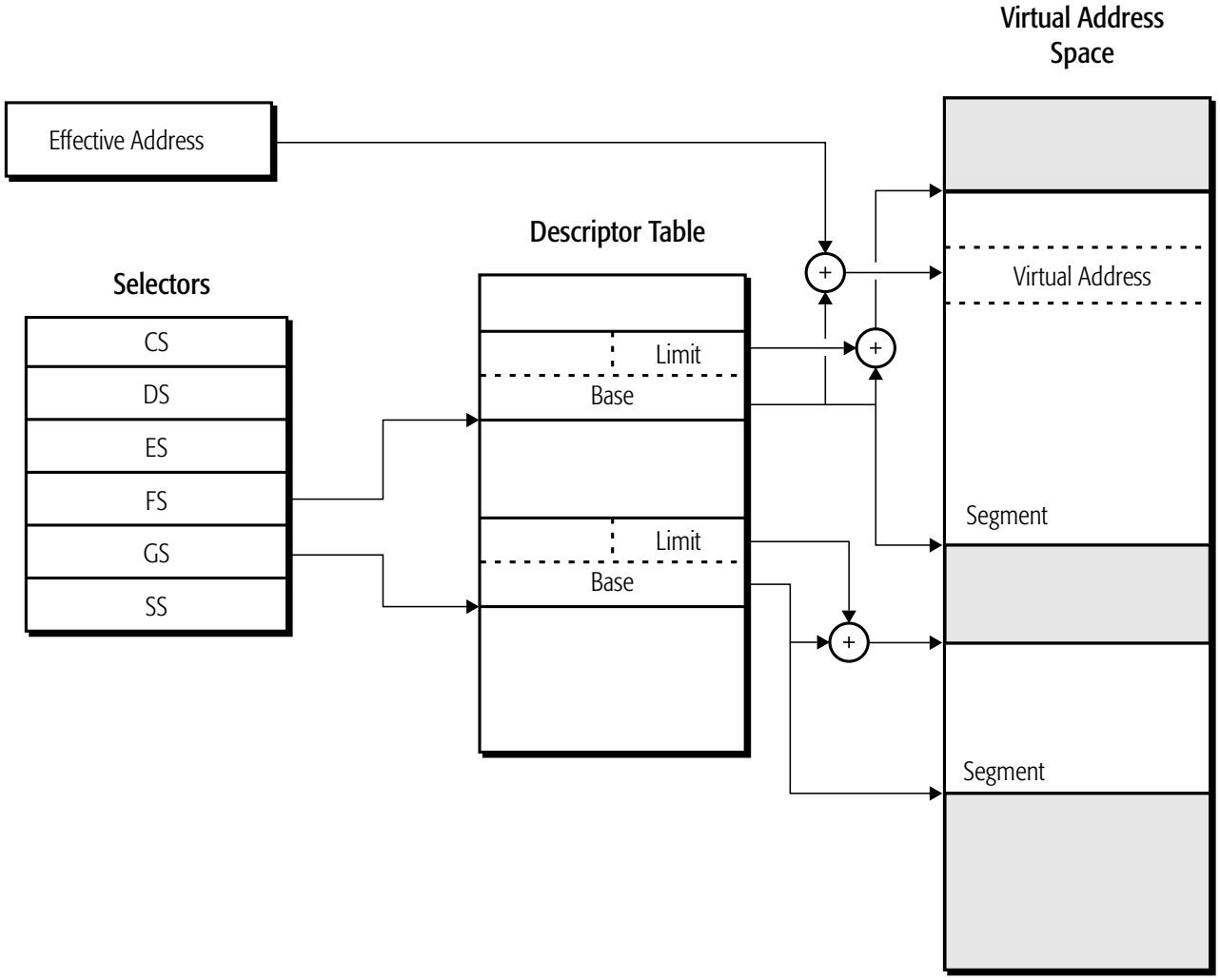


Figure 1-1. Segmented-Memory Model

Flat Segmentation. One special case of segmented memory is the flat-memory model. In the legacy flat-memory model, all segment-base addresses have a value of 0, and the segment limits are fixed at 4 Gbytes. Segmentation cannot be disabled but use of the flat-memory model effectively disables segment translation. The result is a virtual address that equals the effective address. Figure 1-2 on page 7 shows an example of the flat-memory model.

Software running in 64-bit mode automatically uses the flat-memory model. In 64-bit mode, the segment base is treated as if it were 0, and the segment limit is ignored. This allows an effective addresses to access the full virtual-address space supported by the processor.

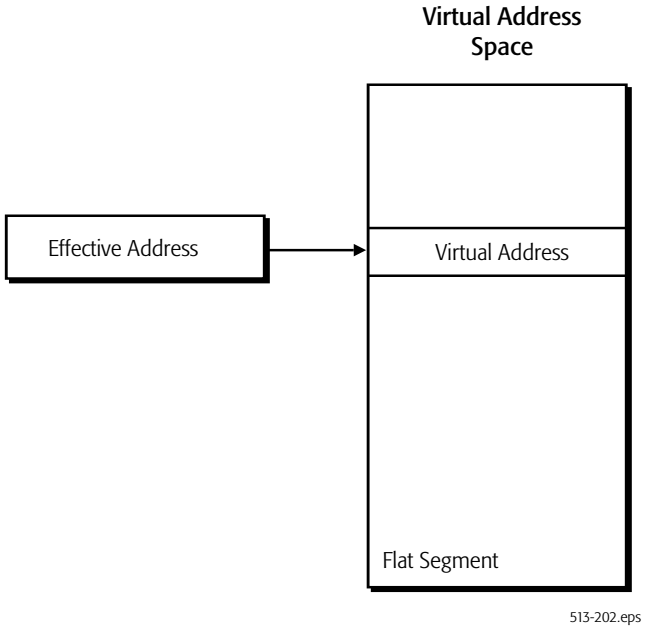


Figure 1-2. Flat Memory Model

1.2.2 Paging

Paging allows software and data to be relocated in physical-address space using fixed-size blocks called *physical pages*. The legacy x86 architecture supports three different physical-page sizes of 4 Kbytes, 2 Mbytes, and 4 Mbytes. As with segment translation, access to physical pages by lesser-privileged software can be restricted.

Page translation uses a hierarchical data structure called a page-translation table to translate virtual pages into physical-pages. The number of levels in the translation-table hierarchy can be as few as one or as many as four, depending on the physical-page size and processor operating mode. Translation tables are aligned on 4-Kbyte boundaries. Physical pages must be aligned on 4-Kbyte, 2-Mbyte, or 4-Mbyte boundaries, depending on the physical-page size.

Each table in the translation hierarchy is indexed by a portion of the virtual-address bits. The entry referenced by the table index contains a pointer to the base address of the next-lower-level table in the translation hierarchy. In the case of the lowest-level table, its entry points to the physical-page base address. The physical page is then indexed by the least-significant bits of the virtual address to yield the physical address.

Figure 1-3 on page 8 shows an example of paged memory with three levels in the translation-table hierarchy. Paging is described in Chapter 5, “Page Translation and Protection.”

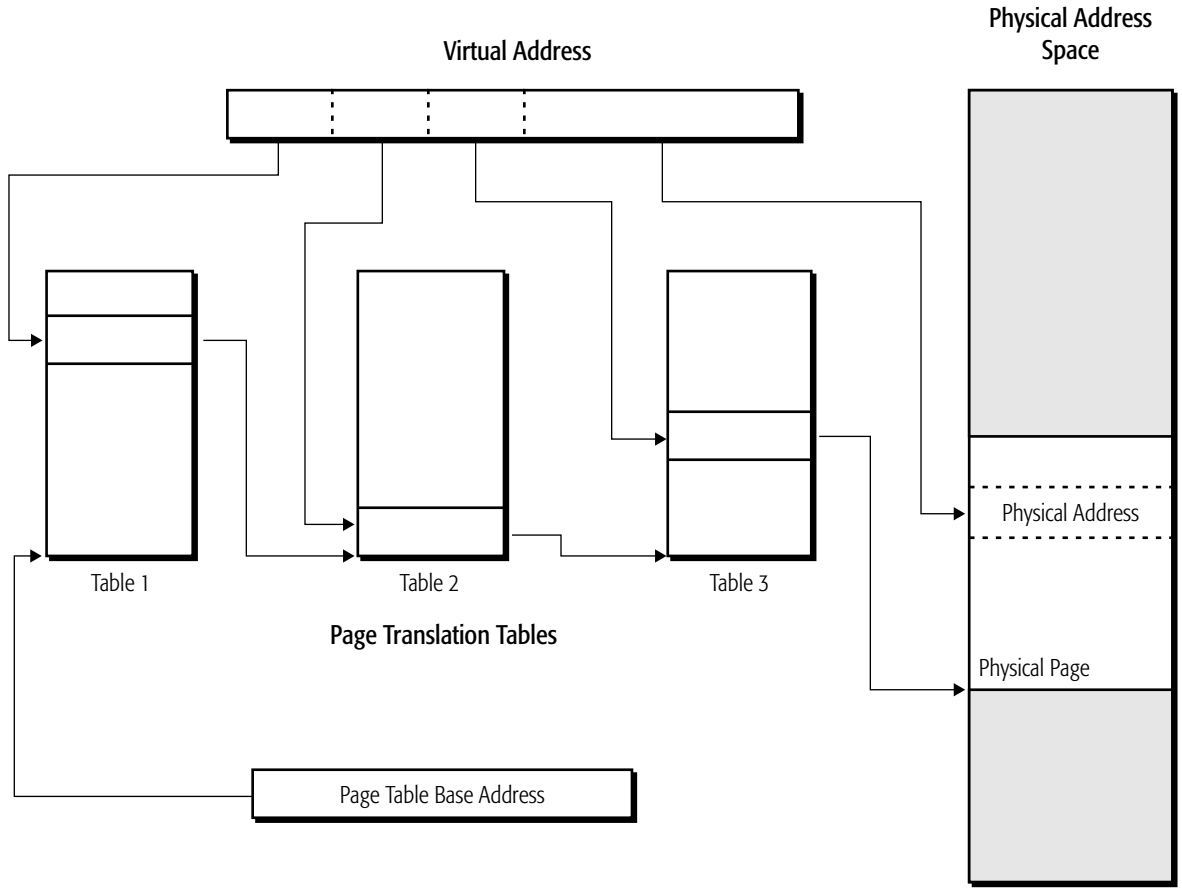


Figure 1-3. Paged Memory Model

Software running in long mode is required to have page translation enabled.

1.2.3 Mixing Segmentation and Paging

Memory-management software can combine the use of segmented memory and paged memory. Because segmentation cannot be disabled, paged-memory management requires some minimum initialization of the segmentation resources. Paging can be completely disabled, so segmented-memory management does not require initialization of the paging resources.

Segments can range in size from a single byte to 4 Gbytes in length. It is therefore possible to map multiple segments to a single physical page and to map multiple physical pages to a single segment. Alignment between segment and physical-page boundaries is not required, but memory-management software is simplified when segment and physical-page boundaries are aligned.

The simplest, most efficient method of memory management is the flat-memory model. In the flat-memory model, all segment base addresses have a value of 0 and the segment limits are fixed at 4 Gbytes. The segmentation mechanism is still used each time a memory reference is made, but because virtual addresses are identical to effective addresses in this model, the segmentation mechanism is effectively ignored. Translation of virtual (or effective) addresses to physical addresses takes place using the paging mechanism only.

Because 64-bit mode disables segmentation, it uses a flat, paged-memory model for memory management. The 4 Gbyte segment limit is ignored in 64-bit mode. Figure 1-4 shows an example of this model.

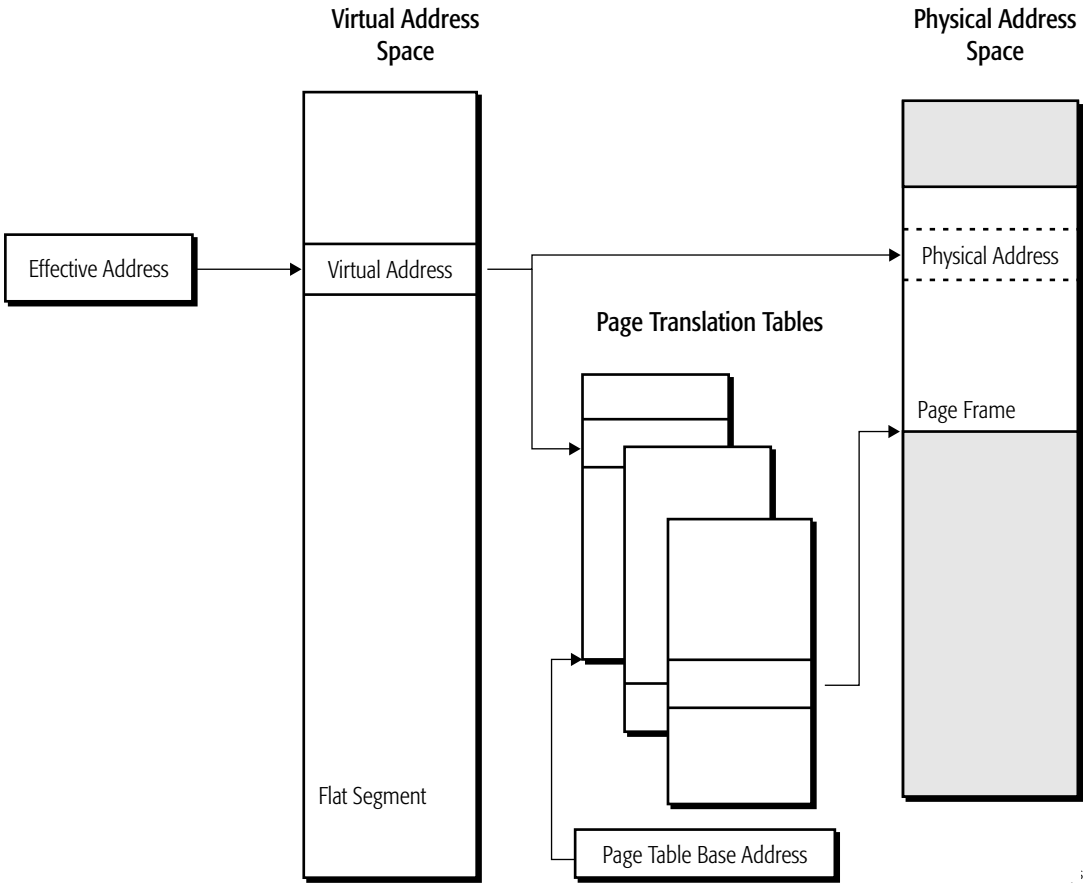


Figure 1-4. 64-Bit Flat, Paged-Memory Model

1.2.4 Real Addressing

Real addressing is a legacy-mode form of address translation used in real mode. This simplified form of address translation is backward compatible with 8086-processor effective-to-physical address translation. In this mode, 16-bit effective addresses are mapped to 20-bit physical addresses, providing a 1-Mbyte physical-address space.

Segment selectors are used in real-address translation, but not as an index into a descriptor table. Instead, the 16-bit segment-selector value is shifted left by 4 bits to form a 20-bit segment-base address. The 16-bit effective address is added to this 20-bit segment base address to yield a 20-bit physical address. If the sum of the segment base and effective address carries over into bit 20, that bit can be optionally truncated to mimic the 20-bit address wrapping of the 8086 processor by using the A20M# input signal to mask the A20 address bit.

A20 address bit masking should only be used in real mode (see next section for information on real mode). Use in other modes may result in address translation errors.

Real-address translation supports a 1-Mbyte physical-address space using up to 64K segments aligned on 16-byte boundaries. Each segment is exactly 64 Kbytes long. Figure 1-5 shows an example of real-address translation.

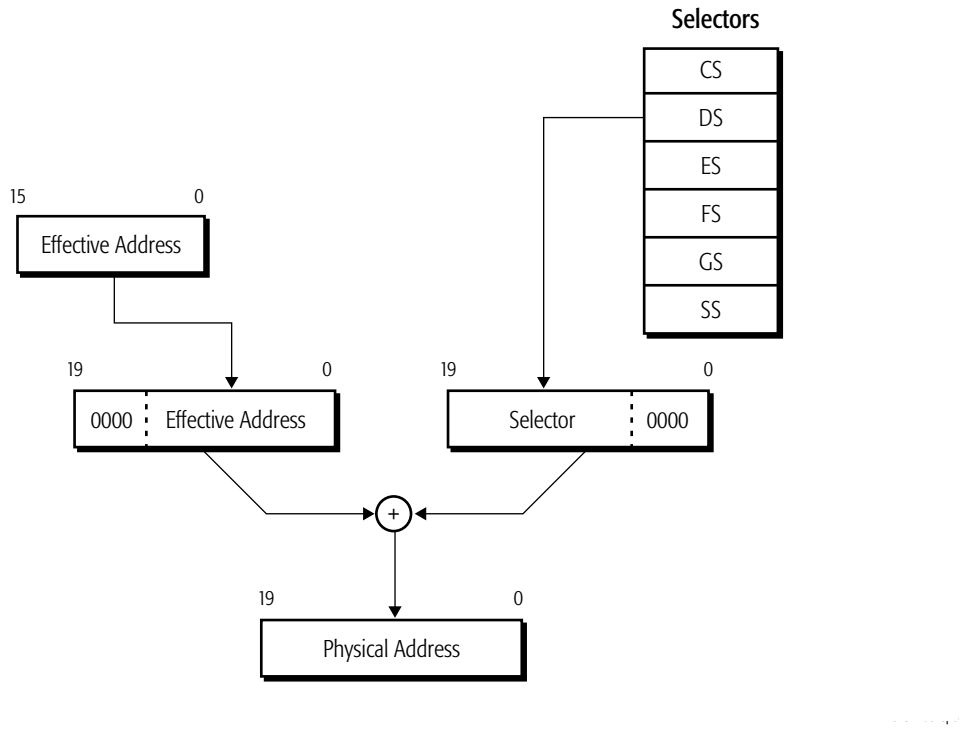


Figure 1-5. Real-Address Memory Model

1.3 Operating Modes

The legacy x86 architecture provides four operating modes or environments that support varying forms of memory management, virtual-memory and physical-memory sizes, and protection:

- Real Mode.
- Protected Mode.
- Virtual-8086 Mode.
- System Management Mode.

The AMD64 architecture supports all these legacy modes, and it adds a new operating mode called *long mode*. Table 1-1 shows the differences between long mode and legacy mode. Software can move between all supported operating modes as shown in Figure 1-6 on page 12. Each operating mode is described in the following sections.

Table 1-1. Operating Modes

Mode		System Software Required	Application Recompile Required	Defaults[1]		Register Extensions[2]	Maximum GPR Width (bits)
				Address Size (bits)	Operand Size (bits)		
Long Mode[3]	64-Bit Mode	New 64-bit OS	yes	64	32	yes	64
	Compatibility Mode		no	32		no	32
				16	16		
Legacy Mode	Protected Mode	Legacy 32-bit OS	no	32	32	no	32
				16	16		
	Virtual-8086 Mode			16	16		32
	Real Mode	Legacy 16-bit OS					

Note:

1. *Defaults can be overridden in most modes using an instruction prefix or system control bit.*
2. *Register extensions include access to the upper eight general-purpose and YMM/XMM registers, uniform access to lower 8 bits of all GPRs, and access to the upper 32 bits of the GPRs.*
3. *Long mode supports only x86 protected mode. It does not support x86 real mode or virtual-8086 mode.*

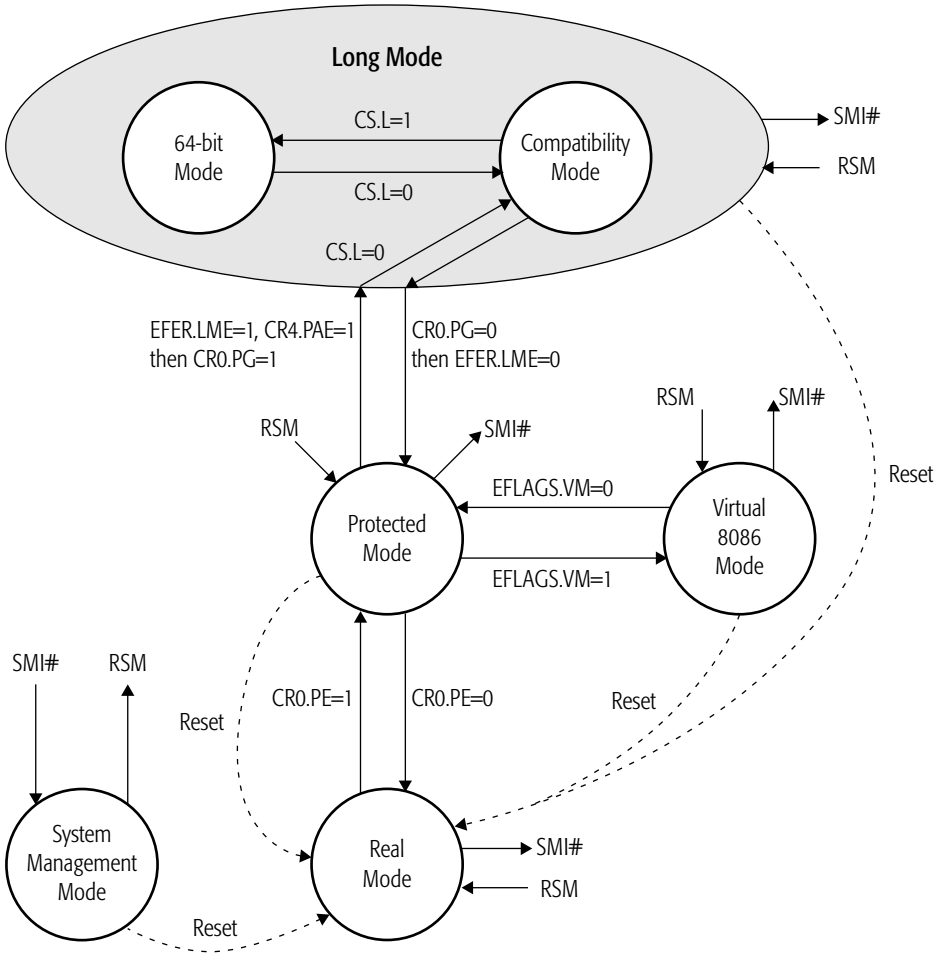


Figure 1-6. Operating Modes of the AMD64 Architecture

1.3.1 Long Mode

Long mode consists of two submodes: *64-bit mode* and *compatibility mode*. 64-bit mode supports several new features, including the ability to address 64-bit virtual-address space. Compatibility mode provides binary compatibility with existing 16-bit and 32-bit applications when running on 64-bit system software.

Throughout this document, references to *long mode* refer collectively to both *64-bit mode* and *compatibility mode*. If a function is specific to either 64-bit mode or compatibility mode, then those specific names are used instead of the name *long mode*.

Before enabling and activating long mode, system software must first enable protected mode. The process of enabling and activating long mode is described in Chapter 14, “Processor Initialization and

Long Mode Activation.” Long mode features are described throughout this document, where applicable.

1.3.2 64-Bit Mode

64-bit mode, a submode of long mode, provides support for 64-bit system software and applications by adding the following features:

- 64-bit virtual addresses (processor implementations can have fewer).
- Access to General Purpose Register bits 63:32
- Access to additional registers through the REX, VEX, and XOP instruction prefixes:
  - eight additional GPRs (R8–R15)
  - eight additional Streaming SIMD Extension (SSE) registers (YMM/XMM8–15)
- 64-bit instruction pointer (RIP).
- New RIP-relative data-addressing mode.
- Flat-segment address space with single code, data, and stack space.

The mode is enabled by the system software on an individual code-segment basis. Although code segments are used to enable and disable 64-bit mode, the legacy segmentation mechanism is largely disabled. Page translation is required for memory management purposes. Because 64-bit mode supports a 64-bit virtual-address space, it requires 64-bit system software and development tools.

In 64-bit mode, the default address size is 64 bits, and the default operand size is 32 bits. The defaults can be overridden on an instruction-by-instruction basis using instruction prefixes. A new REX prefix is introduced for specifying a 64-bit operand size and the new registers.

1.3.3 Compatibility Mode

Compatibility mode, a submode of long mode, allows system software to implement binary compatibility with existing 16-bit and 32-bit x86 applications. It allows these applications to run, without recompilation, under 64-bit system software in long mode, as shown in Table 1-1 on page 11.

In compatibility mode, applications can only access the first 4 Gbytes of virtual-address space. Standard x86 instruction prefixes toggle between 16-bit and 32-bit address and operand sizes.

Compatibility mode, like 64-bit mode, is enabled by system software on an individual code-segment basis. Unlike 64-bit mode, however, segmentation functions the same as in the legacy-x86 architecture, using 16-bit or 32-bit protected-mode semantics. From an application viewpoint, compatibility mode looks like a legacy protected-mode environment. From a system-software viewpoint, the long-mode mechanisms are used for address translation, interrupt and exception handling, and system data-structures.

1.3.4 Legacy Modes

L*egacy mode* consists of three submodes: real mode, protected mode, and virtual-8086 mode. Protected mode can be either paged or unpaged. L*egacy mode* preserves binary compatibility not only with existing x86 16-bit and 32-bit applications but also with existing x86 16-bit and 32-bit system software.

Real Mode. In this mode, also called real-address mode, the processor supports a physical-memory space of 1 Mbyte and operand sizes of 16 bits (default) or 32 bits (with instruction prefixes). Interrupt handling and address generation are nearly identical to the 80286 processor's real mode. Paging is not supported. All software runs at privilege level 0.

Real mode is entered after reset or processor power-up. The mode is not supported when the processor is operating in long mode because long mode requires that paged protected mode be enabled.

Protected Mode. In this mode, the processor supports virtual-memory and physical-memory spaces of 4 Gbytes and operand sizes of 16 or 32 bits. All segment translation, segment protection, and hardware multitasking functions are available. System software can use segmentation to relocate effective addresses in virtual-address space. If paging is not enabled, virtual addresses are equal to physical addresses. Paging can be optionally enabled to allow translation of virtual addresses to physical addresses and to use the page-based memory-protection mechanisms.

In protected mode, software runs at privilege levels 0, 1, 2, or 3. Typically, application software runs at privilege level 3, the system software runs at privilege levels 0 and 1, and privilege level 2 is available to system software for other uses. The 16-bit version of this mode was first introduced in the 80286 processor.

Virtual-8086 Mode. Virtual-8086 mode allows system software to run 16-bit real-mode software on a virtualized-8086 processor. In this mode, software written for the 8086, 8088, 80186, or 80188 processor can run as a privilege-level-3 task under protected mode. The processor supports a virtual-memory space of 1 Mbytes and operand sizes of 16 bits (default) or 32 bits (with instruction prefixes), and it uses real-mode address translation.

Virtual-8086 mode is enabled by setting the virtual-machine bit in the EFLAGS register (EFLAGS.VM). EFLAGS.VM can only be set or cleared when the EFLAGS register is loaded from the TSS as a result of a task switch, or by executing an IRET instruction from privileged software. The POPF instruction cannot be used to set or clear the EFLAGS.VM bit.

Virtual-8086 mode is not supported when the processor is operating in long mode. When long mode is enabled, any attempt to enable virtual-8086 mode is silently ignored.

1.3.5 System Management Mode (SMM)

System management mode (SMM) is an operating mode designed for system-control activities that are typically transparent to conventional system software. Power management is one popular use for system management mode. SMM is primarily targeted for use by platform firmware and specialized low-level device drivers. The code and data for SMM are stored in the SMM memory area, which is isolated from main memory by the SMM output signal.

SMM is entered by way of a system management interrupt (SMI). Upon recognizing an SMI, the processor enters SMM and switches to a separate address space where the SMM handler is located and executes. In SMM, the processor supports real-mode addressing with 4 Gbyte segment limits and default operand, address, and stack sizes of 16 bits (prefixes can be used to override these defaults).

1.4 System Registers

Figure 1-7 on page 16 shows the system registers defined for the AMD64 architecture. System software uses these registers to, among other things, manage the processor operating environment, define system resource characteristics, and to monitor software execution. With the exception of the RFLAGS register, system registers can be read and written only from privileged software.

Except for the descriptor-table registers and task register, the AMD64 architecture defines all system registers to be 64 bits wide. The descriptor table and task registers are defined by the AMD64 architecture to include 64-bit base-address fields, in addition to their other fields.

As shown in Figure 1-7 on page 16, the system registers include:

- *Control Registers*—These registers are used to control system operation and some system features. See “System-Control Registers” on page 41 for details.
- *System-Flags Register*—The RFLAGS register contains system-status flags and masks. It is also used to enable virtual-8086 mode and to control application access to I/O devices and interrupts. See “RFLAGS Register” on page 52 for details.
- *Descriptor-Table Registers*—These registers contain the location and size of descriptor tables stored in memory. Descriptor tables hold segmentation data structures used in protected mode. See “Descriptor Tables” on page 82 for details.
- *Task Register*—The task register contains the location and size in memory of the task-state segment. The hardware-multitasking mechanism uses the task-state segment to hold state information for a given task. The TSS also holds other data, such as the inner-level stack pointers used when changing to a higher privilege level. See “Task Register” on page 365 for details.
- *Debug Registers*—Debug registers are used to control the software-debug mechanism, and to report information back to a debug utility or application. See “Debug Registers” on page 386 for details.

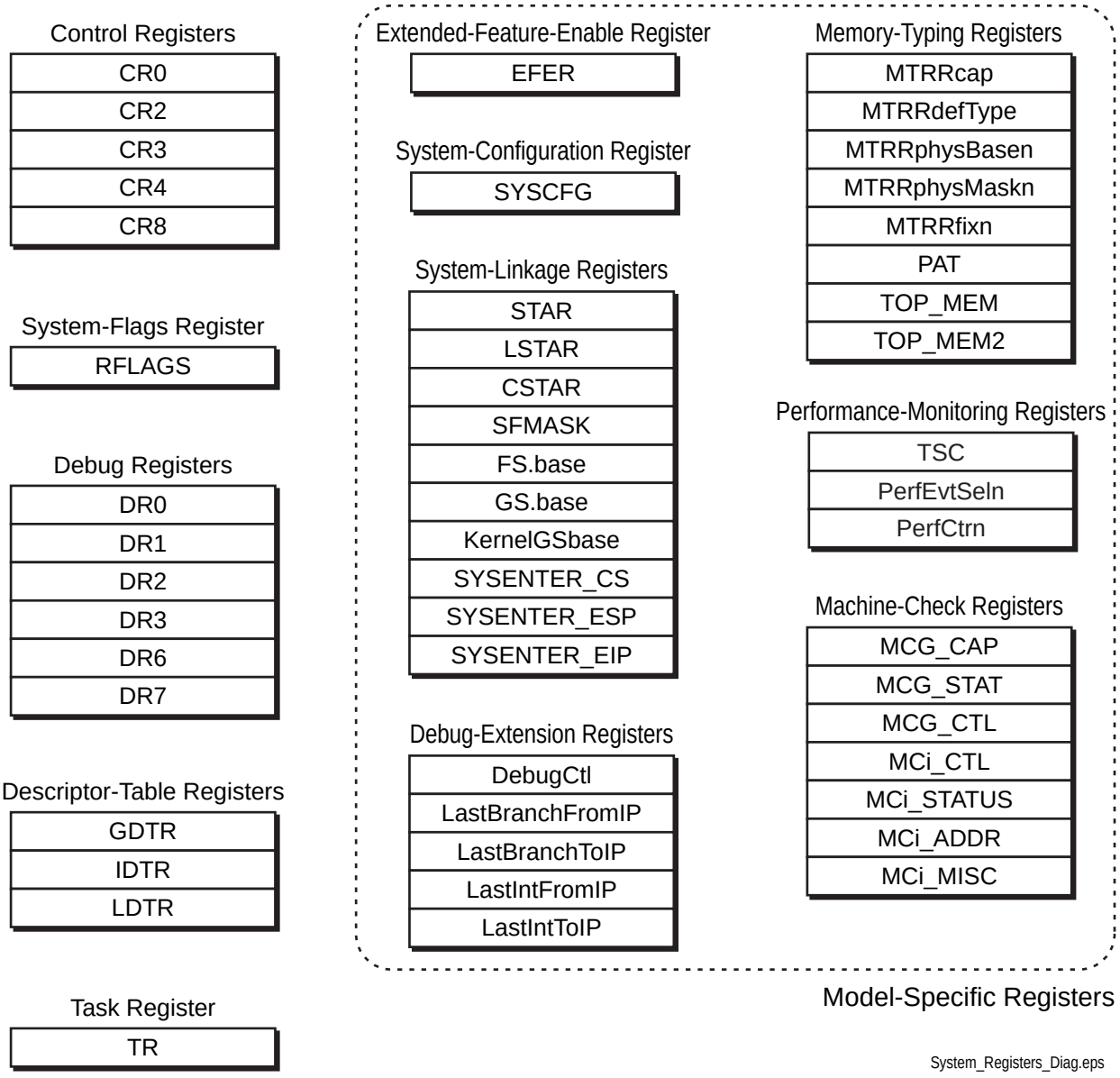


Figure 1-7. System Registers

Also defined as system registers are a number of *model-specific registers* included in the AMD64 architectural definition, and shown in Figure 1-7:

- *Extended-Feature-Enable Register*—The EFER register is used to enable and report status on special features not controlled by the CR*n* control registers. In particular, EFER is used to control activation of long mode. See “Extended Feature Enable Register (EFER)” on page 56 for more information.

- *System-Configuration Register*—The SYSCFG register is used to enable and configure system-bus features. See “System Configuration Register (SYSCFG)” on page 61 for more information.
- *System-Linkage Registers*—These registers are used by system-linkage instructions to specify operating-system entry points, stack locations, and pointers into system-data structures. See “Fast System Call and Return” on page 170 for details.
- *Memory-Typing Registers*—Memory-typing registers can be used to characterize (type) system memory. Typing memory gives system software control over how instructions and data are cached, and how memory reads and writes are ordered. See “MTRRs” on page 209 for details.
- *Debug-Extension Registers*—These registers control additional software-debug reporting features. See “Debug Registers” on page 386 for details.
- *Performance-Monitoring Registers*—Performance-monitoring registers are used to count processor and system events, or the duration of events. See “Performance Monitoring Counters” on page 400 for more information.
- *Machine-Check Registers*—The machine-check registers control the response of the processor to non-recoverable failures. They are also used to report information on such failures back to system utilities designed to respond to such failures. See “Machine Check Architecture MSRs” on page 291 for more information.

1.5 System-Data Structures

Figure 1-8 on page 18 shows the system-data structures defined for the AMD64 architecture. System-data structures are created and maintained by system software for use by the processor when running in protected mode. A processor running in protected mode uses these data structures to manage memory and protection, and to store program-state information when an interrupt or task switch occurs.

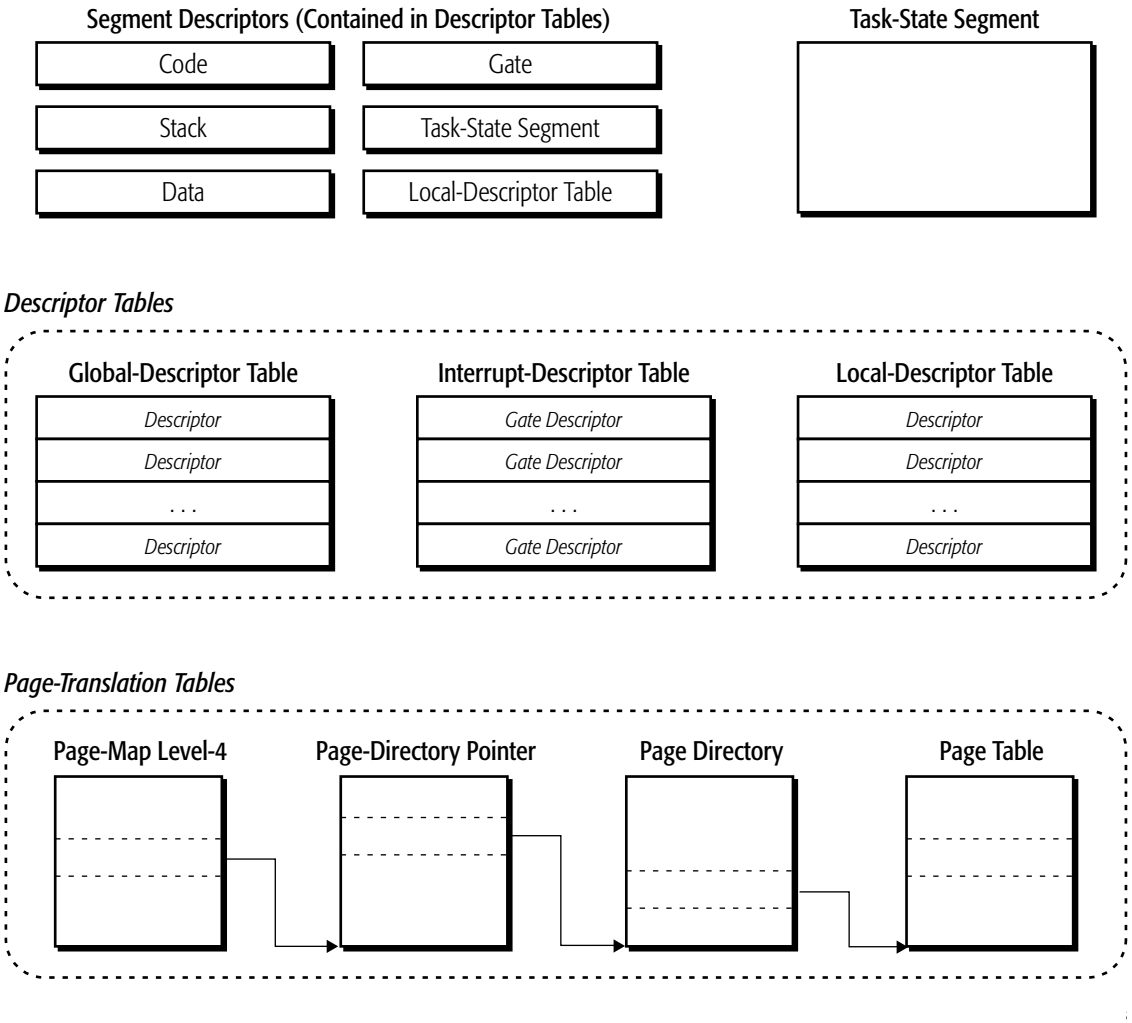


Figure 1-8. System-Data Structures

As shown in Figure 1-8, the system-data structures include:

- *Descriptors*—A descriptor provides information about a segment to the processor, such as its location, size and privilege level. A special type of descriptor, called a *gate*, is used to provide a code selector and entry point for a software routine. Any number of descriptors can be defined, but system software must at a minimum create a descriptor for the currently executing code segment and stack segment. See “Legacy Segment Descriptors” on page 88, and “Long-Mode Segment Descriptors” on page 97 for complete information on descriptors.
- *Descriptor Tables*—As the name implies, descriptor tables hold descriptors. The global-descriptor table holds descriptors available to all programs, while a local-descriptor table holds descriptors used by a single program. The interrupt-descriptor table holds only gate descriptors used by

interrupt handlers. System software must initialize the global-descriptor and interrupt-descriptor tables, while use of the local-descriptor table is optional. See “Descriptor Tables” on page 82 for more information.

- *Task-State Segment*—The task-state segment is a special segment for holding processor-state information for a specific program, or task. It also contains the stack pointers used when switching to more-privileged programs. The hardware multitasking mechanism uses the state information in the segment when suspending and resuming a task. Calls and interrupts that switch stacks cause the stack pointers to be read from the task-state segment. System software must create at least one task-state segment, even if hardware multitasking is not used. See “Legacy Task-State Segment” on page 367, and “64-Bit Task State Segment” on page 371 for details.
- *Page-Translation Tables*—Use of page translation is optional in protected mode, but it is required in long mode. A four-level page-translation data structure is provided to allow long-mode operating systems to translate a 64-bit virtual-address space into a 52-bit physical-address space. Legacy protected mode can use two- or three-level page-translation data structures. See “Page Translation Overview” on page 129 for more information on page translation.

1.6 Interrupts

The AMD64 architecture provides a mechanism for the processor to automatically suspend (interrupt) software execution and transfer control to an interrupt handler when an interrupt or exception occurs. An interrupt handler is privileged software designed to identify and respond to the cause of an interrupt or exception, and return control back to the interrupted software. *Interrupts* can be caused when system hardware signals an interrupt condition using one of the external-interrupt signals on the processor. Interrupts can also be caused by software that executes an interrupt instruction. *Exceptions* occur when the processor detects an abnormal condition as a result of executing an instruction. The term “interrupts” as used throughout this volume includes both interrupts and exceptions when the distinction is unnecessary.

System software not only sets up the interrupt handlers, but it must also create and initialize the data structures the processor uses to execute an interrupt handler when an interrupt occurs. The data structures include the code-segment descriptors for the interrupt-handler software and any data-segment descriptors for data and stack accesses. Interrupt-gate descriptors must also be supplied. Interrupt gates point to interrupt-handler code-segment descriptors, and the entry point in an interrupt handler. Interrupt gates are stored in the interrupt-descriptor table. The code-segment and data-segment descriptors are stored in the global-descriptor table and, optionally, the local-descriptor table.

When an interrupt occurs, the processor uses the interrupt vector to find the appropriate interrupt gate in the interrupt-descriptor table. The gate points to the interrupt-handler code segment and entry point, and the processor transfers control to that location. Before invoking the interrupt handler, the processor saves information required to return to the interrupted program. For details on how the processor transfers control to interrupt handlers, see “Legacy Protected-Mode Interrupt Control Transfers” on page 261, and “Long-Mode Interrupt Control Transfers” on page 272.

Table 1-2 shows the supported interrupts and exceptions, ordered by their vector number. Refer to “Vectors” on page 236 for a complete description of each interrupt, and a description of the interrupt mechanism.

Table 1-2. Interrupts and Exceptions

Vector	Description
0	Integer Divide-by-Zero Exception
1	Debug Exception
2	Non-Maskable-Interrupt
3	Breakpoint Exception (INT 3)
4	Overflow Exception (INTO instruction)
5	Bound-Range Exception (BOUND instruction)
6	Invalid-Opcode Exception
7	Device-Not-Available Exception
8	Double-Fault Exception
9	Coprocessor-Segment-Overrun Exception (reserved in AMD64)
10	Invalid-TSS Exception
11	Segment-Not-Present Exception
12	Stack Exception
13	General-Protection Exception
14	Page-Fault Exception
15	(Reserved)
16	x87 Floating-Point Exception
17	Alignment-Check Exception
18	Machine-Check Exception
19	SIMD Floating-Point Exception
21	Control-Protection Exception
0–255	Interrupt Instructions
0–255	Hardware Maskable Interrupts

1.7 Additional System-Programming Facilities

1.7.1 Hardware Multitasking

A task is any program that the processor can execute, suspend, and later resume executing at the point of suspension. During the time a task is suspended, other tasks are allowed to execute. Each task has its own execution space, consisting of a code segment, data segments, and a stack segment for each privilege level. Tasks can also have their own virtual-memory environment managed by the page-translation mechanism. The state information defining this execution space is stored in the task-state segment (TSS) maintained for each task.

Support for hardware multitasking is provided by implementations of the AMD64 architecture when software is running in legacy mode. Hardware multitasking provides automated mechanisms for switching tasks, saving the execution state of the suspended task, and restoring the execution state of the resumed task. When hardware multitasking is used to switch tasks, the processor takes the following actions:

- The processor automatically suspends execution of the task, allowing any executing instructions to complete and save their results.
- The execution state of a task is saved in the task TSS.
- The execution state of a new task is loaded into the processor from its TSS.
- The processor begins executing the new task at the location specified in the new task TSS.

Use of hardware-multitasking features is optional in legacy mode. Generally, modern operating systems do not use the hardware-multitasking features, and instead perform task management entirely in software. Long mode does not support hardware multitasking at all.

Whether hardware multitasking is used or not, system software must create and initialize at least one task-state segment data-structure. This requirement holds for both long-mode and legacy-mode software. The single task-state segment holds critical pieces of the task execution environment and is referenced during certain control transfers.

Detailed information on hardware multitasking is available in Chapter 12, “Task Management,” along with a full description of the requirements that must be met in initializing a task-state segment when hardware multitasking is not used.

1.7.2 Machine Check

Implementations of the AMD64 architecture support the machine-check exception. This exception is useful in system applications with stringent requirements for reliability, availability, and serviceability. The exception allows specialized system-software utilities to report hardware errors that are generally severe and non-recoverable. Providing the capability to report such errors can allow complex system problems to be pinpointed rapidly.

The machine-check exception is described in Chapter 9, “Machine Check Architecture.” Much of the error-reporting capabilities is implementation dependent. For more information, developers of machine-check error-reporting software should refer to the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* or applicable to your product.

1.7.3 Software Debugging

A software-debugging mechanism is provided in hardware to help software developers quickly isolate programming errors. This capability can be used to debug system software and application software alike. Only privileged software can access the debugging facilities. Generally, software-debug support is provided by a privileged application program rather than by the operating system itself.

The facilities supported by the AMD64 architecture allow debugging software to perform the following:

- Set breakpoints on specific instructions within a program.
- Set breakpoints on an instruction-address match.
- Set breakpoints on a data-address match.
- Set breakpoints on specific I/O-port addresses.
- Set breakpoints to occur on task switches when hardware multitasking is used.
- Single step an application instruction-by-instruction.
- Single step only branches and interrupts.
- Record a history of branches and interrupts taken by a program.

The debugging facilities are fully described in “Software-Debug Resources” on page 386. Some processors provide additional, implementation-specific debug support. For more information, refer to the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

1.7.4 Performance Monitoring

For many software developers, the ability to identify and eliminate performance bottlenecks from a program is nearly as important as quickly isolating programming errors. Implementations of the AMD64 architecture provide hardware performance-monitoring resources that can be used by special software applications to identify such bottlenecks. Non-privileged software can access the performance monitoring facilities, but only if privileged software grants that access.

The performance-monitoring facilities allow the counting of events, or the duration of events. Performance-analysis software can use the data to calculate the frequency of certain events, or the time spent performing specific activities. That information can be used to suggest areas for improvement and the types of optimizations that are helpful.

The performance-monitoring facilities are fully described in “Performance Monitoring Counters” on page 400. The specific events that can be monitored are generally implementation specific. For more information, refer to the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

2 x86 and AMD64 Architecture Differences

The AMD64 architecture is designed to provide full binary compatibility with all previous AMD implementations of the x86 architecture. This chapter summarizes the new features and architectural enhancements introduced by the AMD64 architecture, and compares those features and enhancements with previous AMD x86 processors. Most of the new capabilities introduced by the AMD64 architecture are available only in long mode (64-bit mode, compatibility mode, or both). However, some of the new capabilities are also available in legacy mode, and are mentioned where appropriate.

The material throughout this chapter assumes the reader has a solid understanding of the x86 architecture. For those who are unfamiliar with the x86 architecture, please read the remainder of this volume before reading this chapter.

2.1 Operating Modes

See “Operating Modes” on page 11 for a complete description of the operating modes supported by the AMD64 architecture.

2.1.1 Long Mode

The AMD64 architecture introduces long mode and its two sub-modes: 64-bit mode and compatibility mode.

64-Bit Mode. 64-bit mode provides full support for 64-bit system software and applications. The new features introduced in support of 64-bit mode are summarized throughout this chapter. To use 64-bit mode, a 64-bit operating system and tool chain are required.

Compatibility Mode. Compatibility mode allows 64-bit operating systems to implement binary compatibility with existing 16-bit and 32-bit x86 applications. It allows these applications to run, without recompilation, under control of a 64-bit operating system in long mode. The architectural enhancements introduced by the AMD64 architecture that support compatibility mode are summarized throughout this chapter.

Unsupported Modes. Long mode does not support the following two operating modes:

- *Virtual-8086 Mode*—The virtual-8086 mode bit (EFLAGS.VM) is ignored when the processor is running in long mode. When long mode is enabled, any attempt to enable virtual-8086 mode is silently ignored. System software must leave long mode in order to use virtual-8086 mode.
- *Real Mode*—Real mode is not supported when the processor is operating in long mode because long mode requires that protected mode be enabled.

2.1.2 Legacy Mode

The AMD64 architecture supports a pure x86 legacy mode, which preserves binary compatibility not only with existing 16-bit and 32-bit applications but also with existing 16-bit and 32-bit operating

systems. L*egacy mode* supports real mode, protected mode, and virtual-8086 mode. A reset always places the processor in legacy mode (real mode), and the processor continues to run in legacy mode until system software activates long mode. New features added by the AMD64 architecture that are supported in legacy mode are summarized in this chapter.

2.1.3 System-Management Mode

The AMD64 architecture supports system-management mode (SMM). SMM can be entered from both long mode and legacy mode, and SMM can return directly to either mode. The following differences exist between the support of SMM in the AMD64 architecture and the SMM support found in previous processor generations:

- The SMRAM state-save area format is changed to hold the 64-bit processor state. This state-save area format is used regardless of whether SMM is entered from long mode or legacy mode.
- The auto-halt restart and I/O-instruction restart entries in the SMRAM state-save area are one byte instead of two bytes.
- The initial processor state upon entering SMM is expanded to reflect the 64-bit nature of the processor.
- New conditions exist that can cause a processor shutdown while exiting SMM.
- SMRAM caching considerations are modified because the legacy FLUSH# external signal (writeback, if modified, and invalidate) is not supported on implementations of the AMD64 architecture.

See Chapter 10, “System-Management Mode,” for more information on the SMM differences.

2.2 Memory Model

The AMD64 architecture provides enhancements to the legacy memory model to support very large physical-memory and virtual-memory spaces while in long mode. Some of this expanded support for physical memory is available in legacy mode.

2.2.1 Memory Addressing

Virtual-Memory Addressing. Virtual-memory support is expanded to 64 address bits in long mode. This allows up to 16 exabytes of virtual-address space to be accessed. The virtual-address space supported in legacy mode is unchanged.

Physical-Memory Addressing. Physical-memory support is expanded to 52 address bits in long mode and legacy mode. This allows up to 4 petabytes of physical memory to be accessed. The expanded physical-memory support is achieved by using paging and the page-size extensions.

Note that given processor may implement less than the architecturally-defined physical address size of 52 bits.

Effective Addressing. The effective-address length is expanded to 64 bits in long mode. An effective-address calculation uses 64-bit base and index registers, and sign-extends 8-bit and 32-bit displacements to 64 bits. In legacy mode, effective addresses remain 32 bits long.

2.2.2 Page Translation

The AMD64 architecture defines an expanded page-translation mechanism supporting translation of a 64-bit virtual address to a 52-bit physical address. See “Long-Mode Page Translation” on page 141 for detailed information on the enhancements to page translation in the AMD64 architecture. The enhancements are summarized below.

Physical-Address Extensions (PAE). The AMD64 architecture requires physical-address extensions to be enabled (CR4.PAE=1) before long mode is entered. When PAE is enabled, all paging data-structures are 64 bits, allowing references into the full 52-bit physical-address space supported by the architecture.

Page-Size Extensions (PSE). Page-size extensions (CR4.PSE) are ignored in long mode. Long mode does not support the 4-Mbyte page size enabled by page-size extensions. Long mode does, however, support 4-Kbyte and 2-Mbyte page sizes.

Paging Data Structures. The AMD64 architecture extends the page-translation data structures in support of long mode. The extensions are:

- *Page-map level-4 (PML4)*—Long mode defines a new page-translation data structure, the PML4 table. The PML4 table sits at the top of the page-translation hierarchy and references PDP tables.
- *Page-directory pointer (PDP)*—The PDP tables in long mode are expanded from 4 entries to 512 entries each.
- *Page-directory pointer entry (PDPE)*—Previously undefined fields within the legacy-mode PDPE are defined by the AMD64 architecture.

CR3 Register. The CR3 register is expanded to 64 bits for use in long-mode page translation. When long mode is active, the CR3 register references the base address of the PML4 table. In legacy mode, the upper 32 bits of CR3 are masked by the processor to support legacy page translation. CR3 references the PDP base-address when physical-address extensions are enabled, or the page-directory table base-address when physical-address extensions are disabled.

Legacy-Mode Enhancements. Legacy-mode software can take advantage of the enhancements made to the physical-address extension (PAE) support and page-size extension (PSE) support. The four-level page translation mechanism introduced by long mode is not available to legacy-mode software.

- *PAE*—When physical-address extensions are enabled (CR4.PAE=1), the AMD64 architecture allows legacy-mode software to load up to 52-bit (maximum size) physical addresses into the PDE and PTE. Note that addresses are expanded to the maximum physical address size supported by the implementation.

- *PSE*—The use of page-size extensions allows legacy mode software to define 4-Mbyte pages using the 32-bit page-translation tables. When page-size extensions are enabled (CR4.PSE=1), the AMD64 architecture enhances the 4-Mbyte PDE to support 40 physical-address bits.

See "Legacy-Mode Page Translation" on page 133 for more information on these enhancements.

2.2.3 Segmentation

In long mode, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode:

- In compatibility mode, segmentation functions just as it does in legacy mode, using legacy 16-bit or 32-bit protected mode semantics.
- 64-bit mode requires a flat-memory model for creating a flat 64-bit virtual-address space. Much of the segmentation capability present in legacy mode and compatibility mode is disabled when the processor is running in 64-bit mode.

The differences in the segmentation model as defined by the AMD64 architecture are summarized in the following sections. See Chapter 4, "Segmented Virtual Memory," for a thorough description of these differences.

Descriptor-Table Registers. In long mode, the base-address portion of the descriptor-table registers (GDTR, IDTR, LDTR, and TR) are expanded to 64 bits. The full 64-bit base address can only be loaded by software when the processor is running in 64-bit mode (using the LGDT, LIDT, LLDT, and LTR instructions, respectively). However, the full 64-bit base address is *used* by a processor running in compatibility mode (in addition to 64-bit mode) when making a reference into a descriptor table.

A processor running in legacy mode can only load the low 32 bits of the base address, and the high 32 bits are ignored when references are made to the descriptor tables.

Code-Segment Descriptors. The AMD64 architecture defines a new code-segment descriptor attribute, L (long). In compatibility mode, the processor treats code-segment descriptors as it does in legacy mode, with the exception that the processor recognizes the L attribute. If a code descriptor with L=1 is loaded in compatibility mode, the processor leaves compatibility mode and enters 64-bit mode. In legacy mode, the L attribute is reserved.

The following differences exist for code-segment descriptors in 64-bit mode only:

- The CS base-address field is ignored by the processor.
- The CS limit field is ignored by the processor.
- Only the L (long), D (default size), and DPL (descriptor-privilege level) fields are used by the processor in 64-bit mode. All remaining attributes are ignored.

Data-Segment Descriptors. The following differences exist for data-segment descriptors in 64-bit mode only:

- The DS, ES, and SS descriptor base-address fields are ignored by the processor.

- The FS and GS descriptor base-address fields are expanded to 64 bits and used in effective-address calculations. The 64 bits of base address are mapped to model-specific registers (MSRs), and can only be loaded using the WRMSR instruction.
- The limit fields and attribute fields of all data-segment descriptors (DS, ES, FS, GS, and SS) are ignored by the processor.

In compatibility mode, the processor treats data-segment descriptors as it does in legacy mode. Compatibility mode ignores the high 32 bits of base address in the FS and GS segment descriptors when calculating an effective address.

System-Segment Descriptors. In 64-bit mode only, The LDT and TSS system-segment descriptor formats are expanded by 64 bits, allowing them to hold 64-bit base addresses. LLDT and LTR instructions can be used to load these descriptors into the LDTR and TR registers, respectively, from 64-bit mode.

In compatibility mode and legacy mode, the *formats* of the LDT and TSS system-segment descriptors are unchanged. Also, unlike code-segment and data-segment descriptors, system-segment descriptor limits *are checked* by the processor in long mode.

Some legacy mode LDT and TSS type-field encodings are illegal in long mode (both compatibility mode and 64-bit mode), and others are redefined to new types. See “System Descriptors” on page 99 for additional information.

Gate Descriptors. The following differences exist between gate descriptors in long mode (both compatibility mode and 64-bit mode) and in legacy mode:

- In long mode, all 32-bit gate descriptors are redefined as 64-bit gate descriptors, and are expanded to hold 64-bit offsets. The length of a gate descriptor in long mode is therefore 128 bits (16 bytes), versus the 64 bits (8 bytes) in legacy mode.
- Some type-field encodings are illegal in long mode, and others are redefined to new types. See “Gate Descriptors” on page 101 for additional information.
- The interrupt-gate and trap-gate descriptors define a new field, called the interrupt-stack table (IST) field.

2.3 Protection Checks

The AMD64 architecture makes the following changes to the protection mechanism in long mode:

- The page-protection-check mechanism is expanded in long mode to include the U/S and R/W protection bits stored in the PML4 entries and PDP entries.
- Several system-segment types and gate-descriptor types that are legal in legacy mode are illegal in long mode (compatibility mode and 64-bit mode) and fail type checks when used in long mode.
- Segment-limit checks are disabled in 64-bit mode for the CS, DS, ES, FS, GS, and SS segments. Segment-limit checks remain enabled for the LDT, GDT, IDT and TSS system segments.

  All segment-limit checks are performed in compatibility mode.

- Code and data segments used in 64-bit mode are treated as both readable and writable.

See “Page-Protection Checks” on page 158 and “Segment-Protection Overview” on page 104 for detailed information on the protection-check changes.

2.4 Registers

The AMD64 architecture adds additional registers to the architecture, and in many cases expands the size of existing registers to 64 bits. The 80-bit floating-point stack registers and their overlaid 64-bit MMX™ registers are not modified by the AMD64 architecture.

2.4.1 General-Purpose Registers

In 64-bit mode, the general-purpose registers (GPRs) are 64 bits wide, and eight additional GPRs are available. The GPRs are: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and the new R8–R15 registers. To access the full 64-bit operand size, or the new R8–R15 registers, an instruction must include a new REX instruction-prefix byte (see “REX Prefixes” on page 29 for a summary of this prefix).

In compatibility and legacy modes, the GPRs consist only of the eight legacy 32-bit registers. All legacy rules apply for determining operand size.

2.4.2 YMM/XMM Registers

In 64-bit mode, eight additional YMM/XMM registers are available, YMM/XMM8–15. A REX instruction prefix is used to access these registers. In compatibility and legacy modes, only registers YMM/XMM0–7 are accessible.

2.4.3 Flags Register

The flags register is expanded to 64 bits, and is called RFLAGS. All 64 bits can be accessed in 64-bit mode, but the upper 32 bits are reserved and always read back as zeros. Compatibility mode and legacy mode can read and write only the lower-32 bits of RFLAGS (the legacy EFLAGS).

2.4.4 Instruction Pointer

In long mode, the instruction pointer is extended to 64 bits, to support 64-bit code offsets. This 64-bit instruction pointer is called RIP.

2.4.5 Stack Pointer

In 64-bit mode, the size of the stack pointer, RSP, is always 64 bits. The stack size is not controlled by a bit in the SS descriptor, as it is in compatibility or legacy mode, nor can it be overridden by an instruction prefix. Address-size overrides are ignored for implicit stack references.

2.4.6 Control Registers

The AMD64 architecture defines several enhancements to the control registers (CR*n*). In long mode, all control registers are expanded to 64 bits, although the entire 64 bits can be read and written only from 64-bit mode. A new control register, the task-priority register (CR8 or TPR) is added, and can be read and written from 64-bit mode. Last, the function of the page-enable bit (CR0.PG) is expanded. When long mode is enabled, the PG bit is used to activate and deactivate long mode.

2.4.7 Debug Registers

In long mode, all debug registers are expanded to 64 bits, although the entire 64 bits can be read and written only from 64-bit mode. Expanded register encodings for the decode registers allow up to eight new registers to be defined (DR8–DR15), although presently those registers are not supported by the AMD64 architecture.

2.4.8 Extended Feature Register (EFER)

The EFER is expanded by the AMD64 architecture to include a long-mode-enable bit (LME), and a long-mode-active bit (LMA). These new bits can be accessed from legacy mode and long mode.

2.4.9 Memory Type Range Registers (MTRRs)

The legacy MTRRs are architecturally defined as 64 bits, and can accommodate the maximum 52-bit physical address allowed by the AMD64 architecture. From both long mode and legacy mode, implementations of the AMD64 architecture reference the entire 52-bit physical-address value stored in the MTRRs. Long mode and legacy mode system software can update all 64 bits of the MTRRs to manage the expanded physical-address space.

2.4.10 Other Model-Specific Registers (MSRs)

Several other MSRs have fields holding physical addresses. Examples include the APIC-base register and top-of-memory register. Generally, any model-specific register that contains a physical address is defined architecturally to be 64 bits wide, and can accommodate the maximum physical-address size defined by the AMD64 architecture. When physical addresses are read from MSRs by the processor, the entire value is read regardless of the operating mode. In legacy implementations, the high-order MSR bits are reserved, and software must write those values with zeros. In legacy mode on AMD64 architecture implementations, software can read and write all supported high-order MSR bits.

2.5 Instruction Set

2.5.1 REX Prefixes

REX prefixes are used in 64-bit mode to:

- Specify the new GPRs and YMM/XMM registers.
- Specify a 64-bit operand size.

- Specify additional control registers. One additional control register, CR8, is defined in 64-bit mode.
- Specify additional debug registers (although none are currently defined).

Not all instructions require a REX prefix. The prefix is necessary only if an instruction references one of the extended registers or uses a 64-bit operand. If a REX prefix is used when it has no meaning, it is ignored.

Default 64-Bit Operand Size. In 64-bit mode, two groups of instructions have a default operand size of 64 bits and thus do not need a REX prefix for this operand size:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP. See “Instructions that Reference RSP” on page 31 for additional information.

2.5.2 Segment-Override Prefixes in 64-Bit Mode

In 64-bit mode, the DS, ES, SS, and CS segment-override prefixes have no effect. These four prefixes are no longer treated as segment-override prefixes in the context of multiple-prefix rules. Instead, they are treated as null prefixes.

The FS and GS segment-override prefixes are treated as segment-override prefixes in 64-bit mode. Use of the FS and GS prefixes cause their respective segment bases to be added to the effective address calculation. See “FS and GS Registers in 64-Bit Mode” on page 80 for additional information on using these segment registers.

2.5.3 Operands and Results

The AMD64 architecture provides support for using 64-bit operands and generating 64-bit results when operating in 64-bit mode.

Operand-Size Overrides. In 64-bit mode, the default operand size is 32 bits. A REX prefix can be used to specify a 64-bit operand size. Software uses a legacy operand-size (66h) prefix to toggle to 16-bit operand size. The REX prefix takes precedence over the legacy operand-size prefix.

Zero Extension of Results. In 64-bit mode, when performing 32-bit operations with a GPR destination, the processor zero-extends the 32-bit result into the full 64-bit destination. Both 8-bit and 16-bit operations on GPRs preserve all unwritten upper bits of the destination GPR. This is consistent with legacy 16-bit and 32-bit semantics for partial-width results.

2.5.4 Address Calculations

The AMD64 architecture modifies aspects of effective-address calculation to support 64-bit mode. These changes are summarized in the following sections. See “Memory Addressing” in Volume 1 for details.

Address-Size Overrides. In 64-bit mode, the default-address size is 64 bits. The address size can be overridden to 32 bits by using the address-size prefix (67h). 16-bit addresses are not supported in 64-bit mode. In compatibility mode and legacy mode, address-size overrides function the same as in x86 legacy architecture.

Displacements and Immediates. Generally, displacement and immediate values in 64-bit mode are not extended to 64 bits. They are still limited to 32 bits and are sign extended during effective-address calculations. In 64-bit mode, however, support is provided for some 64-bit displacement and immediate forms of the MOV instruction.

Zero Extending 16-Bit and 32-Bit Addresses. All 16-bit and 32-bit address calculations are zero-extended in long mode to form 64-bit addresses. Address calculations are first truncated to the effective-address size of the current mode (64-bit mode or compatibility mode), as overridden by any address-size prefix. The result is then zero-extended to the full 64-bit address width.

RIP-Relative Addressing. A new addressing form, RIP-relative (instruction-pointer relative) addressing, is implemented in 64-bit mode. The effective address is formed by adding the displacement to the 64-bit RIP of the next instruction.

2.5.5 Instructions that Reference RSP

With the exception of far branches, all instructions that implicitly reference the 64-bit stack pointer, RSP, default to a 64-bit operand size in 64-bit mode (see Table 2-1 for a listing). Pushes and pops of 32-bit stack values are not possible in 64-bit mode with these instructions, but they can be overridden to 16 bits.

Table 2-1. Instructions That Reference RSP

Mnemonic	Opcode (hex)	Description
ENTER	C8	Create Procedure Stack Frame
LEAVE	C9	Delete Procedure Stack Frame
POP reg/mem	8F/0	Pop Stack (register or memory)
POP reg	58-5F	Pop Stack (register)
POP FS	0F A1	Pop Stack into FS Segment Register
POP GS	0F A9	Pop Stack into GS Segment Register
POPF, POPFD, POPFQ	9D	Pop to rFLAGS Word, Doubleword, or Quadword
PUSH imm32	68	Push onto Stack (sign-extended doubleword)
PUSH imm8	6A	Push onto Stack (sign-extended byte)
PUSH reg/mem	FF/6	Push onto Stack (register or memory)
PUSH reg	50-57	Push onto Stack (register)
PUSH FS	0F A0	Push FS Segment Register onto Stack
PUSH GS	0F A8	Push GS Segment Register onto Stack
PUSHF, PUSHFD, PUSHFQ	9C	Push rFLAGS Word, Doubleword, or Quadword onto Stack

2.5.6 Branches

The AMD64 architecture expands two branching mechanisms to accommodate branches in the full 64-bit virtual-address space:

- In 64-bit mode, near-branch semantics are redefined.
- In both 64-bit and compatibility modes, a 64-bit call-gate descriptor is defined for far calls.

In addition, enhancements are made to the legacy SYSCALL and SYSRET instructions.

Near Branches. In 64-bit mode, the operand size for all near branches defaults to 64 bits (see Table 2-2 for a listing). Therefore, these instructions update the full 64-bit RIP without the need for a REX operand-size prefix. The following aspects of near branches default to 64 bits:

- Truncation of the instruction pointer.
- Size of a stack pop or stack push, resulting from a CALL or RET.
- Size of a stack-pointer increment or decrement, resulting from a CALL or RET.
- Size of operand fetched by indirect-branch operand size.

The operand size for near branches can be overridden to 16 bits in 64-bit mode.

Table 2-2. 64-Bit Mode Near Branches, Default 64-Bit Operand Size

Mnemonic	Opcode (hex)	Description
CALL	E8, FF/2	Call Procedure Near
Jcc	many	Jump Conditional Near
JMP	E9, EB, FF/4	Jump Near
LOOP	E2	Loop
LOOPcc	E0, E1	Loop Conditional
RET	C3, C2	Return From Call (near)

The address size of near branches is not forced in 64-bit mode. Such addresses are 64 bits by default, but they can be overridden to 32 bits by a prefix.

The size of the displacement field for relative branches is still limited to 32 bits.

Far Branches Through Long-Mode Call Gates. Long mode redefines the 32-bit call-gate descriptor type as a 64-bit call-gate descriptor and expands the call-gate descriptor size to hold a 64-bit offset. The long-mode call-gate descriptor allows far branches to reference any location in the supported virtual-address space. In long mode, the call-gate mechanism is changed as follows:

- In long mode, CALL and JMP instructions that reference call-gates must reference 64-bit call gates.
- A 64-bit call-gate descriptor must reference a 64-bit code-segment.

- When a control transfer is made through a 64-bit call gate, the 64-bit target address is read from the 64-bit call-gate descriptor. The base address in the target code-segment descriptor is ignored.

Stack Switching. Automatic stack switching is also modified when a control transfer occurs through a call gate in long mode:

- The target-stack pointer read from the TSS is a 64-bit RSP value.
- The SS register is loaded with a null selector. Setting the new SS selector to null allows nested control transfers in 64-bit mode to be handled properly. The SS.RPL value is updated to remain consistent with the newly loaded CPL value.
- The size of pushes onto the new stack is modified to accommodate the 64-bit RIP and RSP values.
- Automatic parameter copying is not supported in long mode.

Far Returns. In long mode, far returns can load a null SS selector from the stack under the following conditions:

- The target operating mode is 64-bit mode.
- The target CPL<3.

Allowing RET to load SS with a null selector under these conditions makes it possible for the processor to unnest far CALLs (and interrupts) in long mode.

Task Gates. Control transfers through task gates are not supported in long mode.

Branches to 64-Bit Offsets. Because immediate values are generally limited to 32 bits, the only way a full 64-bit absolute RIP can be specified in 64-bit mode is with an indirect branch. For this reason, direct forms of far branches are eliminated from the instruction set in 64-bit mode.

SYSCALL and SYSRET Instructions. The AMD64 architecture expands the function of the legacy SYSCALL and SYSRET instructions in long mode. In addition, two new STAR registers, LSTAR and CSTAR, are provided to hold the 64-bit target RIP for the instructions when they are executed in long mode. The legacy STAR register is not expanded in long mode. See “SYSCALL and SYSRET” on page 171 for additional information.

SWAPGS Instruction. The AMD64 architecture provides the SWAPGS instruction as a fast method for system software to load a pointer to system data-structures. SWAPGS is valid only in 64-bit mode. An undefined-opcode exception (#UD) occurs if software attempts to execute SWAPGS in legacy mode or compatibility mode. See “SWAPGS Instruction” on page 173 for additional information.

SYSENTER and SYSEXIT Instructions. The SYSENTER and SYSEXIT instructions are invalid in long mode, and result in an invalid opcode exception (#UD) if software attempts to use them. Software should use the SYSCALL and SYSRET instructions when running in long mode. See “SYSENTER and SYSEXIT (Legacy Mode Only)” on page 173 for additional information.

2.5.7 NOP Instruction

The legacy x86 architecture commonly uses opcode 90h as a one-byte NOP. In 64-bit mode, the processor treats opcode 90h specially in order to preserve this NOP definition. This is necessary because opcode 90h is actually the XCHG EAX, EAX instruction in the legacy architecture. Without special handling in 64-bit mode, the instruction would not be a true no-operation. Therefore, in 64-bit mode the processor treats opcode 90h (the legacy XCHG EAX, EAX instruction) as a true NOP, regardless of a REX operand-size prefix.

This special handling does not apply to the two-byte ModRM form of the XCHG instruction. Unless a 64-bit operand size is specified using a REX prefix byte, using the two-byte form of XCHG to exchange a register with itself does not result in a no-operation, because the default operation size is 32 bits in 64-bit mode.

2.5.8 Single-Byte INC and DEC Instructions

In 64-bit mode, the legacy encodings for the 16 single-byte INC and DEC instructions (one for each of the eight GPRs) are used to encode the REX prefix values. The functionality of these INC and DEC instructions is still available, however, using the ModRM forms of those instructions (opcodes FF /0 and FF /1). See “Single-Byte INC and DEC Instructions in 64-Bit Mode” in Volume 3 for additional information.

2.5.9 MOVSXD Instruction

MOVSXD is a new instruction in 64-bit mode (the legacy ARPL instruction opcode, 63h, is reassigned as the MOVSXD opcode). It reads a fixed-size 32-bit source operand from a register or memory and (if a REX prefix is used with the instruction) sign-extends the value to 64 bits. MOVSXD is analogous to the MOVSX instruction, which sign-extends a byte to a word or a word to a doubleword, depending on the effective operand size. See the instruction reference page for the MOVSXD instruction in Volume 3 for additional information.

2.5.10 Invalid Instructions

Table 2-3 lists instructions that are illegal in 64-bit mode. Table 2-4 on page 35 lists instructions that are invalid in long mode (both compatibility mode and 64-bit mode). Attempted use of these instructions causes an invalid-opcode exception (#UD) to occur.

Table 2-3. Invalid Instructions in 64-Bit Mode

Mnemonic	Opcode (hex)	Description
AAA	37	ASCII Adjust After Addition
AAD	D5	ASCII Adjust Before Division
AAM	D4	ASCII Adjust After Multiply
AAS	3F	ASCII Adjust After Subtraction
BOUND	62	Check Array Bounds
CALL (far)	9A	Procedure Call Far (absolute)

Table 2-3. Invalid Instructions in 64-Bit Mode (continued)

Mnemonic	Opcode (hex)	Description
DAA	27	Decimal Adjust after Addition
DAS	2F	Decimal Adjust after Subtraction
INTO	CE	Interrupt to Overflow Vector
JMP (far)	EA	Jump Far (absolute)
LDS	C5	Load DS Segment Register
LES	C4	Load ES Segment Register
POP DS	1F	Pop Stack into DS Segment
POP ES	07	Pop Stack into ES Segment
POP SS	17	Pop Stack into SS Segment
POPA, POPAD	61	Pop All to GPR Words or Doublewords
PUSH CS	0E	Push CS Segment Selector onto Stack
PUSH DS	1E	Push DS Segment Selector onto Stack
PUSH ES	06	Push ES Segment Selector onto Stack
PUSH SS	16	Push SS Segment Selector onto Stack
PUSHA, PUSHAD	60	Push All GPR Words or Doublewords onto Stack
Redundant Grp1 (undocumented)	82	Redundant encoding of group1 Eb,Ib opcodes
SALC (undocumented)	D6	Set AL According to CF

Table 2-4. Invalid Instructions in Long Mode

Mnemonic	Opcode (hex)	Description
SYSENTER	0F 34	System Call
SYSEXIT	0F 35	System Return

2.5.11 Reassigned Opcodes

Table 2-5 below lists opcodes that are assigned functions in 64-bit mode that differ from their legacy functions.

Table 2-5. Opcodes Reassigned in 64-Bit Mode

Opcode (hex)	Compatibility and Legacy Modes	64-Bit Mode
63	ARPL—Adjust Requestor Privilege Level	MOVSXD—Move Doubleword with Sign Extension
40–4F	DEC—Decrement by 1 INC—Increment by 1	REX Prefix
***Note**: Two-byte versions of DEC and INC are still available in 64-bit mode.*		

2.5.12 FXSAVE and FXRSTOR Instructions

The FXSAVE and FXRSTOR instructions are used to save and restore the entire 128-bit media (XMM), 64-bit media, and x87 instruction-set environment during a context switch. The AMD64 architecture modifies the memory format used by these instructions in order to save and restore the full 64-bit instruction and data pointers, as well as the XMM8–15 registers. Selection of the 32-bit legacy format or the expanded 64-bit format is accomplished by using the corresponding operand size with the FXSAVE and FXRSTOR instructions. When 64-bit software executes an FXSAVE and FXRSTOR with a 32-bit operand size (no operand-size override) the 32-bit legacy format is used. When 64-bit software executes an FXSAVE and FXRSTOR with a 64-bit operand size, the 64-bit format is used.

For more information on the save area formats, see Section 11.4.4, “Saving Media and x87 Execution Unit State,” on page 342

If the fast-FXSAVE/FXRSTOR (FFXSR) feature is enabled in EFER, FXSAVE and FXRSTOR do not save or restore the XMM0–15 registers when executed in 64-bit mode at CPL 0. The x87 environment and MXCSR are saved whether fast-FXSAVE/FXRSTOR is enabled or not. The fast-FXSAVE/FXRSTOR feature has no effect on FXSAVE/FXRSTOR in non 64-bit mode or when CPL > 0.

Software can use the CPUID instruction to determine whether the fast-FXSAVE/FXRSTOR feature is available (CPUID Fn8000_0001h_EDX[FFXSR]). For information on using the CPUID instruction to obtain processor feature information, see Section 3.3, “Processor Feature Identification,” on page 71.

2.6 Interrupts and Exceptions

When a processor is running in long mode, an interrupt or exception causes the processor to enter 64-bit mode. All long-mode interrupt handlers must be implemented as 64-bit code. The AMD64 architecture expands the legacy interrupt-processing and exception-processing mechanism to support

handling of interrupts by 64-bit operating systems and applications. The changes are summarized in the following sections. See “Long-Mode Interrupt Control Transfers” on page 272 for detailed information on these changes.

2.6.1 Interrupt Descriptor Table

The long-mode interrupt-descriptor table (IDT) must contain 64-bit mode interrupt-gate or trap-gate descriptors for all interrupts or exceptions that can occur while the processor is running in long mode. Task gates cannot be used in the long-mode IDT, because control transfers through task gates are not supported in long mode. In long mode, the IDT index is formed by scaling the interrupt vector by 16. In legacy protected mode, the IDT is indexed by scaling the interrupt vector by eight.

2.6.2 Stack Frame Pushes

In legacy mode, the size of an IDT entry (16 bits or 32 bits) determines the size of interrupt-stack-frame pushes, and SS:eSP is pushed only on a CPL change. In long mode, the size of interrupt stack-frame pushes is fixed at eight bytes, because interrupts are handled in 64-bit mode. Long mode interrupts also cause SS:RSP to be pushed unconditionally, rather than pushing only on a CPL change.

2.6.3 Stack Switching

Legacy mode provides a mechanism to automatically switch stack frames in response to an interrupt. In long mode, a slightly modified version of the legacy stack-switching mechanism is implemented, and an alternative stack-switching mechanism—called the interrupt stack table (IST)—is supported.

Long-Mode Stack Switches. When stacks are switched as part of a long-mode privilege-level change resulting from an interrupt, the following occurs:

- The target-stack pointer read from the TSS is a 64-bit RSP value.
- The SS register is loaded with a null selector. Setting the new SS selector to null allows nested control transfers in 64-bit mode to be handled properly. The SS.RPL value is cleared to 0.
- The old SS and RSP are saved on the new stack.

Interrupt Stack Table. In long mode, a new interrupt stack table (IST) mechanism is available as an alternative to the modified legacy stack-switching mechanism. The IST mechanism unconditionally switches stacks when it is enabled. It can be enabled for individual interrupt vectors using a field in the IDT entry. This allows mixing interrupt vectors that use the modified legacy mechanism with vectors that use the IST mechanism. The IST pointers are stored in the long-mode TSS. The IST mechanism is only available when long mode is enabled.

2.6.4 IRET Instruction

In compatibility mode, IRET pops SS:eSP off the stack only if there is a CPL change. This allows legacy applications to run properly in compatibility mode when using the IRET instruction.

In 64-bit mode, IRET unconditionally pops SS:eSP off of the interrupt stack frame, even if the CPL does not change. This is done because the original interrupt always pushes SS:RSP. Because interrupt

stack-frame pushes are always eight bytes in long mode, an IRET from a long-mode interrupt handler (64-bit code) must pop eight-byte items off the stack. This is accomplished by preceding the IRET with a 64-bit REX operand-size prefix.

In long mode, an IRET can load a null SS selector from the stack under the following conditions:

- The target operating mode is 64-bit mode.
- The target CPL<3.

Allowing IRET to load SS with a null selector under these conditions makes it possible for the processor to unnest interrupts (and far CALLs) in long mode.

2.6.5 Task-Priority Register (CR8)

The AMD64 architecture allows software to define up to 15 external interrupt-priority classes. Priority classes are numbered from 1 to 15, with priority-class 1 being the lowest and priority-class 15 the highest.

A new control register (CR8) is introduced by the AMD64 architecture for managing priority classes. This register, also called the *task-priority register* (TPR), uses the four low-order bits for specifying a task priority. How external interrupts are organized into these priority classes is implementation dependent. See “External Interrupt Priorities” on page 258 for information on this feature.

2.6.6 New Exception Conditions

The AMD64 architecture defines a number of new conditions that can cause an exception to occur when the processor is running in long mode. Many of the conditions occur when software attempts to use an address that is not in canonical form. See “Vectors” on page 236 for information on the new exception conditions that can occur in long mode.

2.7 Hardware Task Switching

The legacy hardware task-switch mechanism is disabled when the processor is running in long mode. However, long mode requires system software to create data structures for a single task—the long-mode task.

- *TSS Descriptors*—A new TSS-descriptor type, the 64-bit TSS type, is defined for use in long mode. It is the only valid TSS type that can be used in long mode, and it must be loaded into the TR by executing the LTR instruction in *64-bit mode*. See “TSS Descriptor” on page 364 for additional information.
- *Task Gates*—Because the legacy task-switch mechanism is not supported in long mode, *software cannot use task gates in long mode*. Any attempt to transfer control to another task through a task gate causes a general-protection exception (#GP) to occur.
- *Task-State Segment*—A 64-bit task state segment (TSS) is defined for use in long mode. This new TSS format contains 64-bit stack pointers (RSP) for privilege levels 0–2, interrupt-stack-table

(IST) pointers, and the I/O-map base address. See “64-Bit Task State Segment” on page 371 for additional information.

2.8 Long-Mode vs. Legacy-Mode Differences

Table 2-6 on page 39 summarizes several major system-programming differences between 64-bit mode and legacy protected mode. The third column indicates whether the difference also applies to compatibility mode. “Differences Between Long Mode and Legacy Mode” in Volume 3 summarizes the application-programming model differences.

Table 2-6. Differences Between Long Mode and Legacy Mode

Subject	64-Bit Mode Difference	Applies To Compatibility Mode?
x86 Modes	Real and virtual-8086 modes not supported	Yes
Task Switching	Task switching not supported	Yes
Addressing	64-bit virtual addresses	No
	4-level paging structures	Yes
	PAE must always be enabled	
Loaded Segment (Usage during memory reference)	CS, DS, ES, SS segment bases are ignored	No
	CS, DS, ES, FS, GS, SS segment limits are ignored	
	DS, ES, FS, GS attribute are ignored	
	CS, DS, ES, SS Segment prefixes are ignored	
Exception and Interrupt Handling	All pushes are 8 bytes	Yes
	IDT entries are expanded to 16 bytes	
	SS is not changed for stack switch	
	SS:RSP is pushed unconditionally	
Call Gates	All pushes are 8 bytes	Yes
	16-bit call gates are illegal	
	32-bit call gate type is redefined as 64-bit call gate and is expanded to 16 bytes	
	SS is not changed for stack switch	
System-Descriptor Registers	GDT, IDT, LDT, TR base registers expanded to 64 bits	Yes
System-Descriptor Table Entries and Pseudo-Descriptors	LGDT and LIDT use expanded 10-byte pseudo-descriptors	No
	LLDT and LTR use expanded 16-byte table entries	

3 System Resources

The operating system manages the software-execution environment and general system operation through the use of system resources. These resources consist of system registers (control registers and model-specific registers) and system-data structures (memory-management and protection tables). The system-control registers are described in detail in this chapter; many of the features they control are described elsewhere in this volume. The model-specific registers supported by the AMD64 architecture are introduced in this chapter.

Because of their complexity, system-data structures are described in separate chapters. Refer to the following chapters for detailed information on these data structures:

- Descriptors and descriptor tables are described in Section 4.4 “Segmentation Data Structures and Registers,” on page 75.
- Page-translation tables are described in Section 5.2 “Legacy-Mode Page Translation,” on page 133 and Section 5.3 “Long-Mode Page Translation,” on page 141.
- The task-state segment is described in Section 12.2.4 “Legacy Task-State Segment,” on page 367 and Section 12.2.5 “64-Bit Task State Segment,” on page 371.

Not all processor implementations are required to support all possible features. The last section in this chapter addresses processor-feature identification. System software uses the capabilities described in that section to determine which features are supported so that the appropriate service routines are loaded.

3.1 System-Control Registers

The registers that control the AMD64 architecture operating environment include:

- *CR0*—Provides operating-mode controls and some processor-feature controls.
- *CR2*—This register is used by the page-translation mechanism. It is loaded by the processor with the page-fault virtual address when a page-fault exception occurs.
- *CR3*—This register is also used by the page-translation mechanism. It contains the base address of the highest-level page-translation table, and also contains cache controls for the specified table.
- *CR4*—This register contains additional controls for various operating-mode features.
- *CR8*—This new register, accessible in 64-bit mode using the REX prefix, is introduced by the AMD64 architecture. CR8 is used to prioritize external interrupts and is referred to as the *task-priority register* (TPR).
- *RFLAGS*—This register contains processor-status and processor-control fields. The status and control fields are used primarily in the management of virtual-8086 mode, hardware multitasking, and interrupts.

- *EFER*—This model-specific register contains status and controls for additional features not managed by the CR0 and CR4 registers. Included in this register are the long-mode enable and activation controls introduced by the AMD64 architecture.

Control registers CR1, CR5–CR7, and CR9–CR15 are reserved.

In legacy mode, all control registers and RFLAGS are 32 bits. The EFER register is 64 bits in all modes. The AMD64 architecture expands all 32-bit system-control registers to 64 bits. In 64-bit mode, the MOV CR*n* instructions read or write all 64 bits of these registers (operand-size prefixes are ignored). In compatibility and legacy modes, control-register writes fill the low 32 bits with data and the high 32 bits with zeros, and control-register reads return only the low 32 bits.

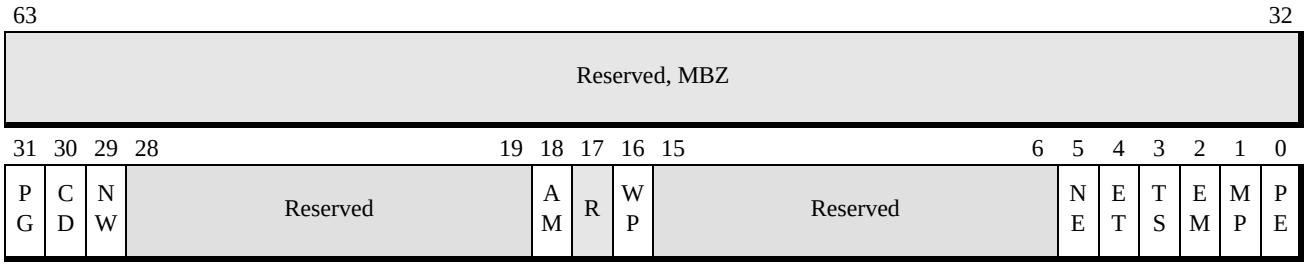
In 64-bit mode, the high 32 bits of CR0 and CR4 are reserved and must be written with zeros. Writing a 1 to any of the high 32 bits results in a general-protection exception, #GP(0). All 64 bits of CR2 are writable. However, the MOV CR*n* instructions *do not* check that addresses written to CR2 are within the virtual-address limitations of the processor implementation.

All CR3 bits are writable, except for unimplemented physical address bits, which must be cleared to 0.

The upper 32 bits of RFLAGS are always read as zero by the processor. Attempts to load the upper 32 bits of RFLAGS with anything other than zero are ignored by the processor.

3.1.1 CR0 Register

The CR0 register is shown in Figure 3-1 on page 43. The legacy CR0 register is identical to the low 32 bits of this register (CR0 bits 31:0).



Bits	Mnemonic	Description	Access type
63:32	—	Reserved	MBZ
31	PG	Paging	R/W
30	CD	Cache Disable	R/W
29	NW	Not Writethrough	R/W
28:19	—	Reserved - do not change	
18	AM	Alignment Mask	R/W
17	—	Reserved - do not change	
16	WP	Write Protect	R/W[1]
15:6	—	Reserved - do not change	
5	NE	Numeric Error	R/W
4	ET	Extension Type	R
3	TS	Task Switched	R/W
2	EM	Emulation	R/W
1	MP	Monitor Coprocessor	R/W
0	PE	Protection Enabled	R/W

1. Conditionally -- see description below.

Figure 3-1. Control Register 0 (CR0)

The functions of the CR0 control bits are (unless otherwise noted, all bits are read/write):

Protected-Mode Enable (PE) Bit. Bit 0. Software enables protected mode by setting PE to 1, and disables protected mode by clearing PE to 0. When the processor is running in protected mode, segment-protection mechanisms are enabled.

See Section 4.9 “Segment-Protection Overview,” on page 104 for information on the segment-protection mechanisms.

Monitor Coprocessor (MP) Bit. Bit 1. Software uses the MP bit with the task-switched control bit (CR0.TS) to control whether execution of the WAIT/FWAIT instruction causes a device-not-available exception (#NM) to occur, as follows:

- If both the monitor-coprocessor and task-switched bits are set (CR0.MP=1 *and* CR0.TS=1), then executing the WAIT/FWAIT instruction causes a device-not-available exception (#NM).
- If either the monitor-coprocessor or task-switched bits are clear (CR0.MP=0 *or* CR0.TS=0), then executing the WAIT/FWAIT instruction proceeds normally.

Software typically should set MP to 1 if the processor implementation supports x87 instructions. This allows the CR0.TS bit to completely control when the x87-instruction context is saved as a result of a task switch.

Emulate Coprocessor (EM) Bit. Bit 2. Software forces all x87 instructions to cause a device-not-available exception (#NM) by setting EM to 1. Likewise, setting EM to 1 forces an invalid-opcode exception (#UD) when an attempt is made to execute any of the 64-bit or 128-bit media instructions except the FXSAVE and FXRSTOR instructions. Attempting to execute these instructions when EM is set results in an #NM exception instead.The exception handlers can emulate these instruction types if desired. Setting the EM bit to 1 does not cause an #NM exception when the WAIT/FWAIT instruction is executed.

Task Switched (TS) Bit. Bit 3. When an attempt is made to execute an x87 or media instruction while TS=1, a device-not-available exception (#NM) occurs. Software can use this mechanism—sometimes referred to as “lazy context-switching”—to save the unit contexts before executing the next instruction of those types. As a result, the x87 and media instruction-unit contexts are saved only when necessary as a result of a task switch.

When a hardware task switch occurs, TS is automatically set to 1. System software that implements software task-switching rather than using the hardware task-switch mechanism can still use the TS bit to control x87 and media instruction-unit context saves. In this case, the task-management software uses a MOV CR0 instruction to explicitly set the TS bit to 1 during a task switch. Software can clear the TS bit by either executing the CLTS instruction or by writing to the CR0 register directly. Long-mode system software can use this approach even though the hardware task-switch mechanism is not supported in long mode.

The CR0.MP bit controls whether the WAIT/FWAIT instruction causes an #NM exception when TS=1.

Extension Type (ET) Bit. Bit 4, read-only. In some early x86 processors, software set ET to 1 to indicate support of the 387DX math-coprocessor instruction set. This bit is now reserved and forced to 1 by the processor. Software cannot clear this bit to 0.

Numeric Error (NE) Bit. Bit 5. Clearing the NE bit to 0 disables internal control of x87 floating-point exceptions and enables external control. When NE is cleared to 0, the IGNNE# input signal controls whether x87 floating-point exceptions are ignored:

- When IGNNE# is 1, x87 floating-point exceptions are ignored.
- When IGNNE# is 0, x87 floating-point exceptions are reported by setting the FERR# input signal to 1. External logic can use the FERR# signal as an external interrupt.

When NE is set to 1, internal control over x87 floating-point exception reporting is enabled and the external reporting mechanism is disabled. It is recommended that software set NE to 1. This enables optimal performance in handling x87 floating-point exceptions.

Write Protect (WP) Bit. Bit 16. Read-only pages are protected from supervisor-level writes when the WP bit is set to 1. When WP is cleared to 0, supervisor software can write into read-only pages.

See Section 5.6 “Page-Protection Checks,” on page 158 for information on the page-protection mechanism. If the shadow stack feature has been enabled (CR4.CET=1), attempting to clear WP to 0 causes a general-protection exception (#GP).

Alignment Mask (AM) Bit. Bit 18. Software enables automatic alignment checking by setting the AM bit to 1 when RFLAGS.AC=1. Alignment checking can be disabled by clearing either AM or RFLAGS.AC to 0. When automatic alignment checking is enabled and CPL=3, a memory reference to an unaligned operand causes an alignment-check exception (#AC).

Not Writethrough (NW) Bit. Bit 29. Ignored. This bit can be set to 1 or cleared to 0, but its value is ignored. The NW bit exists only for legacy purposes.

Cache Disable (CD) Bit. Bit 30. When CD is cleared to 0, the internal caches are enabled. When CD is set to 1, no new data or instructions are brought into the internal caches. However, the processor still accesses the internal caches when CD = 1 under the following situations:

- Reads that hit in an internal cache cause the data to be read from the internal cache that reported the hit.
- Writes that hit in an internal cache cause the cache line that reported the hit to be written back to memory and invalidated in the cache.

Cache misses do not affect the internal caches when CD = 1. Software can prevent cache access by setting CD to 1 and invalidating the caches.

Setting CD to 1 also causes the processor to ignore the page-level cache-control bits (PWT and PCD) when paging is enabled. These bits are located in the page-translation tables and CR3 register. See Section “Page-Level Writethrough (PWT) Bit,” on page 151 and Section “Page-Level Cache Disable (PCD) Bit,” on page 151 for information on page-level cache control.

See Section 7.6 “Memory Caches,” on page 199 for information on the internal caches.

Paging Enable (PG) Bit. Bit 31. Software enables page translation by setting PG to 1, and disables page translation by clearing PG to 0. Page translation cannot be enabled unless the processor is in protected mode (CR0.PE=1). If software attempts to set PG to 1 when PE is cleared to 0, the processor causes a general-protection exception (#GP).

See Section 5.1 “Page Translation Overview,” on page 129 for information on the page-translation mechanism.

Reserved Bits. Bits 28:19, 17, 15:6, and 63:32. When writing the CR0 register, software should set the values of reserved bits to the values found during the previous CR0 read. No attempt should be made to change reserved bits, and software should never rely on the values of reserved bits. In long mode, bits 63:32 are reserved and must be written with zero, otherwise a #GP occurs.

3.1.2 CR2 and CR3 Registers

The CR2 (page-fault linear address) register, shown in Figure 3-2 on page 46 and Figure 3-3 on page 46, and the CR3 (page-translation-table base address) register, shown in Figure 3-4 and Figure 3-5 on page 46, and Figure 3-6 on page 47, are used only by the page-translation mechanism.

Figure 3-2. Control Register 2 (CR2)—Legacy-Mode

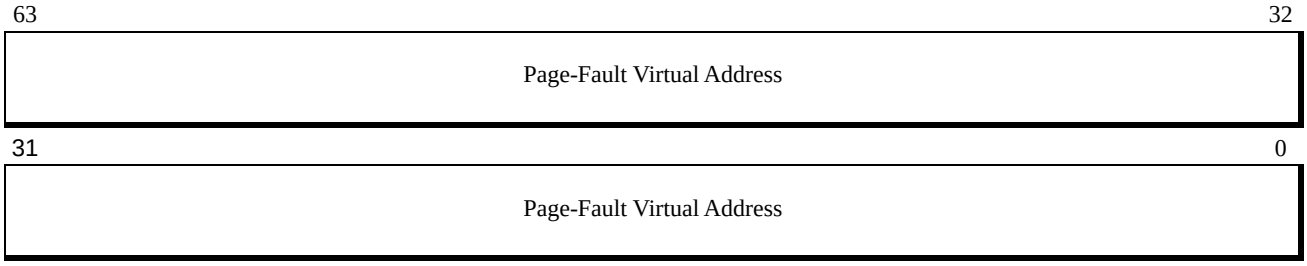


Figure 3-3. Control Register 2 (CR2)—Long Mode

See Section “CR2 Register,” on page 247 for a description of the CR2 register.

The CR3 register is used to point to the base address of the highest-level page-translation table.

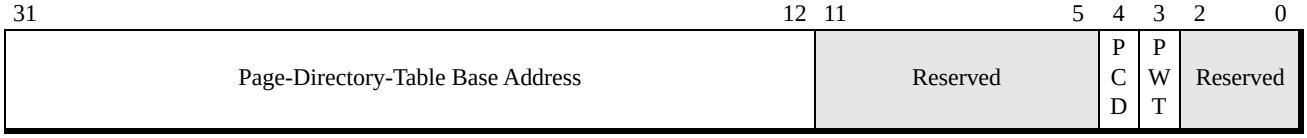


Figure 3-4. Control Register 3 (CR3)—Legacy-Mode Non-PAE Paging

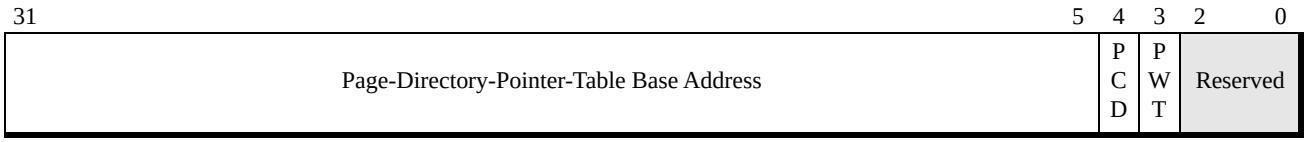


Figure 3-5. Control Register 3 (CR3)—Legacy-Mode PAE Paging

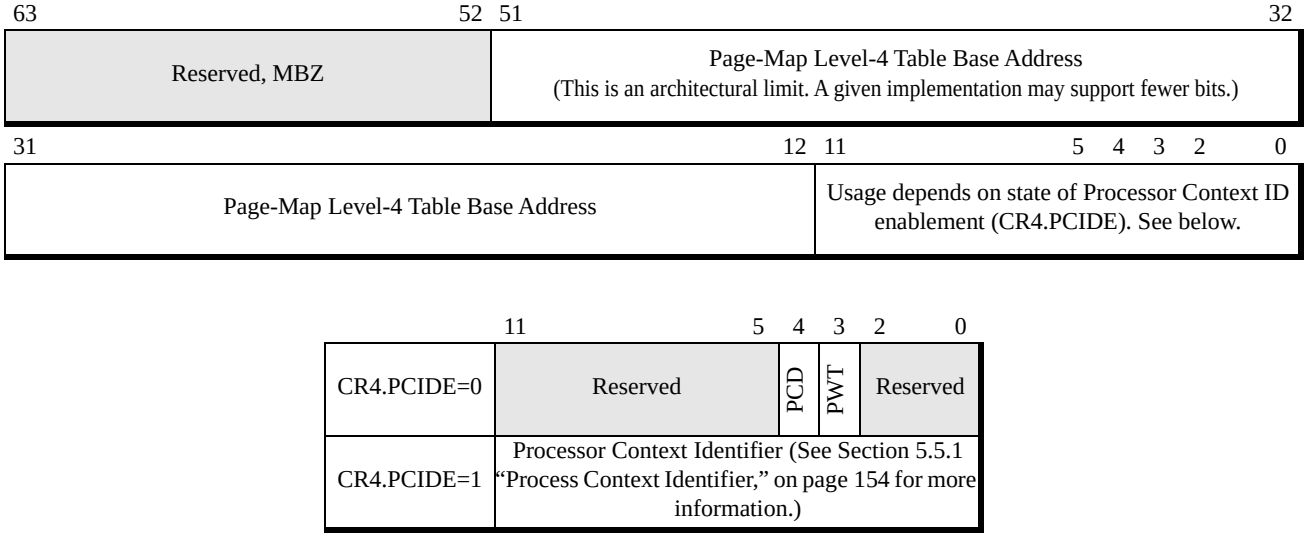
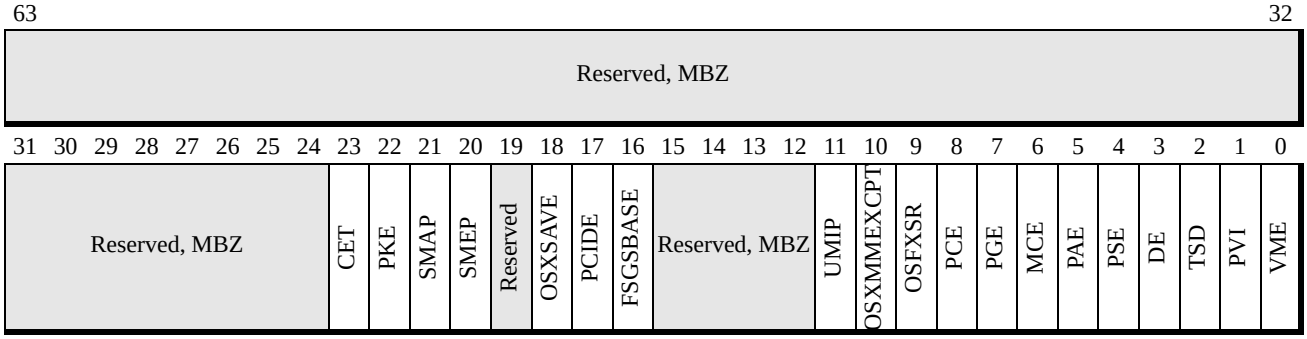


Figure 3-6. Control Register 3 (CR3)—Long Mode

The legacy CR3 register is described in Section 5.2.1 “CR3 Register,” on page 134, and the long-mode CR3 register is described in Section 5.3.2 “CR3,” on page 141.

3.1.3 CR4 Register

The CR4 register is shown in Figure 3-7. In legacy mode, the CR4 register is identical to the low 32 bits of the register (CR4 bits 31:0). The features controlled by the bits in the CR4 register are model-specific extensions. Except for the performance-counter extensions (PCE) feature, software can use the CPUID instruction to verify that each feature is supported before using that feature. See Section 3.3 “Processor Feature Identification,” on page 71 for information on using the CPUID instruction.



Bits	Mnemonic	Description	Access Type
63:24	—	Reserved	MBZ
23	CET	Control-flow Enforcement Technology	R/W
22	PKE	Protection Key Enable	R/W
21	SMAP	Supervisor Mode Access Protection	R/W
20	SMEP	Supervisor Mode Execution Prevention	R/W
19	—	Reserved	MBZ
18	OSXSAVE	XSAVE and Processor Extended States Enable Bit	R/W
17	PCIDE	Process Context Identifier Enable	R/W
16	FSGSBASE	Enable RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE instructions	R/W
15:12	—	Reserved	MBZ
11	UMIP	User Mode Instruction Prevention	R/W
10	OSXMMEXCPT	Operating System Unmasked Exception Support	R/W
9	OSFXSR	Operating System FXSAVE/FXRSTOR Support	R/W
8	PCE	Performance-Monitoring Counter Enable	R/W
7	PGE	Page-Global Enable	R/W
6	MCE	Machine Check Enable	R/W
5	PAE	Physical-Address Extension	R/W
4	PSE	Page Size Extensions	R/W
3	DE	Debugging Extensions	R/W
2	TSD	Time Stamp Disable	R/W
1	PVI	Protected-Mode Virtual Interrupts	R/W
0	VME	Virtual-8086 Mode Extensions	R/W

The function of the CR4 control bits are (all bits are read/write):

Virtual-8086 Mode Extensions (VME). Bit 0. Setting VME to 1 enables hardware-supported performance enhancements for software running in virtual-8086 mode. Clearing VME to 0 disables this support. The enhancements enabled when VME=1 include:

- Virtualized, maskable, external-interrupt control and notification using the VIF and VIP bits in the RFLAGS register. Virtualizing affects the operation of several instructions that manipulate the RFLAGS.IF bit.
- Selective intercept of software interrupts (INT*n* instructions) using the interrupt-redirection bitmap in the TSS.

Protected-Mode Virtual Interrupts (PVI). Bit 1. Setting PVI to 1 enables support for protected-mode virtual interrupts. Clearing PVI to 0 disables this support. When PVI=1, hardware support of two bits in the RFLAGS register, VIF and VIP, is enabled.

Only the STI and CLI instructions are affected by enabling PVI. Unlike the case when CR0.VME=1, the interrupt-redirection bitmap in the TSS cannot be used for selective INT*n* interception.

PVI enhancements are also supported in long mode. See Section 8.10 “Virtual Interrupts,” on page 279 for more information on using PVI.

Time-Stamp Disable (TSD). Bit 2. The TSD bit allows software to control the privilege level at which the time-stamp counter can be read. When TSD is cleared to 0, software running at any privilege level can read the time-stamp counter using the RDTSC or RDTSCP instructions. When TSD is set to 1, only software running at privilege-level 0 can execute the RDTSC or RDTSCP instructions.

Debugging Extensions (DE). Bit 3. Setting the DE bit to 1 enables the I/O breakpoint capability and enforces treatment of the DR4 and DR5 registers as reserved. Software that accesses DR4 or DR5 when DE=1 causes a invalid opcode exception (#UD).

When the DE bit is cleared to 0, I/O breakpoint capabilities are disabled. Software references to the DR4 and DR5 registers are aliased to the DR6 and DR7 registers, respectively.

Page-Size Extensions (PSE). Bit 4. Setting PSE to 1 enables the use of 4-Mbyte physical pages. With PSE=1, the physical-page size is selected between 4 Kbytes and 4 Mbytes using the page-directory entry page-size field (PS). Clearing PSE to 0 disables the use of 4-Mbyte physical pages and restricts all physical pages to 4 Kbytes.

The PSE bit has no effect when physical-address extensions are enabled (CR4.PAE=1). Because long mode requires CR4.PAE=1, the PSE bit is ignored when the processor is running in long mode.

See Section “4-Mbyte Page Translation,” on page 136 for more information on 4-Mbyte page translation.

Physical-Address Extension (PAE). Bit 5. Setting PAE to 1 enables the use of physical-address extensions and 2-Mbyte physical pages. Clearing PAE to 0 disables these features.

With PAE=1, the page-translation data structures are expanded from 32 bits to 64 bits, allowing the translation of up to 52-bit physical addresses. Also, the physical-page size is selectable between 4 Kbytes and 2 Mbytes using the page-directory-entry page-size field (PS). Long mode requires PAE to be enabled in order to use the 64-bit page-translation data structures to translate 64-bit virtual addresses to 52-bit physical addresses.

See Section 5.2.3 “PAE Paging,” on page 137 for more information on physical-address extensions.

Machine-Check Enable (MCE). Bit 6. Setting MCE to 1 enables the machine-check exception mechanism. Clearing this bit to 0 disables the mechanism. When enabled, a machine-check exception (#MC) occurs when an uncorrectable machine-check error is encountered.

Regardless of whether machine-check exceptions are enabled, the processor records enabled-errors when they occur. Error-reporting is performed by the machine-check error-reporting register banks. Each bank includes a control register for enabling error reporting and a status register for capturing errors. Correctable machine-check errors are also reported, but they do not cause a machine-check exception.

See Chapter 9, “Machine Check Architecture,” for a description of the machine-check mechanism, the registers used, and the types of errors captured by the mechanism.

Page-Global Enable (PGE). Bit 7. When page translation is enabled, system-software performance can often be improved by making some page translations *global* to all tasks and procedures. Setting PGE to 1 enables the global-page mechanism. Clearing this bit to 0 disables the mechanism.

When PGE is enabled, system software can set the global-page (G) bit in the lowest level of the page-translation hierarchy to 1, indicating that the page translation is global. Page translations marked as global are not invalidated in the TLB when the page-translation-table base address (CR3) is updated. When the G bit is cleared, the page translation is not global. All supported physical-page sizes also support the global-page mechanism. See Section 5.5.2 “Global Pages,” on page 155 for information on using the global-page mechanism.

Performance-Monitoring Counter Enable (PCE). Bit 8. Setting PCE to 1 allows software running at any privilege level to use the RDPMC instruction. Software uses the RDPMC instruction to read the performance-monitoring counter MSRs, *PerfCtr*n*. Clearing PCE to 0 allows only the most-privileged software (CPL=0) to use the RDPMC instruction.

FXSAVE/FXRSTOR Support (OSFXSR). Bit 9. System software must set the OSFXSR bit to 1 to enable use of the legacy SSE instructions. When this bit is set to 1, it also indicates that system software uses the FXSAVE and FXRSTOR instructions to save and restore the processor state for the x87, 64-bit media, and 128-bit media instructions.

Clearing the OSFXSR bit to 0 indicates that legacy SSE instructions cannot be used. Attempts to use those instructions while this bit is clear result in an invalid-opcode exception (#UD). Software can continue to use the FXSAVE/FXRSTOR instructions for saving and restoring the processor state for the x87 and 64-bit media instructions.

Unmasked Exception Support (OSXMMEXCPT). Bit 10. System software must set the OSXMMEXCPT bit to 1 when it supports the SIMD floating-point exception (#XF) for handling of unmasked 256-bit and 128-bit media floating-point errors. Clearing the OSXMMEXCPT bit to 0 indicates the #XF handler is not supported. When OSXMMEXCPT=0, unmasked 128-bit media floating-point exceptions cause an invalid-opcode exception (#UD). See “SIMD Floating-Point Exception Causes” in Volume 1 for more information on unmasked SSE floating-point exceptions.

User Mode Instruction Prevention (UMIP). Bit 11. Setting UMIP to 1 enables a security mode to restrict certain instructions executing at CPL>0 so that they do not reveal information about structures that are controlled by the processor when it is at CPL=0. When UMIP is enabled, execution of SGDT, SIDT, SLDT, SMSW and STR instructions become available only at CPL=0 and any attempt to execute them with CPL>0 results in a #GP fault with error code 0. See Section 6 “System Instructions,” on page 167 for more information about UMIP.

FSGSBASE. Bit 16. System software must set this bit to 1 to enable the execution of the RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE instructions when supported. When enabled, these instructions allow software running in 64-bit mode at any privilege level to read and write the FS.base and GS.base hidden segment register state. See the discussion of segment registers in 64-bit mode in Section 4.5.3 “Segment Registers in 64-Bit Mode,” on page 80. Also see descriptions of the RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE instructions in Volume 3.

Processor Context Identifier Enable (PCIDE). Bit 17. Enable support for Process Context Identifiers (PCIDs). System software must set this bit to 1 to enable execution of the INVPCID instruction when supported. Can only be set in long mode (EFER.LMA = 1). See Chapter 5.5.1, “Process Context Identifier,” for more information on Process Context Identifiers.

XSAVE and Extended States (OSXSAVE). Bit 18. After verifying hardware support for the extended processor state management instructions, operating system software sets this bit to indicate support for the XGETBV, XSAVE and XRSTOR instructions.

Setting this bit also:

- allows the execution of the XGETBV and XSETBV instructions, and
- enables the XSAVE and XRSTOR instructions to save and restore the x87 FPU state (including MMX registers), along with other processor extended states enabled in XCR0.

After initializing the XSAVE/XRSTOR save area, XSAVEOPT (if supported) may be used to save x87 FPU and other enabled extended processor state. For more information on XSAVEOPT, see individual instruction listing in Chapter 2 of Volume 4.

Note that legacy SSE instruction execution must be enabled prior to enabling extended processor state management.

Supervisor Mode Execution Prevention (SMEP). Bit 20. Setting this bit enables the supervisor mode execution prevention feature, if supported. This feature prevents the execution of instructions that reside in pages accessible by user-mode software when the processor is in supervisor-mode. See Section 5.6 “Page-Protection Checks,” on page 158 for more information.

Supervisor Mode Access Prevention (SMAP). Bit 21. Setting this bit enables the supervisor mode access prevention feature, if supported. This feature prevents certain data accesses to pages accessible by user-mode software when the processor is in supervisor mode. See Section 5.6.6 “Supervisor-Mode Access Prevention(CR4.SMAP) Bit,” on page 160 for more information.

Protection Key Enable (PKE). Bit 22. Enable support for memory Protection Keys. Also enables support for the RDPKRU and WRPKRU instructions. A MOV to CR4 that changes CR4.PKE from 0 to 1 causes all cached entries in the TLB for the logical processor to be invalidated. (See Section 5.6.7 “Memory Protection Keys (MPK) Bit,” on page 161 for more information on memory protection keys.)

Control-flow Enforcement Technology (CET). Bit 23. Setting this bit enables the shadow stack feature. This feature ensures that return addresses read from the stack by RET and IRET instructions originated from a CALL instruction or similar control transfer.

See Section 18 “Shadow Stacks,” on page 647 for more information. Before setting this bit, CR0.WP must be set to 1, otherwise a #GP fault is generated.

CR1 and CR5–CR7 Registers. Control registers CR1, CR5–CR7, and CR9–CR15 are reserved. Attempts by software to use these registers result in an undefined-opcode exception (#UD).

3.1.4 Additional Control Registers in 64-Bit-Mode

In 64-bit mode, additional encodings are available to address up to eight additional control registers. The REX.R bit, in a REX prefix, is used to modify the ModRM *reg* field when that field encodes a control register, as shown in "REX Prefixes" in Volume 3. These additional encodings enable the processor to address CR8–CR15.

One additional control register, CR8, is defined in 64-bit mode for all hardware implementations, as described in "CR8 (Task Priority Register, TPR)," below. Access to the CR9–CR15 registers is implementation-dependent. Any attempt to access an unimplemented register results in an invalid-opcode exception (#UD).

3.1.5 CR8 (Task Priority Register, TPR)

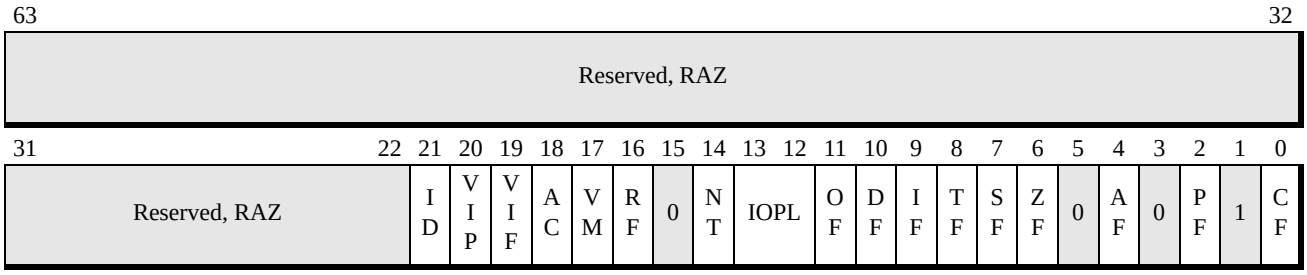
The AMD64 architecture introduces a new control register, CR8, defined as the task priority register (TPR). The register is accessible in 64-bit mode using the REX prefix. See Section 8.5.2 "External Interrupt Priorities," on page 258 for a description of the TPR and how system software can use the TPR for controlling external interrupts.

3.1.6 RFLAGS Register

The RFLAGS register contains two different types of information:

- *Control bits* provide system-software controls and directional information for string operations. Some of these bits can have privilege-level restrictions.
- *Status bits* provide information resulting from logical and arithmetic operations. These are written by the processor and can be read by software running at any privilege level.

Figure 3-7 on page 53 shows the format of the RFLAGS register. The legacy EFLAGS register is identical to the low 32 bits of the register shown in Figure 3-7 (RFLAGS bits 31:0). The term *rFLAGS* is used to refer to the 16-bit, 32-bit, or 64-bit flags register, depending on context.



Bits	Mnemonic	Description	Access type
63:22	—	Reserved	RAZ
21	ID	ID Flag	R/W
20	VIP	Virtual Interrupt Pending	R/W
19	VIF	Virtual Interrupt Flag	R/W
18	AC	Alignment Check	R/W
17	VM	Virtual-8086 Mode	R/W
16	RF	Resume Flag	R/W
15	—	Reserved	RAZ
14	NT	Nested Task	R/W
13:12	IOPL	I/O Privilege Level	R/W
11	OF	Overflow Flag	R/W
10	DF	Direction Flag	R/W
9	IF	Interrupt Flag	R/W
8	TF	Trap Flag	R/W
7	SF	Sign Flag	R/W
6	ZF	Zero Flag	R/W
5	—	Reserved	RAZ
4	AF	Auxiliary Flag	R/W
3	—	Reserved	RAZ
2	PF	Parity Flag	R/W
1	—	Reserved	RA1
0	CF	Carry Flag	R/W

Figure 3-7. RFLAGS Register

The functions of the RFLAGS control and status bits used by application software are described in “Flags Register” in Volume 1. The functions of RFLAGS system bits are (unless otherwise noted, all bits are read/write):

Trap Flag (TF) Bit. Bit 8. Software sets the TF bit to 1 to enable single-step mode during software debug. Clearing this bit to 0 disables single-step mode.

When single-step mode is enabled at the start of an instruction's execution, a debug exception (#DB) occurs immediately after the instruction completes execution. Single stepping is automatically disabled (TF is set to 0) when the #DB exception occurs or when any exception or interrupt occurs.

See Section 13.1.4 “Single Stepping,” on page 398 for information on using the single-step mode during debugging.

Interrupt Flag (IF) Bit. Bit 9. Software sets the IF bit to 1 to enable maskable interrupts. Clearing this bit to 0 causes the processor to ignore maskable interrupts. The state of the IF bit does not affect the response of a processor to non-maskable interrupts, software-interrupt instructions, or exceptions.

The ability to modify the IF bit depends on several factors:

- The current privilege-level (CPL)
- The I/O privilege level (RFLAGS.IOPL)
- Whether or not virtual-8086 mode extensions are enabled (CR4.VME=1)
- Whether or not protected-mode virtual interrupts are enabled (CR4.PVI=1)

See Section 8.1.4 “Masking External Interrupts,” on page 235 for information on interrupt masking. See Section 6.2.3 “Accessing the RFLAGS Register,” on page 175 for information on the specific instructions used to modify the IF bit.

I/O Privilege Level Field (IOPL) Field. Bits 13:12. The IOPL field specifies the privilege level required to execute I/O address-space instructions (i.e., instructions that address the I/O space rather than memory-mapped I/O, such as IN, OUT, INS, OUTS, etc.). For software to execute these instructions, the current privilege-level (CPL) must be equal to or higher than (lower numerical value than) the privilege specified by IOPL (CPL <= IOPL). If the CPL is lower than (higher numerical value than) that specified by the IOPL (CPL > IOPL), the processor causes a general-protection exception (#GP) when software attempts to execute an I/O instruction. See “Protected-Mode I/O” in Volume 1 for information on how IOPL controls access to address-space I/O.

Virtual-8086 mode uses IOPL to control virtual interrupts and the IF bit when virtual-8086 mode extensions are enabled (CR4.VME=1). The protected-mode virtual-interrupt mechanism (PVI) also uses IOPL to control virtual interrupts and the IF bit when PVI is enabled (CR4.PVI=1). See Section 8.10 “Virtual Interrupts,” on page 279 for information on how IOPL is used by the virtual interrupt mechanism.

Nested Task (NT) Bit. Bit 14, IRET reads the NT bit to determine whether the current task is nested within another task. When NT is set to 1, the current task is nested within another task. When NT is cleared to 0, the current task is at the top level (not nested).

The processor sets the NT bit during a task switch resulting from a CALL, interrupt, or exception through a task gate. When an IRET is executed from legacy mode while the NT bit is set, a task switch occurs. See Section 12.3.3 “Task Switches Using Task Gates,” on page 380 for information on switching tasks using task gates, and Section 12.3.4 “Nesting Tasks,” on page 382 for information on task nesting.

Resume Flag (RF) Bit. Bit 16. The RF bit, when set to 1, temporarily disables instruction breakpoint reporting to prevent repeated debug exceptions (#DB) from occurring. This allows an instruction

which had been inhibited by an instruction-breakpoint debug exception to be restarted by the debug exception handler.

The processor clears the RF bit after every instruction is successfully executed, except when the

instruction is:

- An IRET that sets the RF bit.
- JMP, CALL, or INT*n* through a task gate.

In both of the above cases, RF is not cleared to 0 until the *next* instruction successfully executes.

When an exception occurs (or when a string instruction is interrupted), the processor normally sets RF=1 in the RFLAGS image saved on the interrupt stack. However, when a #DB exception occurs as a result of an instruction breakpoint, the processor clears the RF bit to 0 in the interrupt-stack RFLAGS image.

For instruction restart to work properly following an instruction breakpoint, the #DB exception handler must set RF to 1 in the interrupt-stack RFLAGS image. When an IRET is later executed to return to the instruction that caused the instruction-breakpoint #DB exception, the set RF bit (RF=1) is loaded from the interrupt-stack RFLAGS image. RF is not cleared by the processor until the instruction causing the #DB exception successfully executes.

Virtual-8086 Mode (VM) Bit. Bit 17. Software sets the VM bit to 1 to enable virtual-8086 mode. Software clears the VM bit to 0 to disable virtual-8086 mode. System software can only change this bit using a task switch or an IRET. It cannot modify the bit using the POPFD instruction.

Alignment Check (AC) Bit. Bit 18. Software enables automatic alignment checking by setting the AC bit to 1 when CR0.AM=1. Alignment checking can be disabled by clearing either AC or CR0.AM to 0. When automatic alignment checking is enabled and the current privilege-level (CPL) is 3 (least privileged), a memory reference to an unaligned operand causes an alignment-check exception (#AC).

When the supervisor mode access prevention feature is enabled (CR4.SMAP=1), certain supervisor-mode data accesses to pages accessible by user mode are allowed only if RFLAGS.AC=1. See Section 5.6.6 “Supervisor-Mode Access Prevention(CR4.SMAP) Bit,” on page 160 for more information.

Virtual Interrupt (VIF) Bit. Bit 19. The VIF bit is a virtual image of the RFLAGS.IF bit. It is enabled when either virtual-8086 mode extensions are enabled (CR4.VME=1) or protected-mode virtual interrupts are enabled (CR4.PVI=1), and the RFLAGS.IOPL field is less than 3. When enabled, instructions that ordinarily would modify the IF bit actually modify the VIF bit with no effect on the RFLAGS.IF bit.

System software that supports virtual-8086 mode should enable the VIF bit using CR4.VME. This allows 8086 software to execute instructions that can set and clear the RFLAGS.IF bit without causing an exception. With VIF enabled in virtual-8086 mode, those instructions set and clear the VIF bit instead, giving the appearance to the 8086 software that it is modifying the RFLAGS.IF bit. System

software reads the VIF bit to determine whether or not to take the action desired by the 8086 software (enabling or disabling interrupts by setting or clearing the RFLAGS.IF bit).

In long mode, the use of the VIF bit is supported when CR4.PVI=1. See Section 8.10 “Virtual Interrupts,” on page 279 for more information on virtual interrupts.

Virtual Interrupt Pending (VIP) Bit. Bit 20. The VIP bit is provided as an extension to both virtual-8086 mode and protected mode. It is used by system software to indicate that an external, maskable interrupt is pending (awaiting) execution by either a virtual-8086 mode or protected-mode interrupt-service routine. Software must enable virtual-8086 mode extensions (CR4.VME=1) or protected-mode virtual interrupts (CR4.PVI=1) before using VIP.

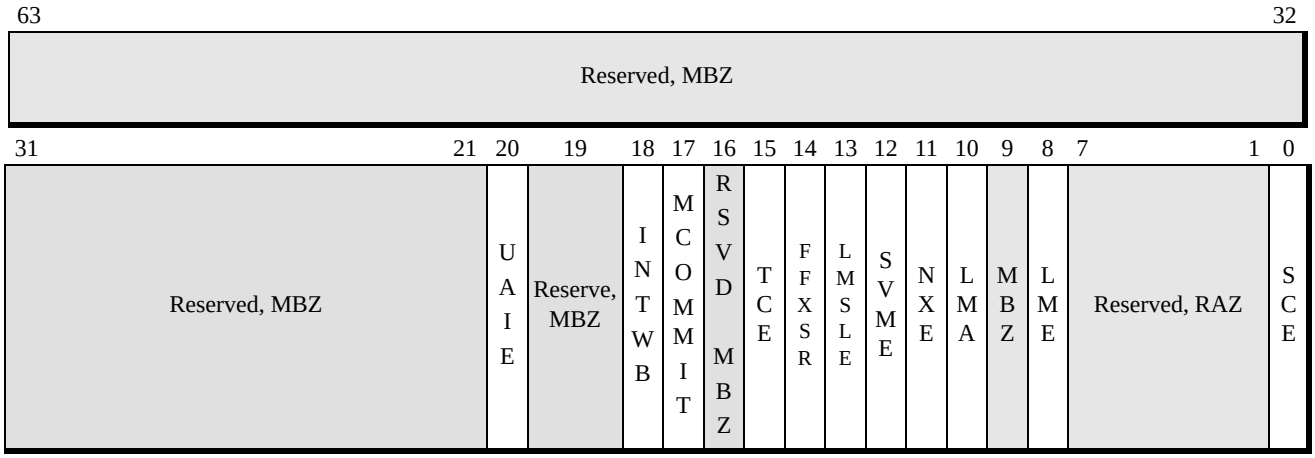
VIP is normally set to 1 by a protected-mode interrupt-service routine that was entered from virtual-8086 mode as a result of an external, maskable interrupt. Before returning to the virtual-8086 mode application, the service routine sets VIP to 1 if EFLAGS.VIF=1. When the virtual-8086 mode application attempts to enable interrupts by clearing EFLAGS.VIF to 0 while VIP=1, a general-protection exception (#GP) occurs. The #GP service routine can then decide whether to allow the virtual-8086 mode service routine to handle the pending external, maskable interrupt. (EFLAGS is specifically referred to in this case because virtual-8086 mode is supported only from legacy mode.)

In long mode, the use of the VIP bit is supported when CR4.PVI=1. See Section 8.10 “Virtual Interrupts,” on page 279 for more information on virtual-8086 mode interrupts and the VIP bit.

Processor Feature Identification (ID) Bit. Bit 21. The ability of software to modify this bit indicates that the processor implementation supports the CPUID instruction. See Section 3.3 “Processor Feature Identification,” on page 71 for more information on the CPUID instruction.

3.1.7 Extended Feature Enable Register (EFER)

The extended-feature-enable register (EFER) contains control bits that enable additional processor features not controlled by the legacy control registers. The EFER is a model-specific register (MSR) with an address of C000_0080h (see Section 3.2 “Model-Specific Registers (MSRs),” on page 59 for more information on MSRs). It can be read and written only by privileged software. Figure 3-8 on page 57 shows the format of the EFER register.



Bits	Mnemonic	Description	Access type
63:21	—	Reserved	MBZ
20	UAIE	Upper Address Ignore Enable	R/W
19	—	Reserved	MBZ
18	INTWB	Interruptible WBINVD/WBNOINVD enable	R/W
17	MCOMMIT	Enable MCOMMIT instruction	R/W
16	—	Reserved	MBZ
15	TCE	Translation Cache Extension	R/W
14	FFXSR	Fast FXSAVE/FXRSTOR	R/W
13	LMSLE	Long Mode Segment Limit Enable	R/W
12	SVME	Secure Virtual Machine Enable	R/W
11	NXE	No-Execute Enable	R/W
10	LMA	Long Mode Active	R/W
9	—	Reserved	MBZ
8	LME	Long Mode Enable	R/W
7:1	—	Reserved	RAZ
0	SCE	System Call Extensions	R/W

Figure 3-8. Extended Feature Enable Register (EFER)

The defined EFER bits shown in Figure 3-8 above are described below:

System-Call Extension (SCE) Bit. Bit 0, read/write. Setting this bit to 1 enables the SYSCALL and SYSRET instructions. Application software can use these instructions for low-latency system calls and returns in a non-segmented (flat) address space. See Section 6.1 “Fast System Call and Return,” on page 170 for additional information.

Long Mode Enable (LME) Bit. Bit 8, read/write. Setting this bit to 1 enables the processor to activate long mode. Long mode is not activated until software enables paging some time later. When paging is enabled after LME is set to 1, the processor sets the EFER.LMA bit to 1, indicating that long mode is not only enabled but also active. See Chapter 14, “Processor Initialization and Long Mode Activation,” for more information on activating long mode.

Long Mode Active (LMA) Bit. Bit 10, read/write. This bit indicates that long mode is active. The processor sets LMA to 1 when both long mode and paging have been enabled by system software. See Chapter 14, “Processor Initialization and Long Mode Activation,” for more information on activating long mode.

When LMA=1, the processor is running either in compatibility mode or 64-bit mode, depending on the value of the L bit in a code-segment descriptor, as shown in Figure 1-6 on page 12.

When LMA=0, the processor is running in legacy mode. In this mode, the processor behaves like a standard 32-bit x86 processor, with none of the new 64-bit features enabled. When writing the EFER register the value of this bit must be preserved. Software must read the EFER register to determine the value of LMA, change any other bits as required and then write the EFER register. An attempt to write a value that differs from the state determined by hardware results in a #GP fault.

No-Execute Enable (NXE) Bit. Bit 11, read/write. Setting this bit to 1 enables the no-execute page-protection feature. The feature is disabled when this bit is cleared to 0. See Section “No Execute (NX) Bit,” on page 152 for more information.

Before setting NXE, system software should verify the processor supports the feature by examining the feature flag CPUID Fn8000_0001_EDX[NX]. See Section 3.3 “Processor Feature Identification,” on page 71 for information on using the CPUID instruction.

Secure Virtual Machine Enable (SVME) Bit. Bit 12, read/write. Enables the SVM extensions. When this bit is zero, the SVM instructions cause #UD exceptions. EFER.SVME defaults to a reset value of zero. The effect of turning off EFER.SVME while a guest is running is undefined; therefore, the VMM should always prevent guests from writing EFER. SVM extensions can be disabled by setting VM_CR.SVME_DISABLE. For more information, see descriptions of LOCK and SMVE_DISABLE bits in Section 15.30.1 “VM_CR MSR (C001_0114h),” on page 566.

Long Mode Segment Limit Enable (LMSLE) bit. Bit 13, read/write. Setting this bit to 1 enables certain limit checks in 64-bit mode. This feature has been deprecated and is not supported by all processor implementations. If CPUID Fn8000_0008_EBX[EferLmlseUnsupported](bit 20)=1, 64-bit mode segment limit checking is not supported and attempting to set EFER.LMSLE =1 causes a #GP exception. See Section 4.12.2 “Data Limit Checks in 64-bit Mode,” on page 123, for more information on these limit checks.

Fast FXSAVE/FXRSTOR (FFXSR) Bit. Bit 14, read/write. Setting this bit to 1 enables the FXSAVE and FXRSTOR instructions to execute faster in 64-bit mode at CPL 0. This is accomplished by not saving or restoring the XMM registers (XMM0-XMM15). The FFXSR bit has no effect when the FXSAVE/FXRSTOR instructions are executed in non 64-bit mode, or when CPL > 0. The FFXSR bit does not affect the save/restore of the legacy x87 floating-point state, or the save/restore of MXCSR.

Before setting FFXSR, system software should verify whether this feature is supported by examining the feature flag CPUID Fn8000_0001_EDX[FFXSR]. See Section 3.3 “Processor Feature Identification,” on page 71 for information on using the CPUID instruction.

Translation Cache Extension (TCE) Bit. Bit 15, read/write. Setting this bit to 1 changes how the INVLPG, INVLPGB, and INVPCID instructions operate on TLB entries. When this bit is 0, these instructions remove the target PTE from the TLB as well as *all* upper-level table entries that are cached in the TLB, whether or not they are associated with the target PTE. When this bit is set, these instructions will remove the target PTE and only those upper-level entries that lead to the target PTE in the page table hierarchy, leaving unrelated upper-level entries intact. This may provide a performance benefit.

Page table management software must be written in a way that takes this behavior into account. Software that was written for a processor that does not cache upper-level table entries may result in stale entries being incorrectly used for translations when TCE is enabled. Software that is compatible with TCE mode will operate in either mode.

For software using INVLPGB to broadcast TLB invalidations, the invalidations are controlled by the EFER.TCE value on the processor executing the INVLPGB instruction.

Before setting TCE, system software should verify that this feature is supported by examining the feature flag CPUID Fn8000_0001_ECX[TCE]. See Section 3.3 “Processor Feature Identification,” on page 71 for information on using the CPUID instruction.

MCOMMIT ENABLE (MCOMMIT) Bit. Bit 17, read/write. Setting this bit to 1 enables the MCOMMIT instruction. When clear, attempting to execute MCOMMIT causes a #UD exception.

INTERRUPTIBLE WBINVD (INTWB) Bit. Bit 18. Setting this bit to 1 allows the WBINVD and WBNOINVD instructions to be interruptible. See WBINVD and WBNOINVD in Volume 3.

UPPER ADDRESS IGNORE ENABLE (UAIE) Bit. Bit 20. Setting this bit to 1 excludes bits [63:57] of an address from the canonical check for some memory references. Section 5.10 “Upper Address Ignore,” on page 164.

Extended control registers (XCR*n*) form a new register space that is available for managing processor architectural features and capabilities. Currently only XCR0 is defined. All other XCR registers are reserved. For more details on the Extended Control Registers, see “Extended Control Registers” in Volume 4, Chapter 1.

3.2 Model-Specific Registers (MSRs)

Processor implementations provide model-specific registers (MSRs) for software control over the unique features supported by that implementation. Software reads and writes MSRs using the privileged RDMSR and WRMSR instructions. Implementations of the AMD64 architecture can contain a mixture of two basic MSR types:

- *Legacy MSRs*. The AMD family of processors often share model-specific features with other x86 processor implementations. Where possible, AMD implementations use the same MSRs for the same functions. For example, the memory-typing and debug-extension MSRs are implemented on many AMD and non-AMD processors.

- *AMD model-specific MSRs*. There are many MSRs common to the AMD family of processors but not to legacy x86 processors. Where possible, AMD implementations use the same AMD-specific MSRs for the same functions.

Every model-specific register, as the name implies, is not necessarily implemented by all members of the AMD family of processors. Appendix A, “MSR Cross-Reference,” lists MSR-address ranges currently used by various AMD and other x86 processors.

The AMD64 architecture includes a number of features that are controlled using MSRs. Those MSRs are shown in Figure 3-9. The EFER register—described in Section 3.1.7 “Extended Feature Enable Register (EFER),” on page 56—is also an MSR.

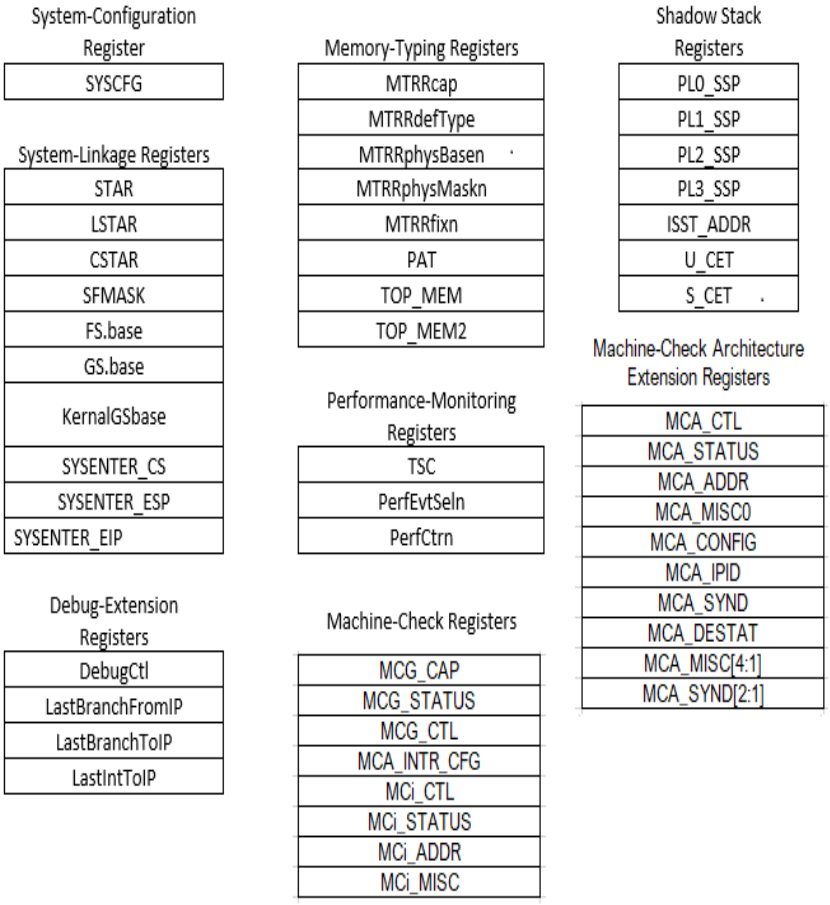
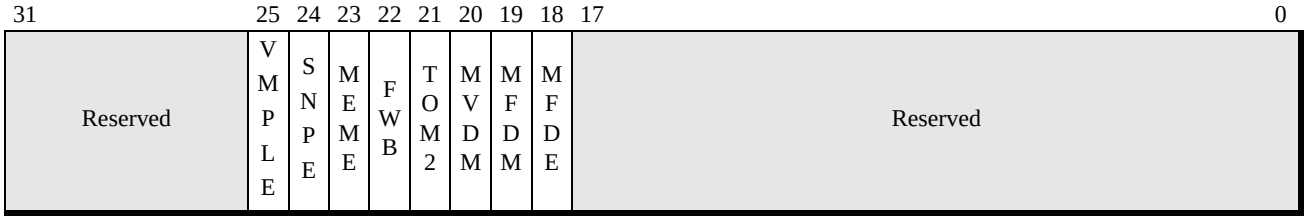


Figure 3-9. AMD64 Architecture Model-Specific Registers

The following sections briefly describe the MSRs in the AMD64 architecture.

3.2.1 System Configuration Register (SYSCFG)

The system-configuration register (SYSCFG) contains control bits for enabling and configuring system bus features. SYSCFG is a model-specific register (MSR) with an address of C001_0010h. Figure 3-10 on page 61 shows the format of the SYSCFG register. Some features are implementation specific, and are described in the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product. Implementation-specific features are not shown in Figure 3-10.



Bits	Mnemonic	Description	Access type
31:26	—	Reserved	MBZ
25	VMPLE	VMPLEn	R/W
24	SNPE	SecureNestedPagingEn	R/W
23	MEME	MemEncryptionModeEn	R/W
22	FWB	Tom2ForceMemTypeWB	R/W
21	TOM2	MtrrTom2En	R/W
20	MVDM	MtrrVarDramEn	R/W
19	MFDM	MtrrFixDramModEn	R/W
18	MFDE	MtrrFixDramEn	R/W
17:0	—	Reserved	MBZ

Figure 3-10. System-Configuration Register (SYSCFG)

The function of the SYSCFG bits are (all bits are read/write unless otherwise noted):

MtrrFixDramEn Bit. Bit 18. Setting this bit to 1 enables use of the RdMem and WrMem attributes in the fixed-range MTRR registers. When cleared, these attributes are disabled. The RdMem and WrMem attributes allow system software to define fixed-range IORRs using the fixed-range MTRRs. See Section 7.9.1 "Extended Fixed-Range MTRR Type-Field Encodings," on page 223 for information on using this feature.

MtrrFixDramModEn Bit. Bit 19. Setting this bit to 1 allows software to read and write the RdMem and WrMem bits. When cleared, writes do not modify the RdMem and WrMem bits, and reads return 0. See Section 7.9.1 "Extended Fixed-Range MTRR Type-Field Encodings," on page 223 for information on using this feature.

MtrrVarDramEn Bit. Bit 20. Setting this bit to 1 enables the TOP_MEM register and the variable-range IORRs. These registers are disabled when the bit is cleared to 0. See Section 7.9.2 “IORRs,” on page 224 and Section 7.9.4 “Top of Memory,” on page 226 for information on using these features.

MtrrTom2En Bit. Bit 21. Setting this bit to 1 enables the TOP_MEM2 register. The register is disabled when this bit is cleared to 0. See Section 7.9.4 “Top of Memory,” on page 226 for information on using this feature.

Tom2ForceMemTypeWB. Bit 22. Setting this bit to 1 forces the default memory type for memory between 4GB and the address specified by TOP_MEM2 to be write back instead of the memory type defined by MTRRdefType[Type]. For this bit to have any effect, MTRRdefType[E] must be 1. MTRR variable-range settings and PAT can be used to override this memory type.

MemEncryptionModeEn. Bit 23. Setting this bit to 1 enables the SME (Section 7.10 “Secure Memory Encryption,” on page 228) and SEV (Section 15.34 “Secure Encrypted Virtualization,” on page 571) memory encryption features. When cleared, these features are disabled. If MSR C001_0015[SmmLock] is set, the MemEncryptionModeEn bit is sticky and cannot be changed from a 1 to a 0.

SecureNestedPagingEn. Bit 24. Setting this bit to 1 enables SEV-SNP (Section 15.36 “Secure Nested Paging (SEV-SNP),” on page 585). When cleared, this feature is disabled. Once this bit is set to 1, it cannot be changed. This bit can only be set if MemEncryptionModeEn is already set or is simultaneously also set to 1. After SecureNestedPagingEn is set to 1, certain MSRs may no longer be written. See Section 15.36.2 “Enabling SEV-SNP,” on page 586 for details.

VMPLEn. Bit 25. Setting this bit to 1 enables the VMPL feature (Section 15.36.7 “Virtual Machine Privilege Levels,” on page 590). Software should set this bit to 1 when SecureNestedPagingEn is being set to 1. Once SecureNestedPagingEn is set to 1, VMPLEn cannot be changed.

3.2.2 System-Linkage Registers

System-linkage MSRs are used by system software to allow fast control transfers between applications and the operating system. The functions of these registers are:

STAR, LSTAR, CSTAR, and SFMASK Registers. These registers are used to provide mode-dependent linkage information for the SYSCALL and SYSRET instructions. STAR is used in legacy modes, LSTAR in 64-bit mode, and CSTAR in compatibility mode. SFMASK is used by the SYSCALL instruction for RFLAGS in long mode.

FS.base and GS.base Registers. These registers allow 64-bit base-address values to be specified for the FS and GS segments, for use in 64-bit mode. See Section “FS and GS Registers in 64-Bit Mode,” on page 80 for a description of the special treatment the FS and GS segments receive.

KernelGSbase Register. This register is used by the SWAPGS instruction. This instruction exchanges the value located in KernelGSbase with the value located in GS.base.

SYSENTER*x* Registers. The SYSENTER_CS, SYSENTER_ESP, and SYSENTER_EIP registers are used to provide linkage information for the SYSENTER and SYSEXIT instructions. These instructions are only used in legacy mode.

The system-linkage instructions and their use of MSRs are described in Section 6.1 “Fast System Call and Return,” on page 170.

3.2.3 Memory-Typing Registers

Memory-typing MSRs are used to characterize, or type, memory. Memory typing allows software to control the cacheability of memory, and determine how accesses to memory are ordered. The memory-typing registers perform the following functions:

MTRRcap Register. This register contains information describing the level of MTRR support provided by the processor.

MTRRdefType Register. This register establishes the default memory type to be used for physical memory that is not specifically characterized using the fixed-range and variable-range MTRRs.

MTRRphysBasen and MTRRphysMaskn Registers. These registers form a register pair that can be used to characterize any address range within the physical-memory space, including all of physical memory. Up to eight address ranges of varying sizes can be characterized using these registers.

MTRRfixn Registers. These registers are used to characterize fixed-size memory ranges in the first 1 Mbytes of physical-memory space.

PAT Register. This register allows memory-type characterization based on the virtual (linear) address. It is an extension to the PCD and PWT memory types supported by the legacy paging mechanism. The PAT mechanism provides the same memory-typing capabilities as the MTRRs, but with the added flexibility provided by the paging mechanism.

TOP_MEM and TOP_MEM2 Registers. These top-of-memory registers allow system software to specify physical addresses ranges as memory-mapped I/O locations.

Refer to Section 7.7 “Memory-Type Range Registers,” on page 208 for more information on using these registers.

3.2.4 Debug-Extension Registers

The debug-extension MSRs provide software-debug capability not available in the legacy debug registers (DR0–DR7). These MSRs allow single stepping and recording of control transfers to take place. The debug-extension registers perform the following functions:

DebugCtl Register. This MSR register provides control over control-transfer recording and single stepping, and external-breakpoint reporting and trace messages.

LastBranch*x* and LastInt*x* Registers. The four registers, LastBranchToIP, LastBranchFromIP, LastIntToIP, and LastIntFromIP, are all used to record the source and target of control transfers when branch recording is enabled.

Refer to Section 13.1.6 “Control-Transfer Breakpoint Features,” on page 398 for more information on using these debug registers.

3.2.5 Performance-Monitoring Registers

The time-stamp counter and performance-monitoring registers are useful in identifying performance bottlenecks. The number of performance counters can vary based on the implementation. These registers perform the following functions:

TSC Register. This register is used to count processor-clock cycles. It can be read using the RDMSR instruction, or it can be read using the either of the *read time-stamp counter* instructions, RDTSC or RDTSCP. System software can make RDTSC or RDTSCP available for use by non-privileged software by clearing the time-stamp disable bit (CR4.TSD) to 0.

***PerfEvtSel*n* Registers.** These registers are used to specify the events counted by the corresponding performance counter, and to control other aspects of its operation.

***PerfCtr*n* Registers.** These registers are performance counters that hold a count of processor, northbridge, or L2 cache events or the duration of events, under the control of the corresponding *PerfEvtSel*n* register. Each *PerfCtr*n* register can be read using the RDMSR instruction, or they can be read using the *read performance-monitor counter* instruction, RDPMC. System software can make RDPMC available for use by non-privileged software by setting the performance-monitor counter enable bit (CR4.PCE) to 1.

Refer to Section 13.2.3 “Using Performance Counters,” on page 407 for more information on using these registers.

3.2.6 Machine-Check Registers

The machine-check registers control the detection and reporting of hardware machine-check errors. The types of errors that can be reported include cache-access errors, load-data and store-data errors, bus-parity errors, and ECC errors. Three types of machine-check MSRs are shown in Figure 3-9 on page 60.

The first type is global machine-check registers, which perform the following functions:

MCG_CAP Register. This register identifies the machine-check capabilities supported by the processor.

MCG_CTL Register. This register provides global control over machine-check-error reporting.

MCG_STATUS Register. This register reports global status on detected machine-check errors.

The second type is error-reporting register banks, which report on machine-check errors associated with a specific processor unit (or group of processor units). There can be different numbers of register banks for each processor implementation, and each bank is numbered from 0 to *i*. The registers in each bank perform the following functions:

MC*i*_CTL Registers. These registers control error-reporting.

MC*i*_STATUS Registers. These registers report machine-check errors.

MC*i*_ADDR Registers. These registers report the machine-check error address.

MC*i*_MISC Registers. These registers report miscellaneous-error information.

The third type is MCA Extension (MCAX) register banks, which report on machine-check errors associated with a specific processor unit (or group of processor units). There can be different numbers of register banks for each processor implementation, and each bank is numbered from 0 to i. Legacy MCA supports up to 32 banks. MCAX bank 0 will alias to Legacy bank 0 registers. Similarly, MCAX bank n will alias to Legacy bank n registers. The registers in each bank perform the following functions:

MCA_CTL Register. This register is an alias to MCi_CTL for banks 0 to 31. For banks 32 and above, this register controls error-reporting.

MCA_STATUS Register. This register is an alias to MCi_Status.

MCA_ADDR Register. This register is an alias to MCi_ADDR.

MCA_MISC0 Register. This register is an alias to MCi_MISC0.

MCA_CONFIG Register. This register holds configuration information for the MCA bank.

MCA_IPID Register. This register holds information which identifies the specific MCA bank.

MCA_SYND Register. This register stores information associated with the error in MCA_STATUS or MCA_DESTAT.

MCA_DESTAT Register. This register reports deferred machine check errors.

MCA_DEADDR Register. This register provides the address associated with the deferred machine check error.

MCA_MISC[4:1] Registers. Extended miscellaneous error-information registers.

MCA_SYND[2:1] Registers. This register contains information associated with the error in MCA_STATUS or MCA_DESTAT.

Refer to Section 9.5 “Using MCA Features,” on page 312 for more information on using these registers.

3.2.7 Shadow Stack Registers

These registers are defined if the shadow stack feature is supported as indicated by CPUID Fn 0000_0007_0 ECX[CET_SS] (bit 7) = 1.

PL0_SSP, PL1_SSP, PL2_SSP Registers. These registers specify the linear address to be loaded into SSP on the next transition to CPLn, where n=0, 1, 2. The linear address must be in canonical format and aligned to 4 bytes.

PL3_SSP Register. The user mode SSP is saved to and restored from this register. The linear address must be in canonical format and aligned to 4 bytes.

ISST_ADDR Register. This register specifies the linear address of the Interrupt SSP Table (ISST). The linear address must be in canonical format.

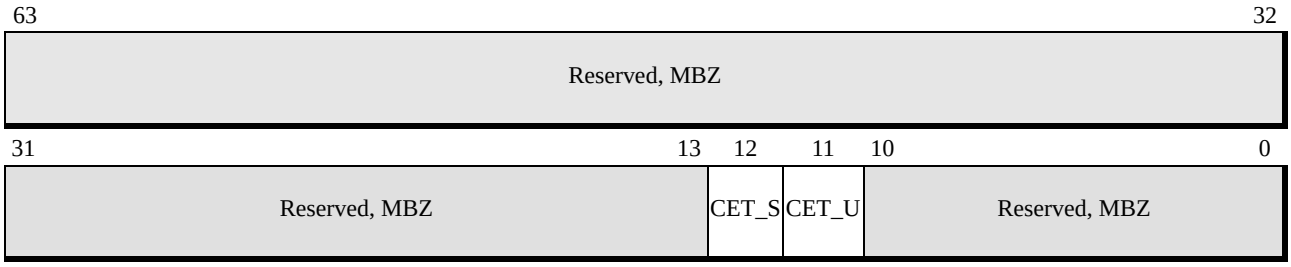
U_CET Register. This register specifies the user mode shadow stack controls.

S_CET Register. This register specifies the supervisor mode shadow stack controls.

3.2.8 Extended State Save MSRs

XSS Register. This register contains a bitmap of supervisor-level state components. System software sets bits in the XSS register bitmap to enable management of corresponding state component by the XSAVES/XRSTORS instructions. XSS register support is indicated by CPUID Fn0000_000D_EAX[XSAVES]_x1 = 1.

The XSS bitmap is defined as follows:



Bits	Mnemonic	Description
63:13	—	Reserved, Must be Zero
12	CET_S	Enables the CET_U state component.
11	CET_U	Enables the CET_S state component.
10:0	—	Reserved, Must be Zero

Figure 3-11. XSS Register

3.2.9 Speculation Control MSRs

Modern processors implement hardware techniques such as branch prediction, speculative execution and out-of-order processing to significantly improve performance. If the processor incorrectly predicts or speculates on an outcome, this is detected and any speculative results are discarded. The processor then supplies the architecturally correct, in-order response to the program's instructions. Even though the speculative results are discarded, microarchitectural side effects may remain which can be detected by software, and which in some cases may lead to side-channel vulnerabilities.

The two speculation control MSRs, SPEC_CTRL (MSR 048h) and PRED_CMD (MSR 049h), enable hardware features that are designed to limit certain types of speculation. Support for these features is indicated by CPUID Fn8000_0008_EBX as described in Table 3-1 below. The presence of a given speculation control feature also implies support for its associated MSR.

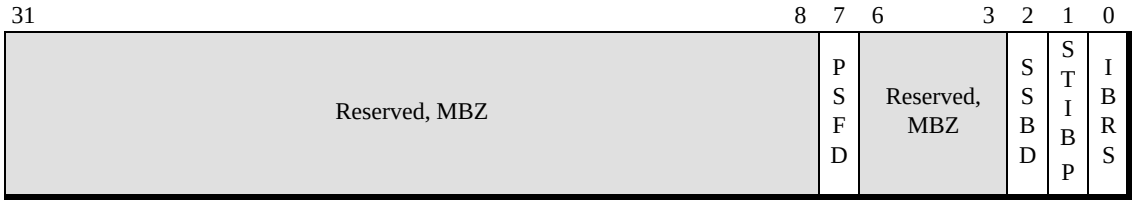
Table 3-1. Speculation Control MSRs

Feature	Indicated by CPUID Function	MSR
IBRS	Fn8000_0008 EBX[14]=1	SPEC_CTRL (MSR 048h)
STIBP	Fn8000_0008 EBX[15]=1	SPEC_CTRL (MSR 048h)
SSBD	Fn8000_0008 EBX[24]=1	SPEC_CTRL (MSR 048h)
PSFD	Fn8000_0008 EBX[28]=1	SPEC_CTRL (MSR 048h)
IBPB	Fn8000_0008 EBX[12]=1	PRED_CMD (MSR 049h)

See ‘Additional CPUID Functions’ at the end of this subsection and APM volume 3 Appendix E.4 ‘CPUID Fn8000_0008_EBX’ for more information on CPUID functions related to speculation control.

SPEC_CTRL (MSR 048h)

SPEC_CTRL (MSR 48h) is a read-write register. Attempts to write a 1 into any reserved bit cause a #GP(0) fault. Unlike most MSRs, a WRMSR to SPEC_CTRL does not serialize memory operations. However, a write to this register is dispatch serializing and prevents execution of younger instructions until the WRMSR has completed. The format of SPEC_CTRL is shown in Figure 3-12.



Bits	Mnemonic	Description	Access type
31:8	—	Reserved	Must be Zero
7	PSFD	Predicted Store Forward Disable.	R/W
6:3	—	Reserved	Must be Zero
2	SSBD	Speculative Store Bypass Disable.	R/W
1	STIBP	Single-Thread Indirect Branch Prediction Mode.	R/W
0	IBRS	Indirect Branch Restricted Speculation.	R/W

Figure 3-12. SPEC_CTRL Register (MSR 048h)

The SPEC_CTRL bits defined in Figure 3-1 are described below:

Indirect Branch Restricted Speculation (IBRS). Bit 0. Setting this bit to 1 prevents indirect branches that occurred in a less privileged prediction mode before this bit was set from influencing the predictions of future indirect branches in a more privileged prediction mode that occur after this bit is set. A lesser privileged prediction mode is defined as CPL 3 or Guest mode, and a more privileged prediction mode is defined as CPL 0-2 or Host mode.

After setting IBRS to 1, if software subsequently

- clears IBRS to 0: The processor may allow older indirect branches that occurred when IBRS was previously 0 to influence future indirect branch predictions.
- writes another 1 to IBRS: The processor starts a new window where older indirect branches do not influence future indirect branch predictions.

Only indirect branches that occurred prior to setting IBRS are prevented from influencing future indirect branches. Therefore, if IBRS were already set on a transition to a more privileged mode from a lesser privileged mode, software at the more privileged mode must write a 1 to IBRS if it requires indirect branch predictions in the new mode to not be influenced by those from the previous mode.

On processors with a shared indirect branch predictor, setting IBRS also prevents indirect branch predictions of one thread from influencing the predictions of its sibling threads, as if SPEC_CTRL[STIBP] was also set. For more information on STIBP, see Single Thread Indirect Branch Predictor below.

Some processors, identified by CPUID Fn8000_0008_EBX[IbrsSameMode] (bit 19) = 1, provide additional speculation limits. For these processors, when IBRS is set, indirect branch predictions are not influenced by any prior indirect branches, regardless of mode (CPL and guest/host) and regardless of whether the prior indirect branches occurred before or after the setting of IBRS.

Although return instructions can be considered a type of indirect branch, IBRS does not affect them. Software requiring IBRS-style indirect branch speculation limits for RET instructions should clear out any return address predictions by executing 32 CALL instructions having a non-zero displacement. Processors implementing more than 32 return predictions include hardware to clear the additional entries when software writes a 1 to IBRS. If the kernel and user virtual address spaces are disjoint with at least one unmapped 4K page separating them, and SMEP is enabled, then there is no need to clear out the return address predictions.

Single Thread Indirect Branch Prediction mode (STIBP). Bit 1. Setting this bit to 1 prevents indirect branch predictions of one thread from influencing the predictions of any sibling threads, on processors where branch prediction resources are shared. RET (return) instructions are not influenced by sibling threads. Therefore, setting STIBP is not required to prevent one thread's RET predictions from influencing the predictions of a sibling thread.

Note that STIBP mode is automatically enabled when SPEC_CTRL[IBRS] is set, regardless of value of SPEC_CTRL[STIBP].

Speculative Store Bypass Disable (SSBD). Bit 2. Setting this bit to 1 prevents load-type instructions from speculatively bypassing older store instructions whose final address have not yet been resolved.

As specified in Section 7.1.1 "Read Ordering," on page 184, loads from memory marked with the proper memory type can read memory out-of-order, speculatively and before older stores have completed. This means it is possible for a load to read and pass forward, in a speculative manner, previous values of the memory location. The processor has logic to correct this occurrence and provide the proper in-order load response to the program. However, this mis-speculation may have resulted in microarchitectural side effects. When SSBD is set to 1, the processor can return speculative load data only if there are no older stores with unknown addresses.

Some legacy processors implement SSBD in a different MSR. On these processors, indicated by CPUID function 8000_0008, EBX[25]=1, SSBD is enabled by setting VIRT_SPEC_CTRL (MSR C001_011F) bit 2. On processors that support both SPEC_CTRL and VIRT_SPEC_CTRL, if SSBD is enabled in either MSR, the processor prevents loads from speculating around older stores. However, it is preferred that software uses SPEC_CTRL[SSBD] in this scenario.

On some processor models, setting SSBD is not needed to prevent speculative loads from bypassing older stores. This is indicated by CPUID Fn8000_0008_EBX[SsbdNotRequired] (bit 26) = 1.

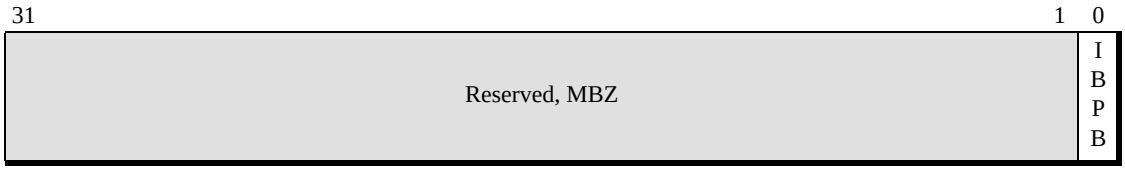
Predicted Store Forward Disable (PSFD). Bit 7. Setting this bit disables Predictive Store Forwarding (PSF).

As specified in Section 7.1.1 "Read Ordering," on page 184, write data for cacheable memory types can be forwarded to read instructions before that data is actually written to memory, via a mechanism called store-to-load forwarding. PSF expands on store-to-load forwarding via a mechanism that predicts the relationship between loads and stores based on past behavior, without waiting for their address calculations to complete. Like other forms of speculative store bypass, an incorrect prediction may result in microarchitectural side effects. PSF speculation can be disabled by setting PFSD.

The PSF feature is also disabled when SPEC_CTRL[SSBD] is set. However, SSBD disables both PSF and speculative store bypass, while PSFD only disables PSF. PSFD may be desirable for software which is concerned with the speculative behavior of PSF but desires a smaller performance impact than setting SSBD.

PRED_CMD (MSR 049h)

PRED_CMD is a write-only register. Attempts to read this register or to write a 1 into any reserved bit cause a #GP(0) fault. Unlike most MSRs, a WRMSR to PRED_CMD does not serialize memory operations. However, a write to this register is dispatch serializing and prevents execution of younger instructions until the WRMSR has completed. The format of the PRED_CMD register is shown in Figure 3-13 below.



Bits	Mnemonic	Description	Access type
31:1	—	Reserved	MBZ
0	IBPB	Indirect Branch Prediction Barrier.	W

Figure 3-13. PRED_CMD Register (MSR 049h)

Indirect Branch Prediction Barrier (IBPB). Bit 0, write only. Setting this bit to 1 prevents the processor from using older indirect branch predictions to influence future indirect branches. This applies to JMP indirect, CALL indirect and to RET (return) instructions. In some implementations, setting IBPB causes the processor to flush all previous indirect branch predictions. As this restricts the processor from using any previous indirect branch information, IBPB is intended to be used by software when switching between contexts that do not trust each other. Examples of such contexts include switching from one user context to another, or from one guest to another. In some implementations, IBPB will only flush the indirect predictions that are accessible to the current thread.

Additional CPUID Functions

The following CPUID functions provide more information on the speculation control features to aid system software in optimizing processor performance:

- CPUID Function 8000_0008_EBX[16] (IBRS always on). When set, indicates that the processor prefers that IBRS is only set once during boot and not changed. If IBRS is set on a processor supporting IBRS always on mode, indirect branches executed in a less privileged prediction mode will not influence branch predictions for indirect branches in a more privileged prediction mode. This eliminates the need for WRMSR instructions to manage speculation effects at elevated-privilege entry and exit points.

- CPUID Function 8000_0008_EBX[17] (STIBP always on). When set, indicates that the processor prefers that STIBP is only set once during boot and not changed. This eliminates the need for a WRMSR at the necessary transition points.
- CPUID Function 8000_0008_EBX[18] (IBRS preferred). When set, indicates that the processor prefers using the IBRS feature instead of other software mitigations such as retpoline. This allows software to remove the software mitigation and utilize the more performant IBRS mechanism.

3.2.10 Hardware Configuration Register (HWCR)

The HWCR register contains control bits that affect the functionality of other features. Some HWCR bits are implementation specific, and are described in the BIOS and Kernel Developer's Guide (BKDG) or Processor Programming Reference Manual applicable to your product. Implementation specific HWCR bits are not listed below.

SmmLock. Bit 0. Enables SMM code lock. When set to 1, SMM code in the ASeg and TSeg memory ranges, and the SMM registers, become read-only and SMI interrupts are not intercepted in SVM. See Section 15.32 "SMM-Lock," on page 570 for information on using this feature.

CpbDis. Bit 25. Core performance boost disable. When set to 1, core performance boost is disabled. See Section 17.2 "Core Performance Boost," on page 641 for further details on this feature.

IRPerfEn. Bit 30. Setting this bit to 1 enables the instructions retired counter. See Section 13.2.1 "Performance Counter MSRs," on page 401 for information on using this feature.

SmmPgCfgLock. Bit 33. Setting this bit to 1 locks the paging configuration while in SMM. See Section 10.3.9 "SMM Page Configuration Lock," on page 332 for information on using this feature.

3.3 Processor Feature Identification

The CPUID instruction provides information about the processor implementation and its capabilities. Software operating at any privilege level can execute the CPUID instruction to collect this information. Software can utilize this information to optimize performance.

The CPUID instruction supports multiple functions, each providing specific information about the processor implementation, including the vendor, model number, revision (stepping), features, cache organization, and name. The multifunction approach allows the CPUID instruction to return a detailed picture of the processor implementation and its capabilities—more detailed information than could be returned by a single function. This flexibility also allows for the addition of new CPUID functions in future processor generations.

The desired function number is loaded into the EAX register before executing the CPUID instruction. CPUID functions are divided into two types:

- *Standard functions* return information about features common to all x86 implementations, including the earliest features offered in the x86 architecture, as well as information about the

presence of features such as support for the AVX and FMA instruction subsets. Standard function numbers are in the range 0000_0000h–0000_FFFFh.

- *Extended functions* return information about AMD-specific features such as long mode and the presence of features such as support for the FMA4 and XOP instruction subsets. Extended function numbers are in the range 8000_0000h–8000_FFFFh.

Feature information is returned in the EAX, EBX, ECX, and EDX registers. Some functions accept a second input parameter passed to the instruction in the ECX register.

In this and the other three volumes of this *Programmer's Manual*, the notation *CPUID FnXXXX_XXXX_RRR[FieldName]_xYY* is used to represent the input parameters and return value that corresponds to a particular processor capability or feature.

In this notation, *XXXX_XXXX* represents the 32-bit value to be placed in the EAX register prior to executing the CPUID instruction. This value is the function number. *RRR* is either EAX, EBX, ECX, or EDX and represents the register to be examined after the execution of the instruction. If the contents of the entire 32-bit register provides the capability information, the notation *[FieldName]* is omitted, otherwise this provides the name of the field within the return value that represents the capability or feature.

When the field is a single bit, this is called a feature flag. Normally, if a feature flag bit is set, the corresponding processor feature is supported and if it is cleared, the feature is not supported. The optional input parameter passed to the CPUID instruction in the ECX register is represented by the notation _x*YY* appended after the return value notation. If a CPUID function does not accept this optional input parameter, this notation is omitted.

For more specific information on the CPUID instruction, see the instruction reference page in Volume 3. For a description of all feature flags related to instruction subset support, see Volume 3, Appendix D, "Instruction Subsets and CPUID Feature Flags." For a comprehensive list of all processor capabilities and feature flags, see Volume 3, Appendix E, "Obtaining Processor Information Via the CPUID Instruction."

4 Segmented Virtual Memory

The legacy x86 architecture supports a segment-translation mechanism that allows system software to relocate and isolate instructions and data anywhere in the virtual-memory space. A segment is a contiguous block of memory within the linear address space. The size and location of a segment within the linear address space is arbitrary. Instructions and data can be assigned to one or more memory segments, each with its own protection characteristics. The processor hardware enforces the rules dictating whether one segment can access another segment.

The segmentation mechanism provides ten segment registers, each of which defines a single segment. Six of these registers (CS, DS, ES, FS, GS, and SS) define user segments. User segments hold software, data, and the stack and can be used by both application software and system software. The remaining four segment registers (GDT, LDT, IDT, and TR) define system segments. System segments contain data structures initialized and used only by system software. Segment registers contain a *base address* pointing to the starting location of a segment, a *limit* defining the segment size, and *attributes* defining the segment-protection characteristics.

Although segmentation provides a great deal of flexibility in relocating and protecting software and data, it is often more efficient to handle memory isolation and relocation with a combination of software and hardware paging support. For this reason, most modern system software bypasses the segmentation features. However, segmentation cannot be completely disabled, and an understanding of the segmentation mechanism is important to implementing long-mode system software.

In long mode, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode:

- In compatibility mode, segmentation functions just as it does in legacy mode, using legacy 16-bit or 32-bit protected mode semantics.
- 64-bit mode, segmentation is disabled, creating a flat 64-bit virtual-address space. As will be seen, certain functions of some segment registers, particularly the system-segment registers, continue to be used in 64-bit mode.

4.1 Real Mode Segmentation

After reset or power-up, the processor always initially enters real mode. Protected modes are entered from real mode.

As noted in “Real Addressing” on page 10, real mode (real-address mode), provides a physical-memory space of 1 Mbyte. In this mode, a 20-bit physical address is determined by shifting a 16-bit segment selector to the left four bits and adding the 16-bit effective address.

Each 64K segment (CS, DS, ES, FS, GS, SS) is aligned on 16-byte boundaries. The *segment base* is the lowest address in a given segment, and is equal to the segment selector * 16. The POP and MOV instructions can be used to load a (possibly) new segment selector into one of the segment registers.

When this occurs, the selector is updated and the segment base is set to selector * 16. The segment limit and segment attributes are unchanged, but are normally 64K (the maximum allowable limit) and read/write data, respectively.

On FAR transfers, CS (code segment) selector is updated to the new value, and the CS segment base is set to selector * 16. The CS segment limit and attributes are unchanged, but are usually 64K and read/write, respectively.

If the interrupt descriptor table (IDT) is used to find the real mode IDT see “Real-Mode Interrupt Control Transfers” on page 259.

The GDT, LDT, and TSS (see below) are not used in real mode.

4.2 Virtual-8086 Mode Segmentation

Virtual-8086 mode supports 16-bit real mode programs running under protected mode (see below). It uses a simple form of memory segmentation, optional paging, and limited protection checking. Programs running in virtual-8086 mode can access up to 1MB of memory space.

As with real mode segmentation, each 64K segment (CS, DS, ES, FS, GS, SS) is aligned on 16-byte boundaries. The *segment base* is the lowest address in a given segment, and is equal to the segment selector * 16. The POP and MOV instructions work exactly as in real mode and can be used to load a (possibly) new segment selector into one of the segment registers. When this occurs, the selector is updated and the segment base is set to selector * 16. The segment limit and segment attributes are unchanged, but are normally 64K (the maximum allowable limit) and read/write data, respectively.

FAR transfers, with the exception of interrupts and exceptions, operate as in real mode. On FAR transfers, the CS (code segment) selector is updated to the new value, and the CS segment base is set to selector * 16. The CS segment limit and attributes are unchanged, but are usually 64K and read/write, respectively. Interrupts and exceptions switch the processor to protected mode. (See Chapter 8, “Exceptions and Interrupts” for more information.)

4.3 Protected Mode Segmented-Memory Models

System software can use the segmentation mechanism to support one of two basic segmented-memory models: a flat-memory model or a multi-segmented model. These segmentation models are supported in legacy mode and in compatibility mode. Each type of model is described in the following sections.

4.3.1 Multi-Segmented Model

In the multi-segmented memory model, each segment register can reference a unique base address with a unique segment size. Segments can be as small as a single byte or as large as 4 Gbytes. When page translation is used, multiple segments can be mapped to a single page and multiple pages can be mapped to a single segment. Figure 1-1 on page 6 shows an example of the multi-segmented model.

The multi-segmented memory model provides the greatest level of flexibility for system software using the segmentation mechanism.

Compatibility mode allows the multi-segmented model to be used in support of legacy software. However, in compatibility mode, the multi-segmented memory model is restricted to the first 4 Gbytes of virtual-memory space. Access to virtual memory above 4 Gbytes requires the use of 64-bit mode, which does not support segmentation.

4.3.2 Flat-Memory Model

The flat-memory model is the simplest form of segmentation to implement. Although segmentation cannot be disabled, the flat-memory model allows system software to bypass most of the segmentation mechanism. In the flat-memory model, all segment-base addresses have a value of 0 and the segment limits are fixed at 4 Gbytes. Clearing the segment-base value to 0 effectively disables segment translation, resulting in a single segment spanning the entire virtual-address space. All segment descriptors reference this single, flat segment. Figure 1-2 on page 7 shows an example of the flat-memory model.

4.3.3 Segmentation in 64-Bit Mode

In 64-bit mode, segmentation is disabled. The segment-base value is ignored and treated as 0 by the segmentation hardware. Likewise, segment limits and most attributes are ignored. There are a few exceptions. The CS-segment DPL, D, and L attributes are used (respectively) to establish the privilege level for a program, the default operand size, and whether the program is running in 64-bit mode or compatibility mode. The FS and GS segments can be used as additional base registers in address calculations, and those segments can have non-zero base-address values. This facilitates addressing thread-local data and certain system-software data structures. See “FS and GS Registers in 64-Bit Mode” on page 80 for details about the FS and GS segments in 64-bit mode. The system-segment registers are always used in 64-bit mode.

4.4 Segmentation Data Structures and Registers

Figure 4-1 on page 76 shows the following data structures used by the segmentation mechanism:

- *Segment Descriptors*—As the name implies, a segment descriptor *describes* a segment, including its location in virtual-address space, its size, protection characteristics, and other attributes.
- *Descriptor Tables*—Segment descriptors are stored in memory in one of three tables. The global-descriptor table (GDT) holds segment descriptors that can be shared among all tasks. Multiple local-descriptor tables (LDT) can be defined to hold descriptors that are used by specific tasks and are not shared globally. The interrupt-descriptor table (IDT) holds gate descriptors that are used to access the segments where interrupt handlers are located.
- *Task-State Segment*—A task-state segment (TSS) is a special type of system segment that contains task-state information and data structures for each task. For example, a TSS holds a copy of the GPRs and EFLAGS register when a task is suspended. A TSS also holds the pointers to privileged-

software stacks. The TSS and task-switch mechanism are described in Chapter 12, “Task Management.”

- *Segment Selectors*—Descriptors are selected for use from the descriptor tables using a segment selector. A segment selector contains an index into either the GDT or LDT. The IDT is indexed using an interrupt vector, as described in “Legacy Protected-Mode Interrupt Control Transfers” on page 261, and in “Long-Mode Interrupt Control Transfers” on page 272.

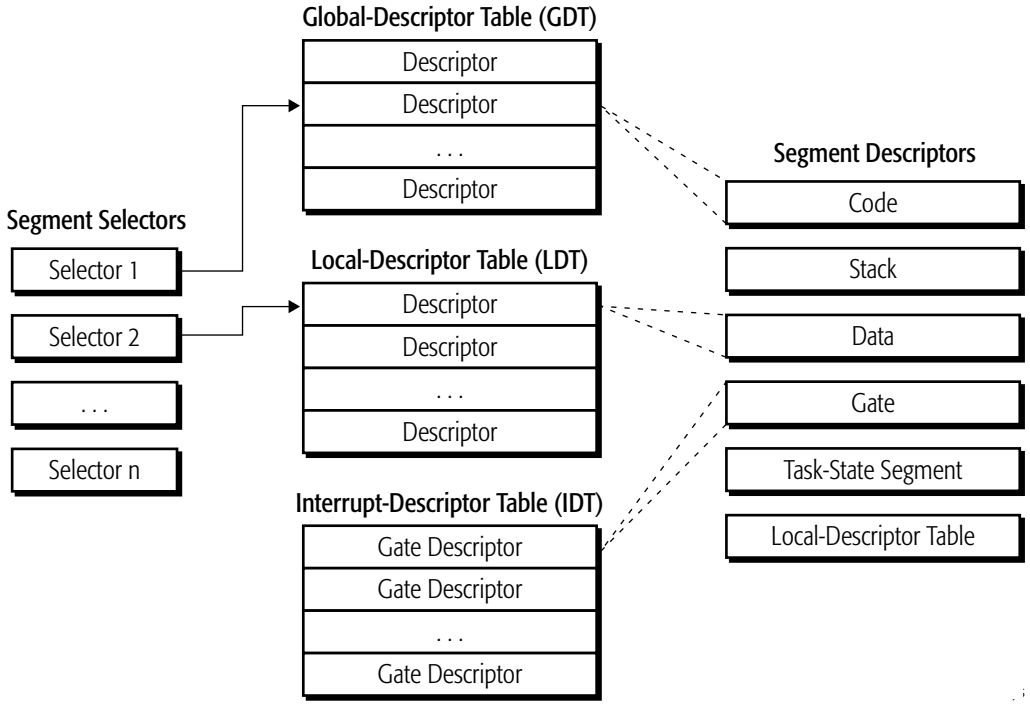


Figure 4-1. Segmentation Data Structures

Figure 4-2 on page 77 shows the registers used by the segmentation mechanism. The registers have the following relationship to the data structures:

- *Segment Registers*—The six segment registers (CS, DS, ES, FS, GS, and SS) are used to point to the user segments. A segment selector selects a descriptor when it is loaded into one of the segment registers. This causes the processor to automatically load the selected descriptor into a software-invisible portion of the segment register.
- *Descriptor-Table Registers*—The three descriptor-table registers (GDTR, LDTR, and IDTR) are used to point to the system segments. The descriptor-table registers identify the virtual-memory location and size of the descriptor tables.
- *Task Register (TR)*—Describes the location and limit of the current task state segment (TSS).

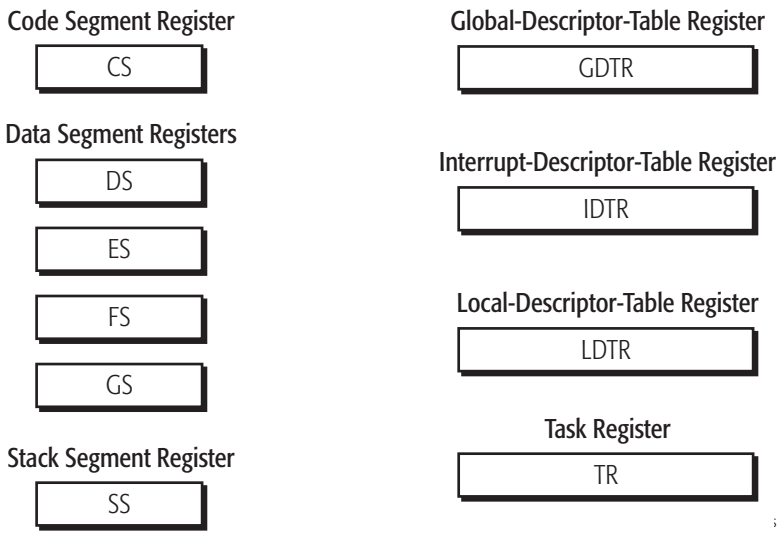


Figure 4-2. Segment and Descriptor-Table Registers

A fourth system-segment register, the TR, points to the TSS. The data structures and registers associated with task-state segments are described in “Task-Management Resources” on page 362.

4.5 Segment Selectors and Registers

4.5.1 Segment Selectors

Segment selectors are pointers to specific entries in the global and local descriptor tables. Figure 4-3 shows the segment selector format.

Bits	Mnemonic	Description	R/W
15:3	SI	Selector Index	R/W
2	TI	Table Indicator	R/W
1:0	RPL	Requestor Privilege Level	R/W

Figure 4-3. Segment Selector

The selector format consists of the following fields:

Selector Index Field. Bits 15:3. The selector-index field specifies an entry in the descriptor table. Descriptor-table entries are eight bytes long, so the selector index is scaled by 8 to form a byte offset into the descriptor table. The offset is then added to either the global or local descriptor-table base address (as indicated by the table-index bit) to form the descriptor-entry address in virtual-address space.

Some descriptor entries in long mode are 16 bytes long rather than 8 bytes (see “Legacy Segment Descriptors” on page 88 for more information on long-mode descriptor-table entries). These expanded descriptors consume two entries in the descriptor table. Long mode, however, continues to scale the selector index by eight to form the descriptor-table offset. It is the responsibility of system software to assign selectors such that they correctly point to the start of an expanded entry.

Table Indicator (TI) Bit. Bit 2. The TI bit indicates which table holds the descriptor referenced by the selector index. When TI=0 the GDT is used and when TI=1 the LDT is used. The descriptor-table base address is read from the appropriate descriptor-table register and added to the scaled selector index as described above.

Requestor Privilege-Level (RPL) Field. Bits 1:0. The RPL represents the privilege level (CPL) the processor is operating under at the time the selector is created.

RPL is used in segment privilege-checks to prevent software running at lesser privilege levels from accessing privileged data. See “Data-Access Privilege Checks” on page 106 and “Control-Transfer Privilege Checks” on page 109 for more information on segment privilege-checks.

Null Selector. Null selectors have a selector index of 0 and TI=0, corresponding to the first entry in the GDT. However, null selectors do not reference the first GDT entry but are instead used to invalidate unused segment registers. A general-protection exception (#GP) occurs if a reference is made to use a segment register containing a null selector in non-64-bit mode. By initializing unused segment registers with null selectors software can trap references to unused segments.

Null selectors can only be loaded into the DS, ES, FS and GS data-segment registers, and into the LDTR descriptor-table register. A #GP occurs if software attempts to load the CS register with a null selector or if software attempts to load the SS register with a null selector in non 64-bit mode or at CPL 3.

If CPUID Fn8000_0021_EAX[NullSelectorClearsBase] (bit 6) = 1, loading a segment register with a null selector clears the base address and limit of the segment register in all cases except a load of DS, ES, FS, or GS by an IRET, IRETD, IRETQ, or RETF instruction that changes the current privilege level, in which case these fields are left untouched. If CPUID Fn8000_0021_EAX[NullSelectorClearsBase] (bit 6) = 0, loading a segment register with a null selector makes the base address and limit of the segment register undefined. Because references to segment registers containing a null selector cause a #GP exception, the segment base and limit values have no effect. However, OS management of segment state may be simplified for processors supporting this clearing functionality.

4.5.2 Segment Registers

Six 16-bit segment registers are provided for referencing up to six segments at one time. All software tasks require segment selectors to be loaded in the CS and SS registers. Use of the DS, ES, FS, and GS segments is optional, but nearly all software accesses data and therefore requires a selector in the DS register. Table 4-1 on page 79 lists the supported segment registers and their functions.

Table 4-1. Segment Registers

Segment Register	Encoding	Segment Register Function
ES	/0	References optional data-segment descriptor entry
CS	/1	References code-segment descriptor entry
SS	/2	References stack segment descriptor entry
DS	/3	References default data-segment descriptor entry
FS	/4	References optional data-segment descriptor entry
GS	/5	References optional data-segment descriptor entry

The processor maintains a *hidden portion* of the segment register in addition to the selector value loaded by software. This hidden portion contains the values found in the descriptor-table entry referenced by the segment selector. The processor loads the descriptor-table entry into the hidden portion when the segment register is loaded. By keeping the corresponding descriptor-table entry in hardware, performance is optimized for the majority of memory references.

Figure 4-4 shows the format of the visible and hidden portions of the segment register. Except for the FS and GS segment base, software cannot directly read or write the hidden portion (shown as gray-shaded boxes in Figure 4-4).

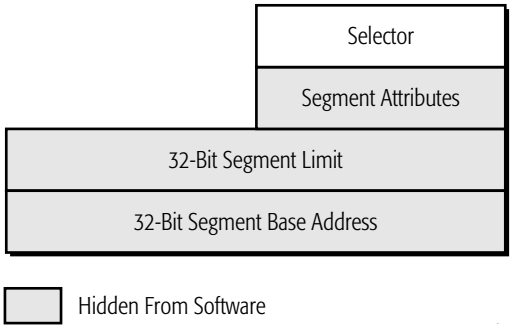


Figure 4-4. Segment-Register Format

CS Register. The CS register contains the segment selector referencing the current code-segment descriptor entry. All instruction fetches reference the CS descriptor. When a new selector is loaded into

the CS register, the current-privilege level (CPL) of the processor is set to that of the CS-segment descriptor-privilege level (DPL).

Data-Segment Registers. The DS register contains the segment selector referencing the default data-segment descriptor entry. The SS register contains the stack-segment selector. The ES, FS, and GS registers are optionally loaded with segment selectors referencing other data segments. Data accesses default to referencing the DS descriptor except in the following two cases:

- The ES descriptor is referenced for string-instruction destinations.
- The SS descriptor is referenced for stack operations.

4.5.3 Segment Registers in 64-Bit Mode

CS Register in 64-Bit Mode. In 64-bit mode, most of the hidden portion of the CS register is ignored. Only the L (long), D (default operation size), and DPL (descriptor privilege-level) attributes are recognized by 64-bit mode. Address calculations assume a CS.base value of 0. CS references do not check the CS.limit value, but instead check that the effective address is in canonical form.

DS, ES, and SS Registers in 64-Bit Mode. In 64-bit mode, the contents of the ES, DS, and SS segment registers are ignored. All fields (base, limit, and attribute) in the hidden portion of the segment registers are ignored.

Address calculations in 64-bit mode that reference the ES, DS, or SS segments are treated as if the segment base is 0. Instead of performing limit checks, the processor checks that all virtual-address references are in canonical form.

Neither enabling and activating long mode nor switching between 64-bit and compatibility modes changes the contents of the visible or hidden portions of the segment registers. These registers remain unchanged during 64-bit mode execution unless explicit segment loads are performed.

FS and GS Registers in 64-Bit Mode. Unlike the CS, DS, ES, and SS segments, the FS and GS segment overrides can be used in 64-bit mode. When FS and GS segment overrides are used in 64-bit mode, their respective base addresses are used in the effective-address (EA) calculation. The complete EA calculation then becomes (FS or GS).base + base + (scale ∗ index) + displacement. The FS.base and GS.base values are also expanded to the full 64-bit virtual-address size, as shown in Figure 4-5. Any overflow in the 64-bit linear address calculation is ignored and the resulting address instead wraps around to the other end of the address space.

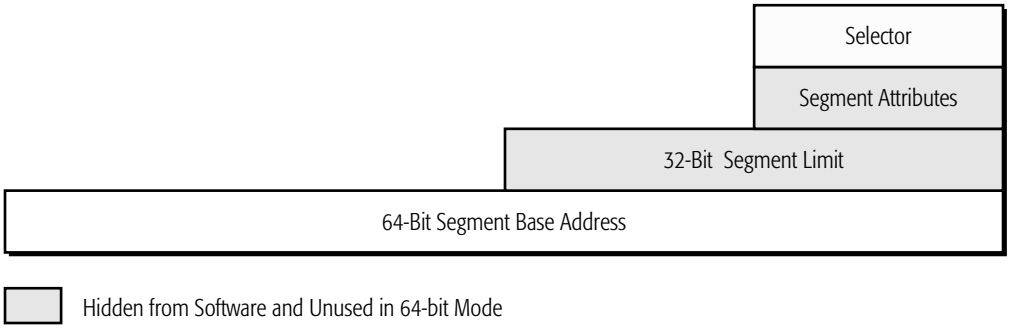


Figure 4-5. FS and GS Segment-Register Format—64-Bit Mode

In 64-bit mode, FS-segment and GS-segment overrides are not checked for limit or attributes. Instead, the processor checks that all virtual-address references are in canonical form.

Segment register-load instructions (MOV to Sreg and POP Sreg) load only a 32-bit base-address value into the hidden portion of the FS and GS segment registers. The base-address bits above the low 32 bits are cleared to 0 as a result of a segment-register load. When a null selector is loaded into FS or GS, the contents of the corresponding hidden descriptor register are not altered.

There are two methods to update the contents of the FS.base and GS.base hidden descriptor fields. The first is available exclusively to privileged software (CPL = 0). The FS.base and GS.base hidden descriptor-register fields are mapped to MSRs. Privileged software can load a 64-bit base address in canonical form into FS.base or GS.base using a single WRMSR instruction. The FS.base MSR address is C000_0100h while the GS.base MSR address is C000_0101h.

The second method of updating the FS and GS base fields is available to software running at any privilege level (when supported by the implementation and enabled by setting CR4[FSGSBASE]). The WRFSBASE and WRGSBASE instructions copy the contents of a GPR to the FS.base and GS.base fields respectively. When the operand size is 32 bits, the upper doubleword of the base is cleared. WRFSBASE and WRGSBASE are only supported in 64-bit mode.

The addresses written into the expanded FS.base and GS.base registers must be in canonical form. Any instruction that attempts to write a non-canonical address to these registers causes a general-protection exception (#GP) to occur.

When in compatibility mode, the FS and GS overrides operate as defined by the legacy x86 architecture regardless of the value loaded into the high 32 bits of the hidden descriptor-register base-address field. Compatibility mode ignores the high 32 bits when calculating an effective address.

4.6 Descriptor Tables

Descriptor tables are used by the segmentation mechanism when protected mode is enabled (CR0.PE=1). These tables hold descriptor entries that describe the location, size, and privilege attributes of a segment. All memory references in protected mode access a descriptor-table entry.

As previously mentioned, there are three types of descriptor tables supported by the x86 segmentation mechanism:

- Global descriptor table (GDT)
- Local descriptor table (LDT)
- Interrupt descriptor table (IDT)

Software establishes the location of a descriptor table in memory by initializing its corresponding descriptor-table register. The descriptor-table registers and the descriptor tables are described in the following sections.

4.6.1 Global Descriptor Table

Protected-mode system software must create a global descriptor table (GDT). The GDT contains code-segment and data-segment descriptor entries (user segments) for segments that can be shared by all tasks. In addition to the user segments, the GDT can also hold gate descriptors and other system-segment descriptors. System software can store the GDT anywhere in memory and should protect the segment containing the GDT from non-privileged software.

Segment selectors point to the GDT when the table-index (TI) bit in the selector is cleared to 0. The selector index portion of the segment selector references a specific entry in the GDT. Figure 4-6 on page 83 shows how the segment selector indexes into the GDT. One special form of a segment selector is the *null selector*. A null selector points to the first entry in the GDT (the selector index is 0 and TI=0). However, null selectors do not reference memory, so the first GDT entry cannot be used to describe a segment (see “Null Selector” on page 78 for information on using the null selector). The first usable GDT entry is referenced with a selector index of 1.

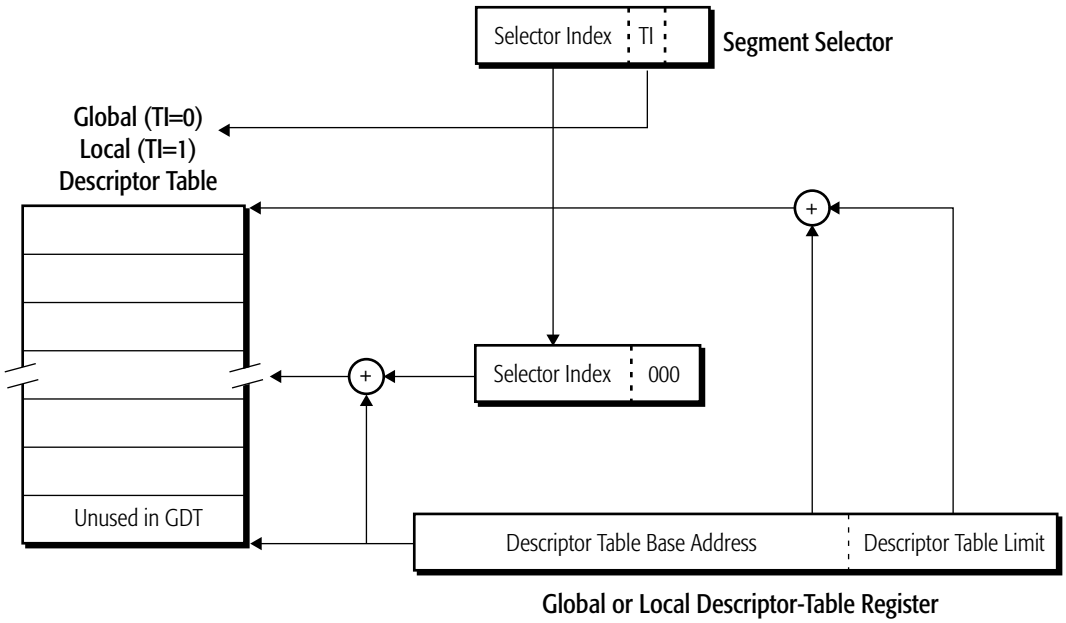


Figure 4-6. Global and Local Descriptor-Table Access

4.6.2 Global Descriptor-Table Register

The global descriptor-table register (GDTR) points to the location of the GDT in memory and defines its size. This register is loaded from memory using the LGDT instruction (see “LGDT and LIDT Instructions” on page 176). Figure 4-7 shows the format of the GDTR in legacy mode and compatibility mode.

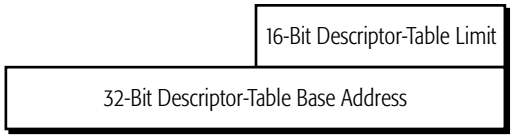


Figure 4-7. GDTR and IDTR Format—Legacy Modes

Figure 4-8 on page 84 shows the format of the GDTR in 64-bit mode.

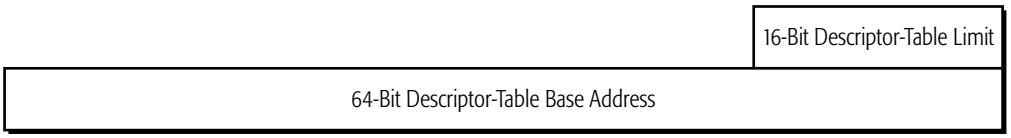


Figure 4-8. GDTR and IDTR Format—Long Mode

The GDTR contains two fields:

Limit. 2 bytes. These bits define the 16-bit limit, or size, of the GDT in bytes. The limit value is added to the base address to yield the ending byte address of the GDT. A general-protection exception (#GP) occurs if software attempts to access a descriptor beyond the GDT limit.

The offsets into the descriptor tables are not extended by the AMD64 architecture in support of long mode. Therefore, the GDTR and IDTR limit-field sizes are unchanged from the legacy sizes. The processor does check the limits in long mode during GDT and IDT accesses.

Base Address. 8 bytes. The base-address field holds the starting byte address of the GDT in virtual-memory space. The GDT can be located at any byte address in virtual memory, but system software should align the GDT on a quadword boundary to avoid the potential performance penalties associated with accessing unaligned data.

The AMD64 architecture increases the base-address field of the GDTR to 64 bits so that system software running in long mode can locate the GDT anywhere in the 64-bit virtual-address space. The processor ignores the high-order 4 bytes of base address when running in legacy mode.

4.6.3 Local Descriptor Table

Protected-mode system software can optionally create a local descriptor table (LDT) to hold segment descriptors belonging to a single task or even multiple tasks. The LDT typically contains code-segment and data-segment descriptors as well as gate descriptors referenced by the specified task. Like the GDT, system software can store the LDT anywhere in memory and should protect the segment containing the LDT from non-privileged software.

Segment selectors point to the LDT when the table-index bit (TI) in the selector is set to 1. The selector index portion of the segment selector references a specific entry in the LDT (see Figure 4-6 on page 83). Unlike the GDT, however, a selector index of 0 references the first entry in the LDT (when TI=1, the selector is not a null selector).

LDTs are described by system-segment descriptor entries located in the GDT, and a GDT can contain multiple LDT descriptors. The LDT system-segment descriptor defines the location, size, and privilege rights for the LDT. Figure 4-9 on page 85 shows the relationship between the LDT and GDT data structures.

Loading a null selector into the LDTR is useful if software does not use an LDT. This causes a #GP if an erroneous reference is made to the LDT.

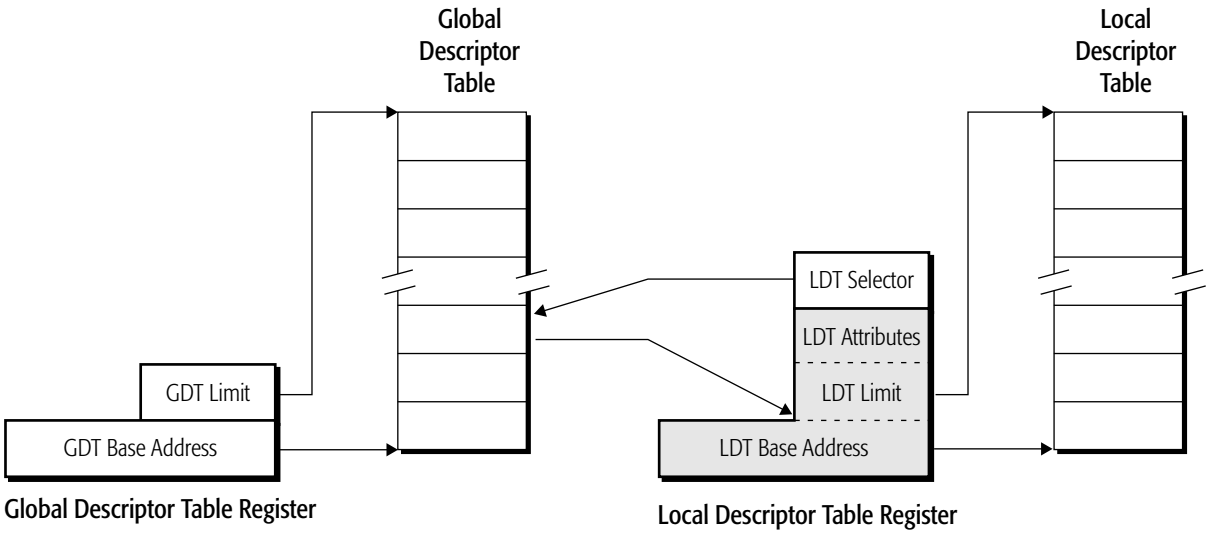


Figure 4-9. Relationship between the LDT and GDT

4.6.4 Local Descriptor-Table Register

The local descriptor-table register (LDTR) points to the location of the LDT in memory, defines its size, and specifies its attributes. The LDTR has two portions. A *visible* portion holds the LDT selector, and a *hidden* portion holds the LDT descriptor. When the LDT selector is loaded into the LDTR, the processor automatically loads the LDT descriptor from the GDT into the hidden portion of the LDTR. The LDTR is loaded in one of two ways:

- Using the LLDT instruction (see “LLDT and LTR Instructions” on page 177).
- Performing a task switch (see “Switching Tasks” on page 375).

Figure 4-10 on page 86 shows the format of the LDTR in legacy mode.

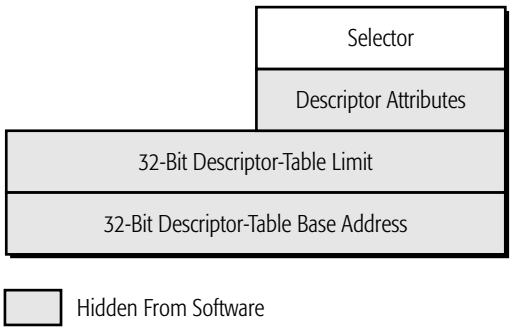


Figure 4-10. LDTR Format—Legacy Mode

Figure 4-11 shows the format of the LDTR in long mode (both compatibility mode and 64-bit mode).

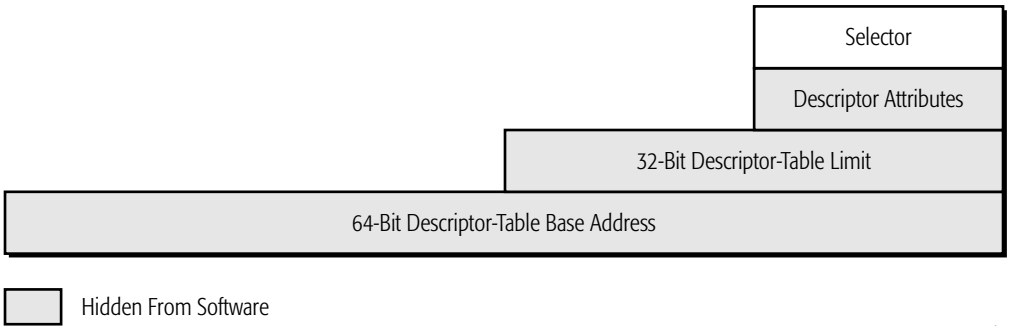


Figure 4-11. LDTR Format—Long Mode

The LDTR contains four fields:

LDT Selector. 2 bytes. These bits are loaded explicitly from the TSS during a task switch, or by using the LLDT instruction. The LDT selector must point to an LDT system-segment descriptor entry in the GDT. If it does not, a general-protection exception (#GP) occurs.

The following three fields are loaded automatically from the LDT descriptor in the GDT as a result of loading the LDT selector. The register fields are shown as shaded boxes in Figure 4-10 and Figure 4-11.

Base Address. The base-address field holds the starting byte address of the LDT in virtual-memory space. Like the GDT, the LDT can be located anywhere in system memory, but software should align the LDT on a quadword boundary to avoid performance penalties associated with accessing unaligned data.

The AMD64 architecture expands the base-address field of the LDTR to 64 bits so that system software running in long mode can locate an LDT anywhere in the 64-bit virtual-address space. The processor ignores the high-order 32 base-address bits when running in legacy mode. Because the LDTR is loaded from the GDT, the system-segment descriptor format (LDTs are system segments) has been expanded by the AMD64 architecture in support of 64-bit mode. See “Long Mode Descriptor Summary” on page 103 for more information on this expanded format. The high-order base-address bits are only loaded from 64-bit mode using the LLDT instruction (see “LLDT and LTR Instructions” on page 177 for more information on this instruction).

Limit. This field defines the limit, or size, of the LDT in bytes. The LDT limit as stored in the LDTR is 32 bits. When the LDT limit is loaded from the GDT descriptor entry, the 20-bit limit field in the descriptor is expanded to 32 bits and scaled based on the value of the descriptor granularity (G) bit. For details on the limit biasing and granularity, see “Granularity (G) Bit” on page 90.

If an attempt is made to access a descriptor beyond the LDT limit, a general-protection exception (#GP) occurs.

The offsets into the descriptor tables are not extended by the AMD64 architecture in support of long mode. Therefore, the LDTR limit-field size is unchanged from the legacy size. The processor does check the LDT limit in long mode during LDT accesses.

Attributes. This field holds the descriptor attributes, such as privilege rights, segment presence and segment granularity.

4.6.5 Interrupt Descriptor Table

The final type of descriptor table is the interrupt descriptor table (IDT). Multiple IDTs can be maintained by system software. System software selects a specific IDT by loading the interrupt descriptor table register (IDTR) with a pointer to the IDT. As with the GDT and LDT, system software can store the IDT anywhere in memory and should protect the segment containing the IDT from non-privileged software.

The IDT can contain only the following types of gate descriptors:

- Interrupt gates
- Trap gates
- Task gates.

The use of gate descriptors by the interrupt mechanism is described in Chapter 8, “Exceptions and Interrupts.” A general-protection exception (#GP) occurs if the IDT descriptor referenced by an interrupt or exception is not one of the types listed above.

IDT entries are selected using the interrupt vector number rather than a selector value. The interrupt vector number is scaled by the interrupt-descriptor entry size to form an offset into the IDT. The interrupt-descriptor entry size depends on the processor operating mode as follows:

- In long mode, interrupt descriptor-table entries are 16 bytes.

- In legacy mode, interrupt descriptor-table entries are eight bytes.

Figure 4-12 shows how the interrupt vector number indexes the IDT.

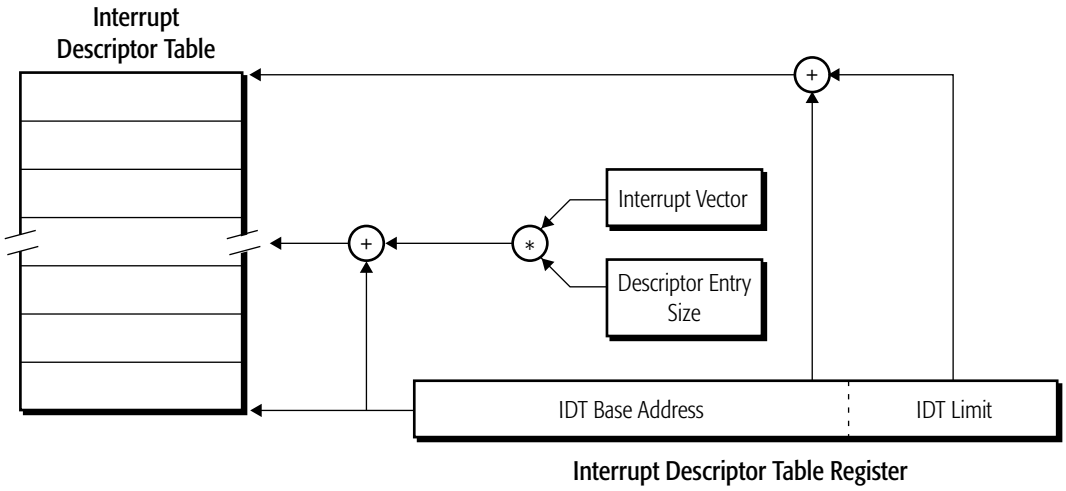


Figure 4-12. Indexing an IDT

4.6.6 Interrupt Descriptor-Table Register

The interrupt descriptor-table register (IDTR) points to the IDT in memory and defines its size. This register is loaded from memory using the LIDT instruction (see “LGDT and LIDT Instructions” on page 176). The format of the IDTR is identical to that of the GDTR in all modes. Figure 4-7 on page 83 shows the format of the IDTR in legacy mode. Figure 4-8 on page 84 shows the format of the IDTR in long mode.

The offsets into the descriptor tables are not extended by the AMD64 architecture in support of long mode. Therefore, the IDTR limit-field size is unchanged from the legacy size. The processor does check the IDT limit in long mode during IDT accesses.

4.7 Legacy Segment Descriptors

4.7.1 Descriptor Format

Segment descriptors define, protect, and isolate segments from each other. There are two basic types of descriptors, each of which are used to describe different segment (or gate) types:

- *User Segments*—These include code segments and data segments. Stack segments are a type of data segment.

- *System Segments*—System segments consist of LDT segments and task-state segments (TSS). Gate descriptors are another type of system-segment descriptor. Rather than describing segments, gate descriptors point to program entry points.

Figure 4-13 shows the generic format for user-segment and system-segment descriptors. User and system segments are differentiated using the S bit. S=1 indicates a user segment, and S=0 indicates a system segment. Gray shading indicates the field or bit is reserved. The format for a gate descriptor differs from the generic segment descriptor, and is described separately in “Gate Descriptors” on page 95.

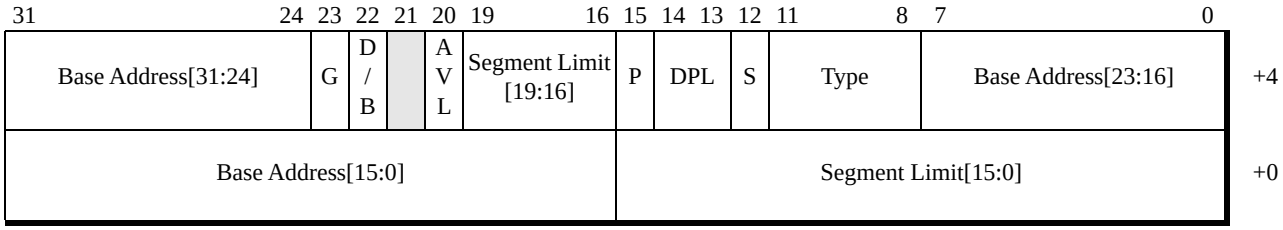


Figure 4-13. Generic Segment Descriptor—Legacy Mode

Figure 4-13 shows the fields in a generic, legacy-mode, 8-byte (two doubleword) segment descriptor. In this figure, the upper doubleword (located at byte offset +4) is shown on top and the lower doubleword (located at byte offset +0) is shown on the bottom. The fields are defined as follows:

Segment Limit. The 20-bit segment limit is formed by concatenating bits 19:16 of the upper doubleword with bits 15:0 of lower doubleword. The segment limit defines the segment size, in bytes. The granularity (G) bit controls how the segment-limit field is scaled (see “Granularity (G) Bit” on page 90). For data segments, the expand-down (E) bit determines whether the segment limit defines the lower or upper segment-boundary (see “Expand-Down (E) Bit” on page 93).

If software references a segment descriptor with an address beyond the segment limit, a general-protection exception (#GP) occurs. The #GP occurs if any part of the memory reference falls outside the segment limit. For example, a doubleword (4-byte) address reference causes a #GP if one or more bytes are located beyond the segment limit.

Base Address. The 32-bit base address is formed by concatenating bits 31:24 of the upper doubleword with bits 7:0 of the same doubleword and bits 15:0 of the lower doubleword. The segment-base address field locates the start of a segment in virtual-address space.

S Bit and Type Field. Bit 12 and bits 11:8 of the upper doubleword. The S and Type fields, together, specify the descriptor type and its access characteristics. Table 4-2 summarizes the descriptor types by S-field encoding and gives a cross reference to descriptions of the Type-field encodings.

Table 4-2. Descriptor Types

S Field	Descriptor Type	Type-Field Encoding
0 (System)	LDT	See Table 4-5 on page 94
	TSS	
	Gate	
1 (User)	Code	See Table 4-3 on page 92
	Data	See Table 4-4 on page 93

Descriptor Privilege-Level (DPL) Field. Bits 14:13 of the upper doubleword. The DPL field indicates the descriptor-privilege level of the segment. DPL can be set to any value from 0 to 3, with 0 specifying the most privilege and 3 the least privilege. See “Data-Access Privilege Checks” on page 106 and “Control-Transfer Privilege Checks” on page 109 for more information on how the DPL is used during segment privilege-checks.

Present (P) Bit. Bit 15 of the upper doubleword. The segment-present bit indicates that the segment referenced by the descriptor is loaded in memory. If a reference is made to a descriptor entry when P = 0, a segment-not-present exception (#NP) occurs. This bit is set and cleared by system software and is never altered by the processor.

Available To Software (AVL) Bit. Bit 20 of the upper doubleword. This field is available to software, which can write any value to it. The processor does not set or clear this field.

Default Operand Size (D/B) Bit. Bit 22 of the upper doubleword. The default operand-size bit is found in code-segment and data-segment descriptors but not in system-segment descriptors. Setting this bit to 1 indicates a 32-bit default operand size, and clearing it to 0 indicates a 16-bit default size. The effect this bit has on a segment depends on the segment-descriptor type. See “Code-Segment Default-Operand Size (D) Bit” on page 92 for a description of the D bit in code-segment descriptors. “Data-Segment Default Operand Size (D/B) Bit” on page 94 describes the D bit in data-segment descriptors, including stack segments, where the bit is referred to as the “B” bit.

Granularity (G) Bit. Bit 23 of the upper doubleword. The granularity bit specifies how the segment-limit field is scaled. Clearing the G bit to 0 indicates that the limit field is not scaled. In this case, the limit equals the number of bytes available in the segment. Setting the G bit to 1 indicates that the limit field is scaled by 4 Kbytes (4096 bytes). Here, the limit field equals the number of 4-Kbyte *blocks* available in the segment.

Setting a limit of 0 indicates a 1-byte segment limit when G = 0. Setting the same limit of 0 when G = 1 indicates a segment limit of 4095.

Reserved Bits. Generally, software should clear all reserved bits to 0, so they can be defined in future revisions to the AMD64 architecture.

4.7.2 Code-Segment Descriptors

Figure 4-14 shows the code-segment descriptor format (gray shading indicates the bit is reserved). All software tasks require that a segment selector, referencing a valid code-segment descriptor, is loaded into the CS register. Code segments establish the processor operating mode and execution privilege-level. The segments generally contain only instructions and are execute-only, or execute and read-only. Software cannot write into a segment whose selector references a code-segment descriptor.

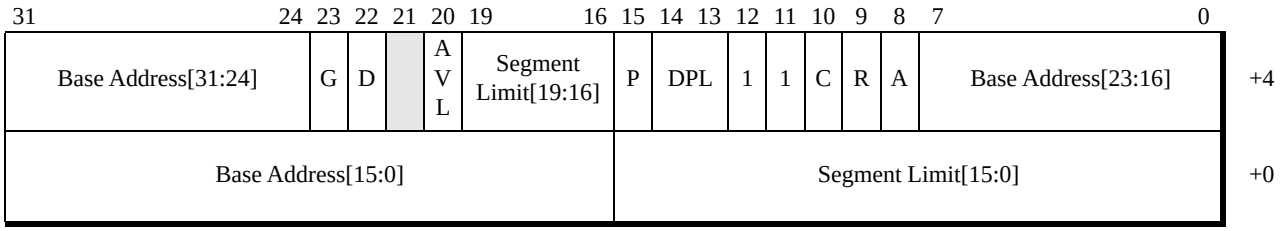


Figure 4-14. Code-Segment Descriptor—Legacy Mode

Code-segment descriptors have the S bit set to 1, identifying the segments as user segments. Type-field bit 11 differentiates code-segment descriptors (bit 11 set to 1) from data-segment descriptors (bit 11 cleared to 0). The remaining type-field bits (10:8) define the access characteristics for the code-segment, as follows:

Conforming (C) Bit. Bit 10 of the upper doubleword. Setting this bit to 1 identifies the code segment as *conforming*. When control is transferred to a higher-privilege conforming code-segment (C=1) from a lower-privilege code segment, the processor CPL does not change. Transfers to non-conforming code-segments (C = 0) with a higher privilege-level than the CPL can occur only through gate descriptors. See “Control-Transfer Privilege Checks” on page 109 for more information on conforming and non-conforming code-segments.

Readable (R) Bit. Bit 9 of the upper doubleword. Setting this bit to 1 indicates the code segment is both executable and readable as data. When this bit is cleared to 0, the code segment is executable, but attempts to read data from the code segment cause a general-protection exception (#GP) to occur.

Accessed (A) Bit. Bit 8 of the upper doubleword. The accessed bit is set to 1 by the processor when the descriptor is copied from the GDT or LDT into the CS register. This bit is only cleared by software.

Table 4-3 on page 92 summarizes the code-segment type-field encodings.

Table 4-3. Code-Segment Descriptor Types

Hex Value	Type Field				Description
	Bit 11 (Code/Data)	Bit 10	Bit 9	Bit 8	
		Conforming (C)	Readable (R)	Accessed (A)	
8	1	0	0	0	Execute-Only
9		0	0	1	Execute-Only — Accessed
A		0	1	0	Execute/Readable
B		0	1	1	Execute/Readable — Accessed
C		1	0	0	Conforming, Execute-Only
D		1	0	1	Conforming, Execute-Only — Accessed
E		1	1	0	Conforming, Execute/Readable
F		1	1	1	Conforming, Execute/Readable — Accessed

Code-Segment Default-Operand Size (D) Bit. Bit 22 of byte +4. In code-segment descriptors, the D bit selects the default operand size and address sizes. In legacy mode, when D=0 the default operand size and address size is 16 bits and when D=1 the default operand size and address size is 32 bits. Instruction prefixes can be used to override the operand size or address size, or both.

4.7.3 Data-Segment Descriptors

Figure 4-15 shows the data-segment descriptor format. Data segments contain non-executable information and can be accessed as read-only or read/write. They are referenced using the DS, ES, FS, GS, or SS data-segment registers. The DS data-segment register holds the segment selector for the default data segment. The ES, FS and GS data-segment registers hold segment selectors for additional data segments usable by the current software task.

The stack segment is a special form of data-segment register. It is referenced using the SS segment register and must be read/write. When loading the SS register, the processor requires that the selector reference a valid, writable data-segment descriptor.

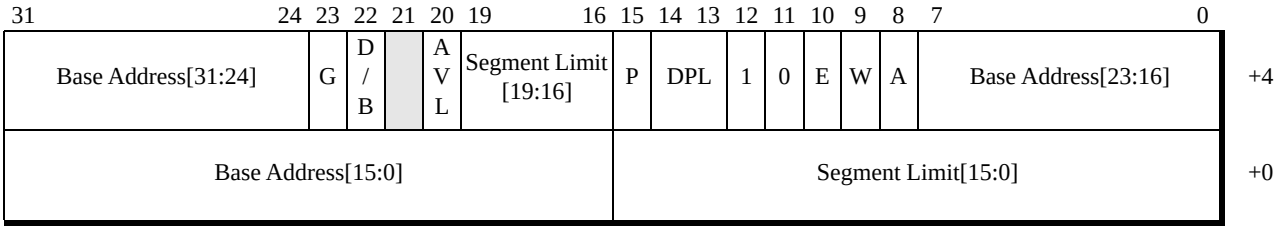


Figure 4-15. Data-Segment Descriptor—Legacy Mode

Data-segment descriptors have the S bit set to 1, identifying them as user segments. Type-field bit 11 differentiates data-segment descriptors (bit 11 cleared to 0) from code-segment descriptors (bit 11 set to 1). The remaining type-field bits (10:8) define the data-segment access characteristics, as follows:

Expand-Down (E) Bit. Bit 10 of the upper doubleword. Setting this bit to 1 identifies the data segment as *expand-down*. In expand-down segments, the segment limit defines the *lower* segment boundary while the base is the upper boundary. Valid segment offsets in expand-down segments lie in the byte range limit+1 to FFFFh or FFFF_FFFFh, depending on the value of the data segment default operand size (D/B) bit.

Expand-down segments are useful for stacks, which grow in the downward direction as elements are pushed onto the stack. The stack pointer, ESP, is *decremented* by an amount equal to the operand size as a result of executing a PUSH instruction.

Clearing the E bit to 0 identifies the data segment as expand-up. Valid segment offsets in expand-up segments lie in the byte range 0 to segment limit.

Writable (W) Bit. Bit 9 of the upper doubleword. Setting this bit to 1 identifies the data segment as read/write. When this bit is cleared to 0, the segment is read-only. A general-protection exception (#GP) occurs if software attempts to write into a data segment when W=0.

Accessed (A) Bit. Bit 8 of the upper doubleword. The accessed bit is set to 1 by the processor when the descriptor is copied from the GDT or LDT into one of the data-segment registers or the stack-segment register. This bit is only cleared by software.

Table 4-4 summarizes the data-segment type-field encodings.

Table 4-4. Data-Segment Descriptor Types

Hex Value	Type Field				Description
	Bit 11 (Code/Data)	Bit 10	Bit 9	Bit 8	
		Expand-Down (E)	Writable (W)	Accessed (A)	
0	0	0	0	0	Read-Only
1		0	0	1	Read-Only — Accessed
2		0	1	0	Read/Write
3		0	1	1	Read/Write — Accessed
4		1	0	0	Expand-down, Read-Only
5		1	0	1	Expand-down, Read-Only — Accessed
6		1	1	0	Expand-down, Read/Write
7		1	1	1	Expand-down, Read/Write — Accessed

Data-Segment Default Operand Size (D/B) Bit. Bit 22 of the upper doubleword. For expand-down data segments (E=1), setting D=1 sets the upper bound of the segment at 0_FFFF_FFFFh. Clearing D=0 sets the upper bound of the segment at 0_FFFFh.

In the case where a data segment is referenced by the stack selector (SS), the D bit is referred to as the B bit. For stack segments, the B bit sets the default stack size. Setting B=1 establishes a 32-bit stack referenced by the 32-bit ESP register. Clearing B=0 establishes a 16-bit stack referenced by the 16-bit SP register.

4.7.4 System Descriptors

There are two general types of system descriptors: system-segment descriptors and gate descriptors. System-segment descriptors are used to describe the LDT and TSS segments. Gate descriptors do not describe segments, but instead hold pointers to code-segment descriptors. Gate descriptors are used for protected-mode control transfers between less-privileged and more-privileged software.

System-segment descriptors have the S bit cleared to 0. The type field is used to differentiate the various LDT, TSS, and gate descriptors from one another. Table 4-5 summarizes the system-segment type-field encodings.

Table 4-5. System-Segment Descriptor Types (S=0)—Legacy Mode

Hex Value	Type Field (Bits 11:8)	Description
0	0000	Reserved (Illegal)
1	0001	Available 16-bit TSS
2	0010	LDT
3	0011	Busy 16-bit TSS
4	0100	16-bit Call Gate
5	0101	Task Gate
6	0110	16-bit Interrupt Gate
7	0111	16-bit Trap Gate
8	1000	Reserved (Illegal)
9	1001	Available 32-bit TSS
A	1010	Reserved (Illegal)
B	1011	Busy 32-bit TSS
C	1100	32-bit Call Gate
D	1101	Reserved (Illegal)
E	1110	32-bit Interrupt Gate
F	1111	32-bit Trap Gate

Figure 4-16 shows the legacy-mode system-segment descriptor format used for referencing LDT and TSS segments (gray shading indicates the bit is reserved). This format is also used in compatibility mode. The system-segments are used as follows:

- The LDT typically holds segment descriptors belonging to a single task (see “Local Descriptor Table” on page 84).
- The TSS is a data structure for holding processor-state information. Processor state is saved in a TSS when a task is suspended, and state is restored from the TSS when a task is restarted. System software must create at least one TSS referenced by the task register, TR. See “Legacy Task-State Segment” on page 367 for more information on the TSS.

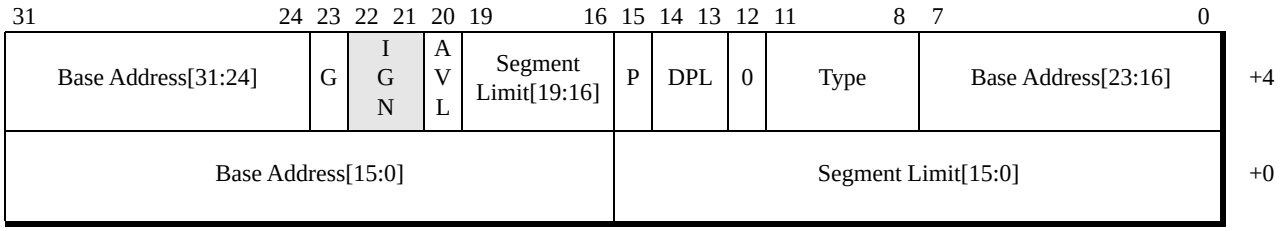


Figure 4-16. LDT and TSS Descriptor—Legacy/Compatibility Modes

4.7.5 Gate Descriptors

Gate descriptors hold pointers to code segments and are used to control access between code segments with different privilege levels. There are four types of gate descriptors:

- *Call Gates*—These gates (Figure 4-17 on page 96) are located in the GDT or LDT and are used to control access between code segments in the same task or in different tasks. See “Control Transfers Through Call Gates” on page 113 for information on how call gates are used to control access between code segments operating in the same task. The format of a call-gate descriptor is shown in Figure 4-17 on page 96.
- *Interrupt Gates* and *Trap Gates*—These gates (Figure 4-18 on page 96) are located in the IDT and are used to control access to interrupt-service routines. “Legacy Protected-Mode Interrupt Control Transfers” on page 261 contains information on using these gates for interrupt-control transfers. The format of interrupt-gate and trap-gate descriptors is shown in Figure 4-17 on page 96.
- *Task Gates*—These gates (Figure 4-19 on page 96) are used to control access between different tasks. They are also used to transfer control to interrupt-service routines if those routines are themselves a separate task. See “Task-Management Resources” on page 362 for more information on task gates and their use.

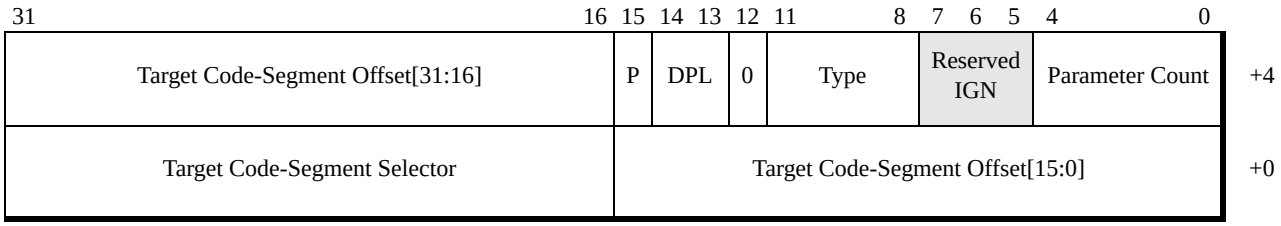


Figure 4-17. Call-Gate Descriptor—Legacy Mode

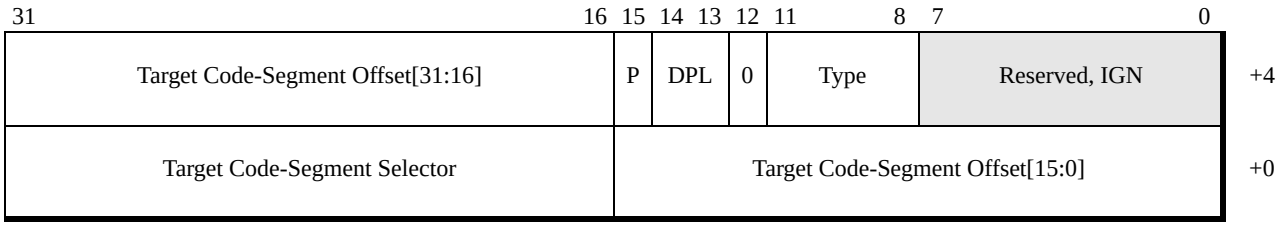


Figure 4-18. Interrupt-Gate and Trap-Gate Descriptors—Legacy Mode

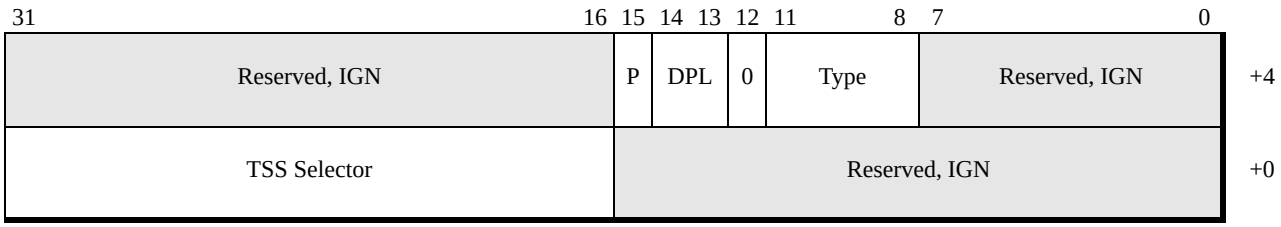


Figure 4-19. Task-Gate Descriptor—Legacy Mode

There are several differences between the gate-descriptor format and the system-segment descriptor format. These differences are described as follows, from least-significant to most-significant bit positions:

Target Code-Segment Offset. The 32-bit segment offset is formed by concatenating bits 31:16 of byte +4 with bits 15:0 of byte +0. The segment-offset field specifies the target-procedure entry point (offset) into the segment. This field is loaded into the EIP register as a result of a control transfer using the gate descriptor.

Target Code-Segment Selector. Bits 31:16 of byte +0. The segment-selector field identifies the target-procedure segment descriptor, located in either the GDT or LDT. The segment selector is loaded into the CS segment register as a result of a control transfer using the gate descriptor.

TSS Selector. Bits 31:16 of byte +0 (task gates only). This field identifies the target-task TSS descriptor, located in any of the three descriptor tables (GDT, LDT, and IDT).

Parameter Count (Call Gates Only). Bits 4:0 of byte +4. Legacy-mode call-gate descriptors contain a 5-bit *parameter-count* field. This field specifies the number of parameters to be copied from the currently-executing program stack to the target program stack during an automatic stack switch. Automatic stack switches are performed by the processor during a control transfer through a call gate to a greater privilege-level. The parameter size depends on the call-gate size as specified in the type field. 32-bit call gates copy 4-byte parameters, and 16-bit call gates copy 2-byte parameters. See “Stack Switching” on page 117 for more information on call-gate parameter copying.

4.8 Long-Mode Segment Descriptors

The interpretation of descriptor fields is changed in long mode, and in some cases the format is expanded. The changes depend on the operating mode (compatibility mode or 64-bit mode) and on the descriptor type. The following sections describe the changes.

4.8.1 Code-Segment Descriptors

Code segments continue to exist in long mode. Code segments and their associated descriptors and selectors are needed to establish the processor operating mode as well as execution privilege-level. The new L attribute specifies whether the processor is running in compatibility mode or 64-bit mode (see “Long (L) Attribute Bit” on page 98). Figure 4-20 shows the long-mode code-segment descriptor format. In compatibility mode, the code-segment descriptor is interpreted and behaves just as it does in legacy mode as described in “Code-Segment Descriptors” on page 91.

In Figure 4-20, gray shading indicates the code-segment descriptor fields that are *ignored in 64-bit mode* when the descriptor is used during a memory reference. However, the fields are loaded whenever the segment register is loaded in 64-bit mode.

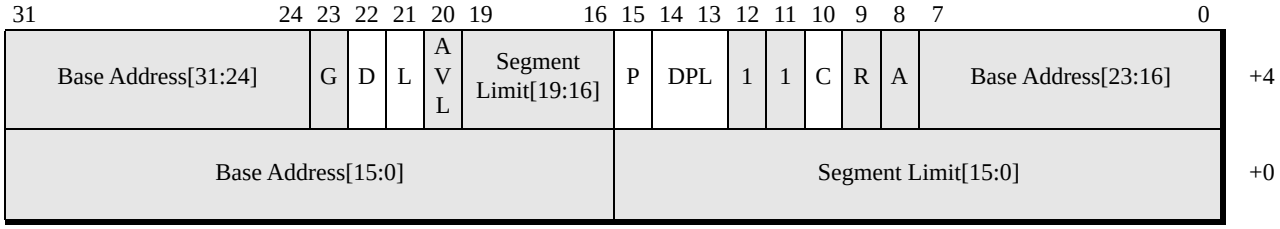


Figure 4-20. Code-Segment Descriptor—Long Mode

Fields Ignored in 64-Bit Mode. Segmentation is disabled in 64-bit mode, and code segments span all of virtual memory. In this mode, code-segment base addresses are ignored. For the purpose of virtual-address calculations, the base address is treated as if it has a value of zero.

Segment-limit checking is not performed, and both the segment-limit field and granularity (G) bit are ignored. Instead, the virtual address is checked to see if it is in canonical-address form.

The readable (R) and accessed (A) attributes in the type field are also ignored.

Long (L) Attribute Bit. Bit 21 of byte +4. Long mode introduces a new attribute, the *long* (L) bit, in code-segment descriptors. This bit specifies that the processor is running in 64-bit mode (L=1) or compatibility mode (L=0). When the processor is running in legacy mode, this bit is reserved.

Compatibility mode maintains binary compatibility with legacy 16-bit and 32-bit applications. Compatibility mode is selected on a code-segment basis, and it allows legacy applications to coexist under the same 64-bit system software along with 64-bit applications running in 64-bit mode. System software running in long mode can execute existing 16-bit and 32-bit applications by clearing the L bit of the code-segment descriptor to 0.

When L=0, the legacy meaning of the code-segment D bit (see “Code-Segment Default-Operand Size (D) Bit” on page 92)—and the address-size and operand-size prefixes—are observed. Segmentation is enabled when L=0. From an application viewpoint, the processor is in a legacy 16-bit or 32-bit operating environment (depending on the D bit), even though long mode is activated.

If the processor is running in 64-bit mode (L=1), the only valid setting of the D bit is 0. This setting produces a default operand size of 32 bits and a default address size of 64 bits. The combination L=1 and D=1 is reserved for future use.

“Instruction Prefixes” in Volume 3 describes the effect of the code-segment L and D bits on default operand and address sizes when long mode is activated. These default sizes can be overridden with operand size, address size, and REX prefixes.

4.8.2 Data-Segment Descriptors

Data segments continue to exist in long mode. Figure 4-21 shows the long-mode data-segment descriptor format. In compatibility mode, data-segment descriptors are interpreted and behave just as they do in legacy mode.

In Figure 4-21, gray shading indicates the fields that are *ignored in 64-bit mode* when the descriptor is used during a memory reference. However, the fields are loaded whenever the segment register is loaded in 64-bit mode.

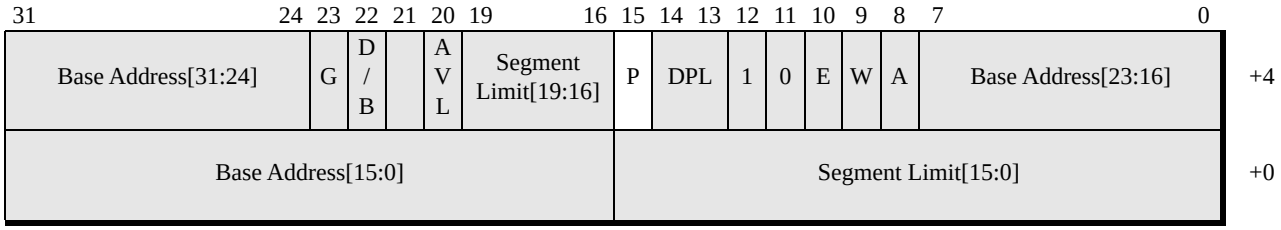


Figure 4-21. Data-Segment Descriptor—Long Mode

Fields Ignored in 64-Bit Mode. Segmentation is disabled in 64-bit mode. The interpretation of the segment-base address depends on the segment register used:

- In data-segment descriptors referenced by the DS, ES and SS segment registers, the base-address field is ignored. For the purpose of virtual-address calculations, the base address is treated as if it has a value of zero.
- Data segments referenced by the FS and GS segment registers receive special treatment in 64-bit mode. For these segments, the base address field is not ignored, and a non-zero value can be used in virtual-address calculations. A 64-bit segment-base address can be specified using model-specific registers. See “FS and GS Registers in 64-Bit Mode” on page 80 for more information.

Segment-limit checking is not performed on any data segments in 64-bit mode, and both the segment-limit field and granularity (G) bit are ignored. The D/B bit is unused in 64-bit mode.

The expand-down (E), writable (W), and accessed (A) type-field attributes are ignored.

A data-segment-descriptor DPL field is ignored in 64-bit mode, and segment-privilege checks are not performed on data segments. System software can use the page-protection mechanisms to isolate and protect data from unauthorized access.

4.8.3 System Descriptors

In long mode, the allowable system-descriptor types encoded by the type field are changed. Some descriptor types are modified, and others are illegal. The changes are summarized in Table 4-6. An attempt to use an illegal descriptor type causes a general-protection exception (#GP).

Table 4-6. System-Segment Descriptor Types—Long Mode

Hex Value	Type Field				Description
	Bit 11	Bit 10	Bit 9	Bit 8	
0	0	0	0	0	Reserved (Illegal)
1	0	0	0	1	
2	0	0	1	0	64-bit LDT[1]
3	0	0	1	1	Reserved (Illegal)
4	0	1	0	0	
5	0	1	0	1	
6	0	1	1	0	
7	0	1	1	1	
8	1	0	0	0	
9	1	0	0	1	Available 64-bit TSS
A	1	0	1	0	Reserved (Illegal)
B	1	0	1	1	Busy 64-bit TSS
C	1	1	0	0	64-bit Call Gate

Note(s):
1. In 64-bit mode only. In compatibility mode, the type specifies a 32-bit LDT.

Table 4-6. System-Segment Descriptor Types—Long Mode (continued)

Hex Value	Type Field				Description
	Bit 11	Bit 10	Bit 9	Bit 8	
D	1	1	0	1	Reserved (Illegal)
E	1	1	1	0	64-bit Interrupt Gate
F	1	1	1	1	64-bit Trap Gate

Note(s):
1. In 64-bit mode only. In compatibility mode, the type specifies a 32-bit LDT.

In long mode, the modified system-segment descriptor types are:

- The 32-bit LDT (02h), which is redefined as the 64-bit LDT.
- The available 32-bit TSS (09h), which is redefined as the available 64-bit TSS.
- The busy 32-bit TSS (0Bh), which is redefined as the busy 64-bit TSS.

In 64-bit mode, the LDT and TSS system-segment descriptors are expanded by 64 bits, as shown in Figure 4-22. In this figure, gray shading indicates the fields that are *ignored in 64-bit mode*. Expanding the descriptors allows them to hold 64-bit base addresses, so their segments can be located anywhere in the virtual-address space. The expanded descriptor can be loaded into the corresponding descriptor-table register (LDTR or TR) only from 64-bit mode. In compatibility mode, the legacy system-segment descriptor format, shown in Figure 4-16 on page 95, is used. See “LLDT and LTR Instructions” on page 177 for more information.

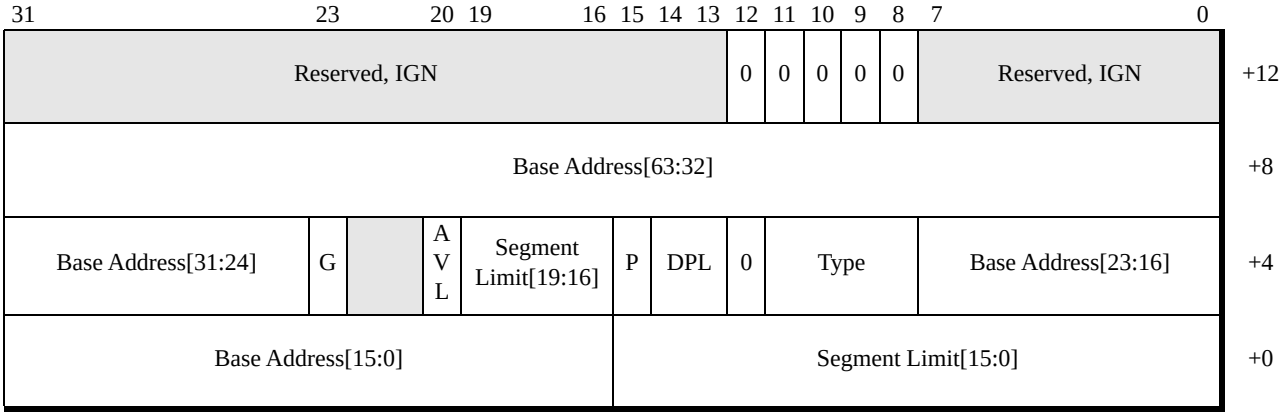


Figure 4-22. System-Segment Descriptor—64-Bit Mode

The 64-bit system-segment base address must be in canonical form. Otherwise, a general-protection exception occurs with a selector error-code, #GP(selector), when the system segment is loaded. System-segment limit values are checked by the processor in both 64-bit and compatibility modes, under the control of the granularity (G) bit.

Figure 4-22 shows that bits 12:8 of doubleword +12 must be cleared to 0. These bits correspond to the S and Type fields in a legacy descriptor. Clearing these bits to 0 corresponds to an illegal type in legacy

mode and causes a #GP if an attempt is made to access the upper half of a 64-bit mode system-segment descriptor as a legacy descriptor or as the lower half of a 64-bit mode system-segment descriptor.

4.8.4 Gate Descriptors

As shown in Table 4-6 on page 99, the allowable gate-descriptor types are changed in long mode. Some gate-descriptor types are modified and others are illegal. The modified gate-descriptor types in long mode are:

- The 32-bit call gate (0Ch), which is redefined as the 64-bit call gate.
- The 32-bit interrupt gate (0Eh), which is redefined as the 64-bit interrupt gate.
- The 32-bit trap gate (0Fh), which is redefined as the 64-bit trap gate.

In long mode, several gate-descriptor types are illegal. An attempt to use these gates causes a general-protection exception (#GP) to occur. The illegal gate types are:

- The 16-bit call gate (04h).
- The task gate (05h).
- The 16-bit interrupt gate (06h).
- The 16-bit trap gate (07h).

In long mode, gate descriptors are expanded by 64 bits, allowing them to hold 64-bit offsets. The 64-bit call-gate descriptor is shown in Figure 4-23 and the 64-bit interrupt gate and trap gate are shown in Figure 4-24 on page 102. In these figures, gray shading indicates the fields that are *ignored in long mode*. The interrupt and trap gates contain an additional field, the IST, that is not present in the call gate—see “IST Field (Interrupt and Trap Gates)” on page 102.

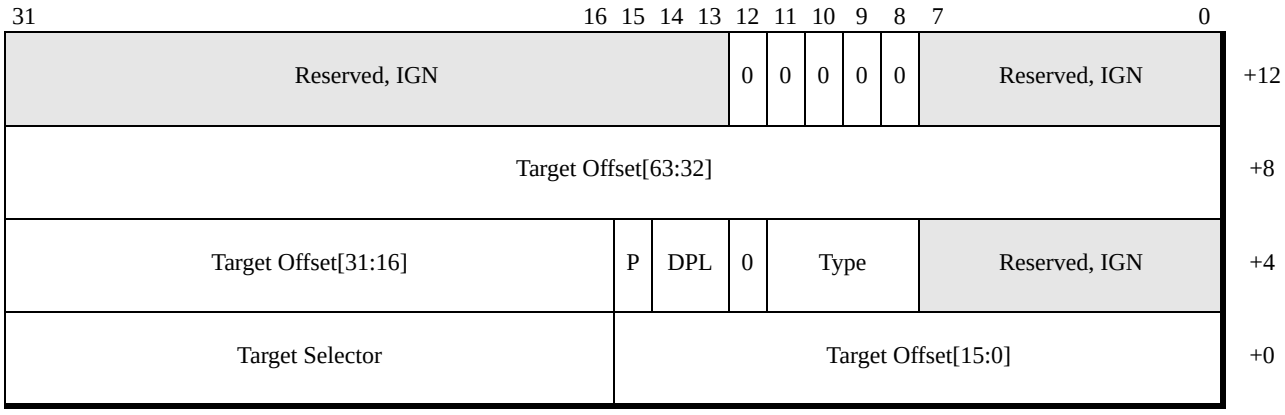


Figure 4-23. Call-Gate Descriptor—Long Mode

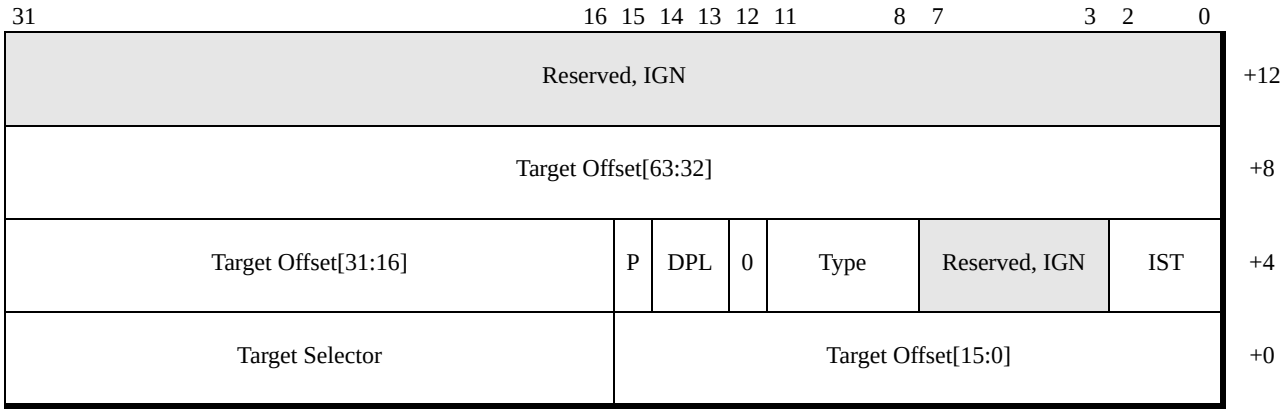


Figure 4-24. Interrupt-Gate and Trap-Gate Descriptors—Long Mode

The target code segment referenced by a long-mode gate descriptor must be a 64-bit code segment (CS.L=1, CS.D=0). If the target is not a 64-bit code segment, a general-protection exception, #GP(error), occurs. The error code reported depends on the gate type:

- Call gates report the target code-segment selector as the error code.
- Interrupt and trap gates report the interrupt vector number as the error code.

A general-protection exception, #GP(0), occurs if software attempts to reference a long-mode gate descriptor with a target-segment offset that is not in canonical form.

It is possible for software to store legacy and long mode gate descriptors in the same descriptor table. Figure 4-23 on page 101 shows that bits 12:8 of byte +12 in a long-mode call gate must be cleared to 0. These bits correspond to the S and Type fields in a legacy call gate. Clearing these bits to 0 corresponds to an illegal type in legacy mode and causes a #GP if an attempt is made to access the upper half of a 64-bit mode call-gate descriptor as a legacy call-gate descriptor.

It is not necessary to clear these same bits in a long-mode interrupt gate or trap gate. In long mode, the interrupt-descriptor table (IDT) must contain 64-bit interrupt gates or trap gates. The processor automatically indexes the IDT by scaling the interrupt vector by 16. This makes it impossible to access the upper half of a long-mode interrupt gate, or trap gate, as a legacy gate when the processor is running in long mode.

IST Field (Interrupt and Trap Gates). Bits 2:0 of byte +4. Long-mode interrupt gate and trap gate descriptors contain a new, 3-bit interrupt-stack-table (IST) field not present in legacy gate descriptors. The IST field is used as an index into the IST portion of a long-mode TSS. If the IST field is not 0, the index references an IST pointer in the TSS, which the processor loads into the RSP register when an interrupt occurs. If the IST index is 0, the processor uses the legacy stack-switching mechanism (with some modifications) when an interrupt occurs. See “Interrupt-Stack Table” on page 276 for more information.

Count Field (Call Gates). The count field found in legacy call-gate descriptors is not supported in long-mode call gates. In long mode, the field is reserved and should be cleared to zero.

4.8.5 Long Mode Descriptor Summary

System descriptors and gate descriptors are expanded by 64 bits to handle 64-bit base addresses in long mode or 64-bit mode. The mode in which the expansion occurs depends on the purpose served by the descriptor, as follows:

- *Expansion Only In 64-Bit Mode*—The system descriptors and pseudo-descriptors that are loaded into the GDTR, IDTR, LDTR, and TR registers are expanded only in 64-bit mode. They are not expanded in compatibility mode.
- *Expansion In Long Mode*—Gate descriptors (call gates, interrupt gates, and trap gates) are expanded in long mode (both 64-bit mode and compatibility mode). Task gates and 16-bit gate descriptors are illegal in long mode.

The AMD64 architecture redefines several of the descriptor-entry fields in support of long mode. The specific change depends on whether the processor is in 64-bit mode or compatibility mode. Table 4-7 summarizes the changes in the descriptor entry field when the descriptor entry is loaded into a segment register (as opposed to when the segment register is subsequently used to access memory).

Table 4-7. Descriptor-Entry Field Changes in Long Mode

<table>
<tr><th rowspan="2">Descriptor Field</th><th rowspan="2">Descriptor Type</th><th colspan="2">Long Mode</th></tr>
<tr><th>Compatibility Mode</th><th>64-Bit Mode</th></tr>
<tr><td rowspan="3">Limit</td><td>Code</td><td rowspan="3">Same as legacy x86</td><td rowspan="3">Same as legacy x86</td></tr>
<tr><td>Data</td></tr>
<tr><td>System</td></tr>
<tr><td>Offset</td><td>Gate</td><td>Expanded to 64 bits</td><td>Expanded to 64 bits</td></tr>
<tr><td rowspan="3">Base</td><td>Code</td><td rowspan="3">Same as legacy x86</td><td rowspan="4">Same as legacy x86</td></tr>
<tr><td>Data</td></tr>
<tr><td>System</td></tr>
<tr><td>Selector</td><td>Gate</td><td>Same as legacy x86</td></tr>
<tr><td>IST[1]</td><td>Gate</td><td colspan="2">Interrupt and trap gates only. (New for long mode.)</td></tr>
<tr><td rowspan="4">S and Type</td><td>Code</td><td rowspan="2">Same as legacy x86</td><td rowspan="2">Same as legacy x86</td></tr>
<tr><td>Data</td></tr>
<tr><td>System</td><td colspan="2">Types 02h, 09h, and 0Bh redefined
Types 01h and 03h are illegal</td></tr>
<tr><td>Gate</td><td colspan="2">Types 0Ch, 0Eh, and 0Fh redefined
Types 04h–07h are illegal</td></tr>
<tr><td colspan="4">Note(s):
1. Not available (reserved) in legacy mode.</td></tr>
</table>

Table 4-7. Descriptor-Entry Field Changes in Long Mode (continued)

Descriptor Field	Descriptor Type	Long Mode	
		Compatibility Mode	64-Bit Mode
DPL	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		
	Gate		
Present	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		
	Gate		
Default Size	Code	Same as legacy x86	D=0 Indicates 64-bit address, 32-bit data D=1 Reserved
	Data		Same as legacy x86
Long[1]	Code	Specifies compatibility mode	Specifies 64-bit mode
Granularity	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		
Available	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		

Note(s):
1. Not available (reserved) in legacy mode.

4.9 Segment-Protection Overview

The AMD64 architecture is designed to fully support the legacy segment-protection mechanism. The segment-protection mechanism provides system software with the ability to restrict program access into other software routines and data.

Segment-level protection remains enabled in compatibility mode. 64-bit mode eliminates most type checking, and limit checking is not performed, except on accesses to system-descriptor tables.

The preferred method of implementing memory protection in a long-mode operating system is to rely on the page-protection mechanism as described in “Page-Protection Checks” on page 158. System software still needs to create basic segment-protection data structures for 64-bit mode. These structures are simplified, however, by the use of the flat-memory model in 64-bit mode, and the limited segmentation checks performed when executing in 64-bit mode.

4.9.1 Privilege-Level Concept

Segment protection is used to isolate and protect programs and data from each other. The segment-protection mechanism supports four privilege levels in protected mode. The privilege levels are designated with a numerical value from 0 to 3, with 0 being the most privileged and 3 being the least privileged. System software typically assigns the privilege levels in the following manner:

- *Privilege-level 0 (most privilege)*—This level is used by critical system-software components that require direct access to, and control over, all processor and system resources. This can include platform firmware, memory-management functions, and interrupt handlers.
- *Privilege-levels 1 and 2 (moderate privilege)*—These levels are used by less-critical system-software services that can access and control a limited scope of processor and system resources. Software running at these privilege levels might include some device drivers and library routines. These software routines can call more-privileged system-software services to perform functions such as memory garbage-collection and file allocation.
- *Privilege-level 3 (least privilege)*—This level is used by application software. Software running at privilege-level 3 is normally prevented from directly accessing most processor and system resources. Instead, applications request access to the protected processor and system resources by calling more-privileged service routines to perform the accesses.

Figure 4-25 shows the relationship of the four privilege levels to each other.

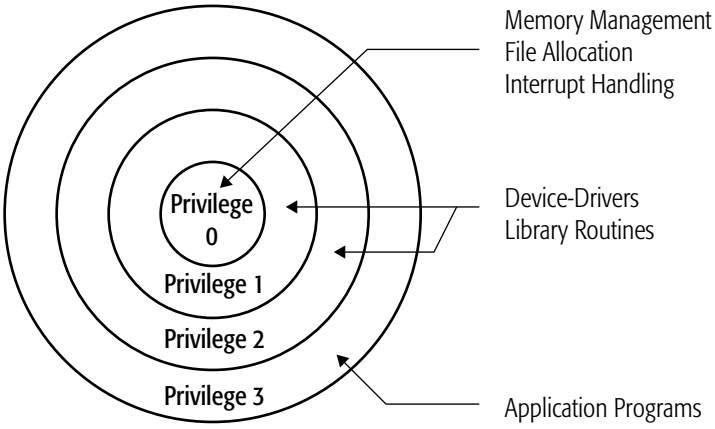


Figure 4-25. Privilege-Level Relationships

4.9.2 Privilege-Level Types

There are three types of privilege levels the processor uses to control access to segments. These are CPL, DPL, and RPL.

Current Privilege-Level. The current privilege-level (CPL) is the privilege level at which the processor is currently executing. The CPL is stored in an internal processor register that is invisible to

software. Software changes the CPL by performing a control transfer to a different code segment with a new privilege level.

Descriptor Privilege-Level. The descriptor privilege-level (DPL) is the privilege level that system software assigns to individual segments. The DPL is used in privilege checks to determine whether software can access the segment referenced by the descriptor. In the case of gate descriptors, the DPL determines whether software can access the descriptor reference by the gate. The DPL is stored in the segment (or gate) descriptor.

Requestor Privilege-Level. The requestor privilege-level (RPL) reflects the privilege level of the program that created the selector. The RPL can be used to let a called program know the privilege level of the program that initiated the call. The RPL is stored in the selector used to reference the segment (or gate) descriptor.

The following sections describe how the CPL, DPL, and RPL are used by the processor in performing privilege checks on data accesses and control transfers. Failure to pass a protection check generally causes an exception to occur.

4.10 Data-Access Privilege Checks

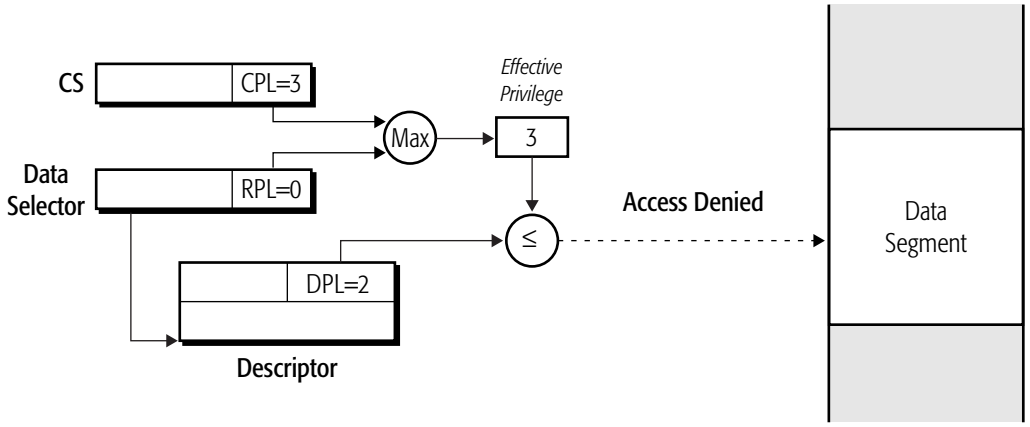
4.10.1 Accessing Data Segments

Before loading a data-segment register (DS, ES, FS, or GS) with a segment selector, the processor checks the privilege levels as follows to see if access is allowed:

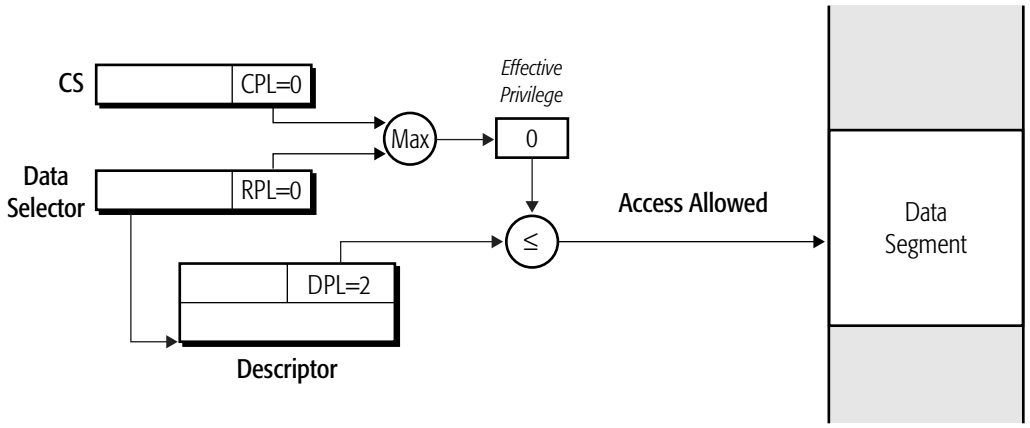
1. The processor compares the CPL with the RPL in the data-segment selector and determines the effective privilege level for the data access. The processor sets the effective privilege level to the lowest privilege (numerically-higher value) indicated by the comparison.
2. The processor compares the effective privilege level with the DPL in the descriptor-table entry referenced by the segment selector. If the effective privilege level is greater than or equal to (numerically lower-than or equal-to) the DPL, then the processor loads the segment register with the data-segment selector. The processor automatically loads the corresponding descriptor-table entry into the hidden portion of the segment register.

   If the effective privilege level is lower than (numerically greater-than) the DPL, a general-protection exception (#GP) occurs and the segment register is not loaded.

Figure 4-26 on page 107 shows two examples of data-access privilege checks.



Example 1: Privilege Check Fails



Example 2: Privilege Check Passes

Figure 4-26. Data-Access Privilege-Check Examples

Example 1 in Figure 4-26 shows a failing data-access privilege check. The effective privilege level is 3 because CPL=3. This value is greater than the descriptor DPL, so access to the data segment is denied.

Example 2 in Figure 4-26 shows a passing data-access privilege check. Here, the effective privilege level is 0 because both the CPL and RPL have values of 0. This value is less than the descriptor DPL, so access to the data segment is allowed, and the data-segment register is successfully loaded.

4.10.2 Accessing Stack Segments

Before loading the stack segment register (SS) with a segment selector, the processor checks the privilege levels as follows to see if access is allowed:

1. The processor checks that the CPL and the stack-selector RPL are *equal*. If they are not equal, a general-protection exception (#GP) occurs and the SS register is not loaded.
2. The processor compares the CPL with the DPL in the descriptor-table entry referenced by the segment selector. The two values *must be equal*. If they are not equal, a #GP occurs and the SS register is not loaded.

Figure 4-27 shows two examples of stack-access privilege checks. In Example 1 the CPL, stack-selector RPL, and stack segment-descriptor DPL are all equal, so access to the stack segment using the SS register is allowed. In Example 2, the stack-selector RPL and stack segment-descriptor DPL are both equal. However, the CPL is not equal to the stack segment-descriptor DPL, and access to the stack segment through the SS register is denied.

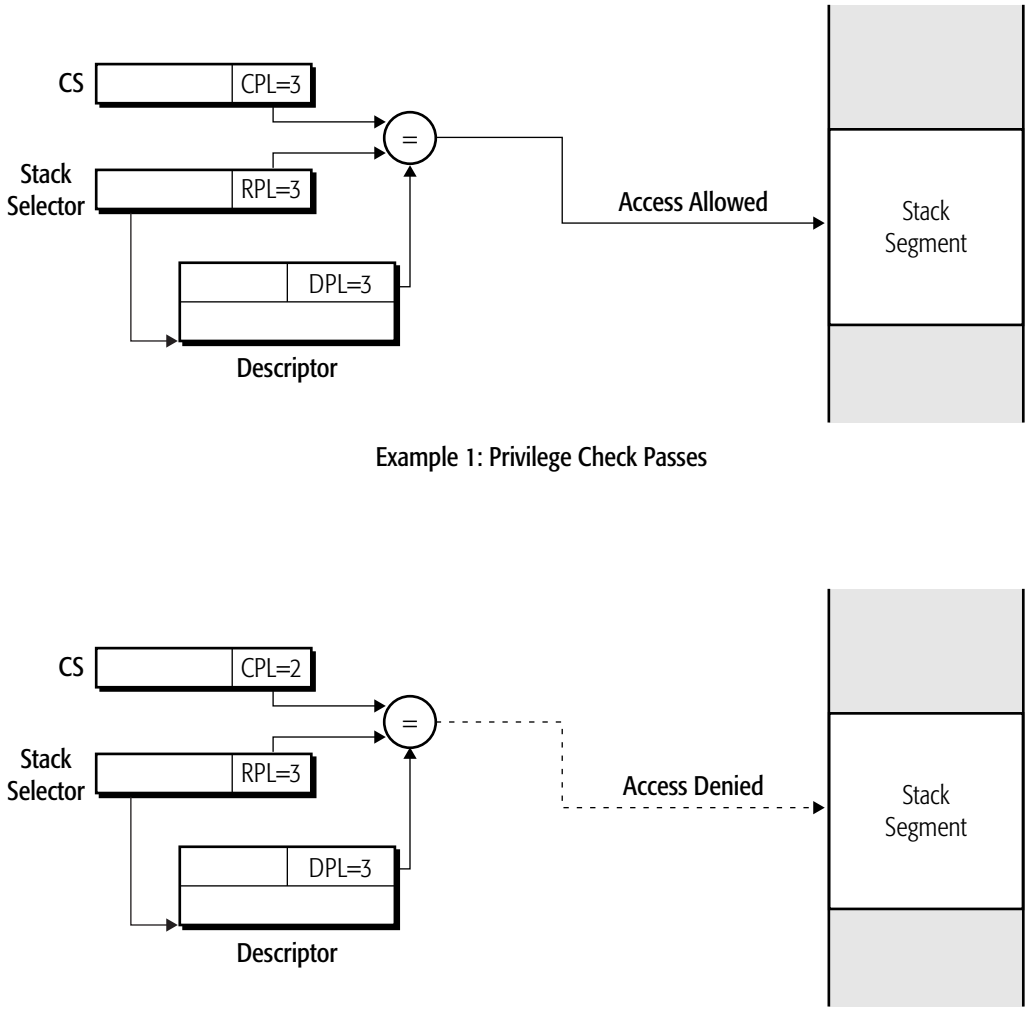


Figure 4-27. Stack-Access Privilege-Check Examples

4.11 Control-Transfer Privilege Checks

Control transfers between code segments (also called *far control transfers*) cause the processor to perform privilege checks to determine whether the source program is allowed to transfer control to the target program. If the privilege checks pass, access to the target code-segment is granted. When access is granted, the target code-segment selector is loaded into the CS register. The rIP register is updated with the target CS offset taken from either the far-pointer operand or the gate descriptor. Privilege checks are not performed during *near control transfers* because such transfers do not change segments.

The following mechanisms can be used by software to perform far control transfers:

- System-software control transfers using the *system-call* and *system-return* instructions. See "SYSCALL and SYSRET" on page 171 and "SYSENTER and SYSEXIT (Legacy Mode Only)" on page 173 for more information on these instructions. SYSCALL and SYSRET are the preferred method of performing control transfers in long mode. *SYSENTER and SYSEXIT are not supported in long mode.*
- Direct control transfers using CALL and JMP instructions. These are discussed in the next section, "Direct Control Transfers."
- Call-gate control transfers using CALL and JMP instructions. These are discussed in "Control Transfers Through Call Gates" on page 113.
- Return control transfers using the RET instruction. These are discussed in "Return Control Transfers" on page 120.
- Interrupts and exceptions, including the INT*n* and IRET instructions. These are discussed in Chapter 8, "Exceptions and Interrupts."
- Task switches initiated by CALL and JMP instructions. Task switches are discussed in Chapter 12, "Task Management." *The hardware task-switch mechanism is not supported in long mode.*

4.11.1 Direct Control Transfers

A *direct control transfer* occurs when software executes a far-CALL or a far-JMP instruction without using a call gate. The privilege checks and type of access allowed as a result of a direct control transfer depends on whether the target code segment is conforming or nonconforming. The code-segment-descriptor conforming (C) bit indicates whether or not the target code-segment is conforming (see "Conforming (C) Bit" on page 91 for more information on the conforming bit).

Privilege levels are not changed as a result of a direct control transfer. Program stacks are not automatically switched by the processor as they are with privilege-changing control transfers through call gates (see "Stack Switching" on page 117 for more information on automatic stack switching during privilege-changing control transfers).

Nonconforming Code Segments. Software can perform a direct control transfer to a nonconforming code segment only if the target code-segment descriptor DPL and the CPL are equal and the RPL is less than or equal to the CPL. Software must use a call gate to transfer control to a

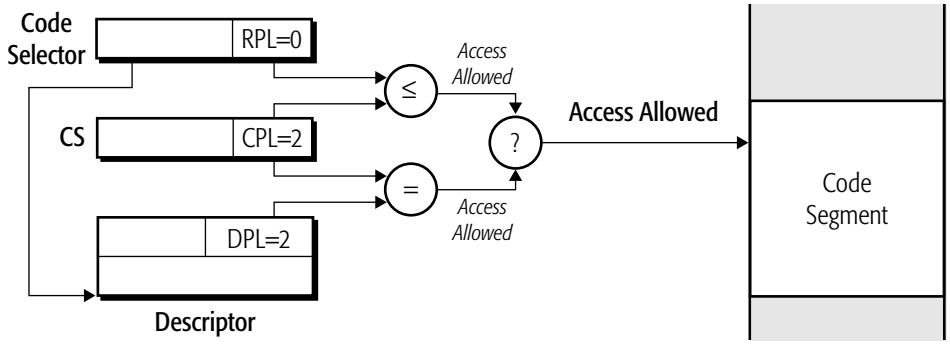
more-privileged, nonconforming code segment (see “Control Transfers Through Call Gates” on page 113 for more information).

In far calls and jumps, the far pointer (CS:rIP) references the target code-segment descriptor. Before loading the CS register with a nonconforming code-segment selector, the processor checks as follows to see if access is allowed:

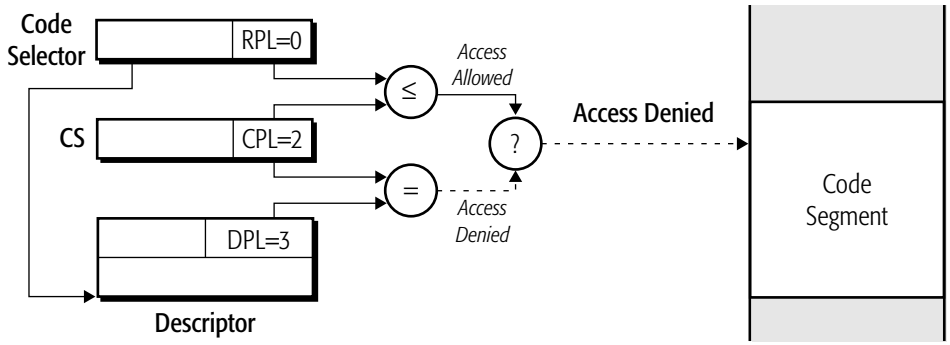
1. *DPL = CPL Check*—The processor compares the target code-segment descriptor DPL with the currently executing program CPL. If they are equal, the processor performs the next check. If they are not equal, a general-protection exception (#GP) occurs.
2. *RPL ≤ CPL Check*—The processor compares the target code-segment selector RPL with the currently executing program CPL. If the RPL is less than or equal to the CPL, access is allowed. If the RPL is greater than the CPL, a #GP exception occurs.

If access is allowed, the processor loads the CS and rIP registers with their new values and begins executing from the target location. The CPL is *not changed*—the target-CS selector RPL value is disregarded when the selector is loaded into the CS register.

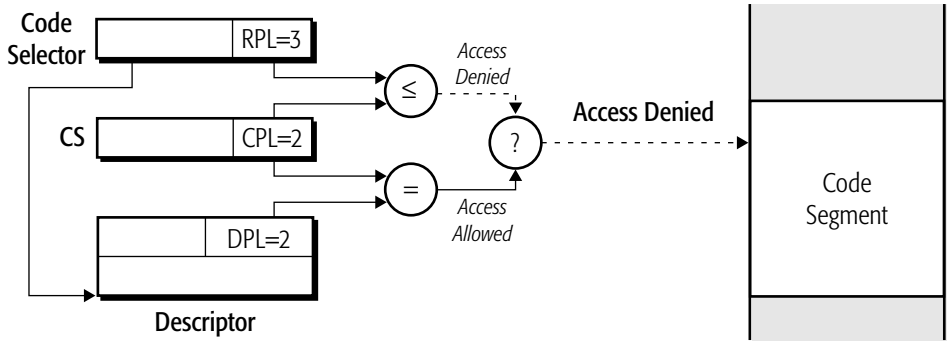
Figure 4-28 on page 111 shows three examples of privilege checks performed as a result of a far control transfer to a nonconforming code-segment. In Example 1, access is allowed because CPL = DPL and RPL ≤ CPL. In Example 2, access is denied because CPL ≠ DPL. In Example 3, access is denied because RPL > CPL.



Example 1: Privilege Check Passes



Example 2: Privilege Check Fails



Example 3: Privilege Check Fails

Figure 4-28. Nonconforming Code-Segment Privilege-Check Examples

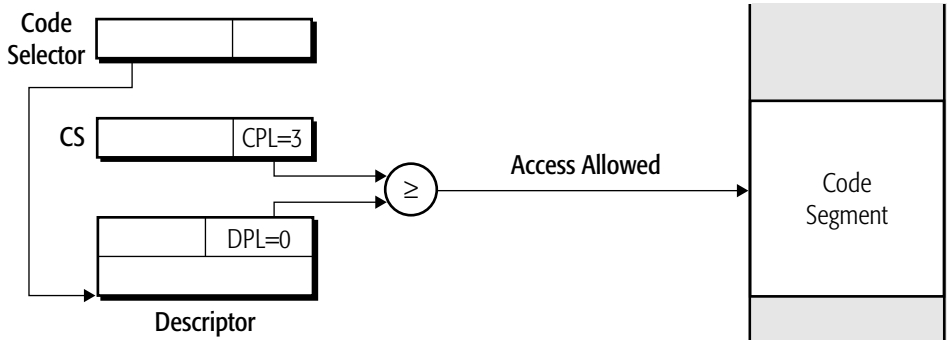
Conforming Code Segments. On a direct control transfer to a conforming code segment, the target code-segment descriptor DPL can be lower than (at a greater privilege) the CPL. Before loading the

CS register with a conforming code-segment selector, the processor compares the target code-segment descriptor DPL with the currently-executing program CPL. If the DPL is less than or equal to the CPL, access is allowed. If the DPL is greater than the CPL, a #GP exception occurs.

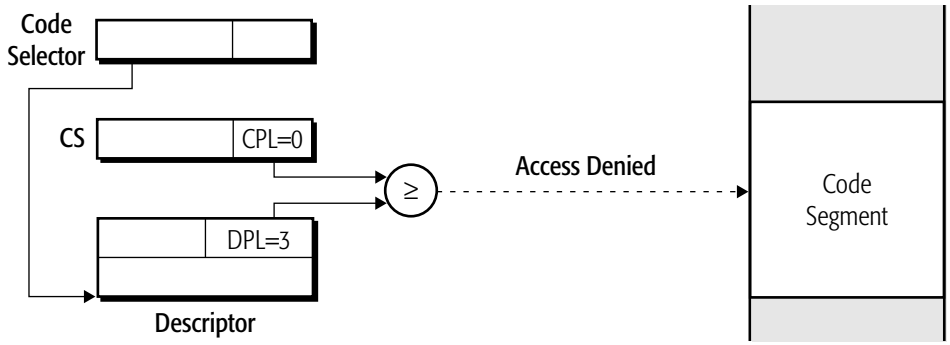
On an access to a conforming code segment, the RPL is ignored and not involved in the privilege check.

When access is allowed, the processor loads the CS and rIP registers with their new values and begins executing from the target location. The CPL is *not changed*—the target CS-descriptor DPL value is disregarded when the selector is loaded into the CS register. The target program runs at the same privilege as the program that called it.

Figure 4-29 shows two examples of privilege checks performed as a result of a direct control transfer to a conforming code segment. In Example 1, access is allowed because the CPL of 3 is greater than the DPL of 0. As the target code selector is loaded into the CS register, the old CPL value of 3 replaces the target-code selector RPL value, and the target program executes with CPL=3. In Example 2, access is denied because CPL < DPL.



Example 1: Privilege Check Passes



Example 2: Privilege Check Fails

Figure 4-29. Conforming Code-Segment Privilege-Check Examples

4.11.2 Control Transfers Through Call Gates

Control transfers to more-privileged code segments are accomplished through the use of *call gates*. Call gates are a type of descriptor that contain pointers to code-segment descriptors and control access to those descriptors. System software uses call gates to establish protected entry points into system-service routines.

Transfer Mechanism. The pointer operand of a far-CALL or far-JMP instruction consists of two pieces: a code-segment selector (CS) and a code-segment offset (rIP). In a call-gate transfer, the CS selector points to a call-gate descriptor rather than a code-segment descriptor, and the rIP is ignored (but required by the instruction).

Figure 4-30 shows a call-gate control transfer in legacy mode. The call-gate descriptor contains segment-selector and segment-offset fields (see “Gate Descriptors” on page 95 for a detailed description of the call-gate format and fields). These two fields perform the same function as the pointer operand in a direct control-transfer instruction. The segment-selector field points to the target code-segment descriptor, and the segment-offset field is the instruction-pointer offset into the target code-segment. The code-segment base taken from the code-segment descriptor is added to the offset field in the call-gate descriptor to create the target virtual address (linear address).

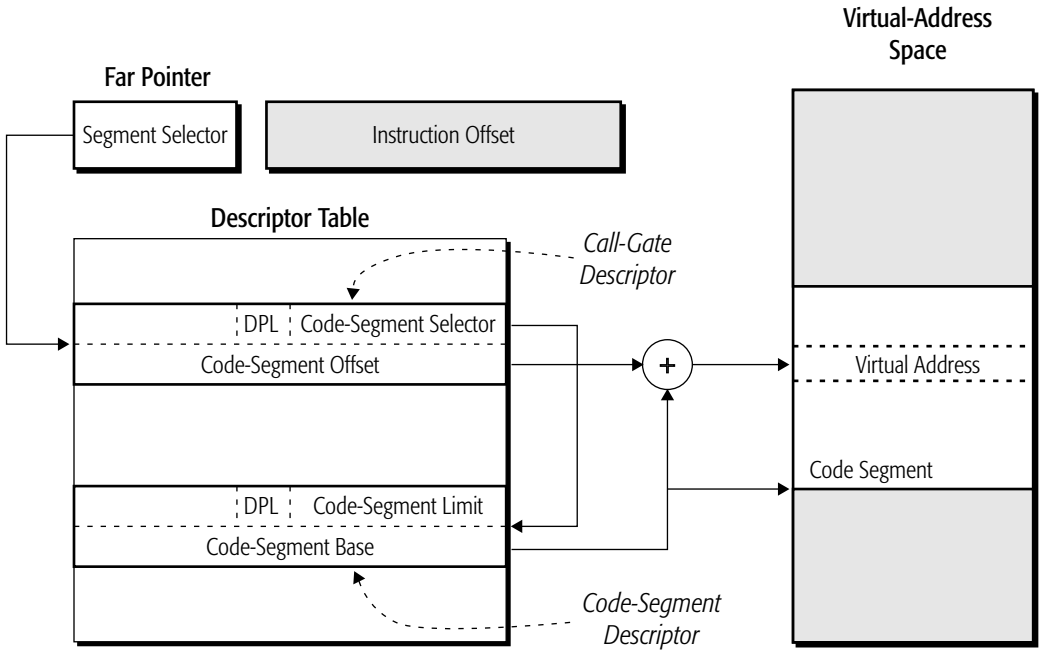


Figure 4-30. Legacy-Mode Call-Gate Transfer Mechanism

Figure 4-31 shows a call-gate control transfer in long mode. The long-mode call-gate descriptor format is expanded by 64 bits to hold a full 64-bit offset into the virtual-address space. Only long-mode call gates can be referenced in long mode (64-bit mode and compatibility mode). The legacy-mode 32-bit call-gate types are redefined in long mode as 64-bit types, and 16-bit call-gate types are illegal.

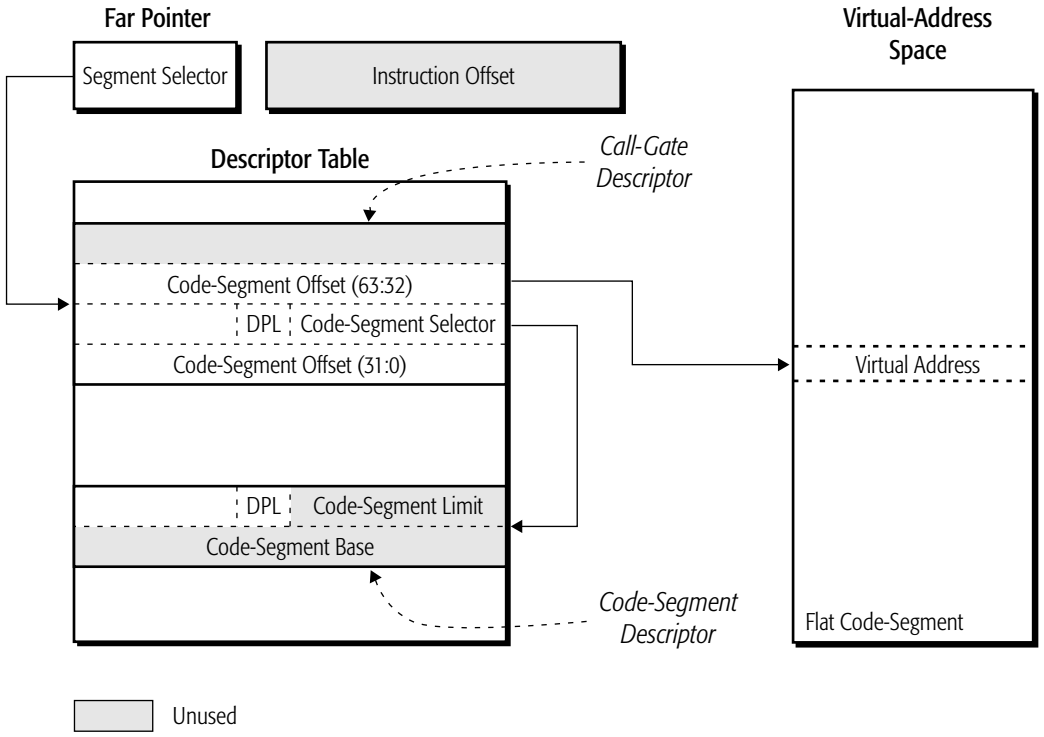


Figure 4-31. Long-Mode Call-Gate Access Mechanism

A long-mode call gate must reference a 64-bit code-segment descriptor. In 64-bit mode, the code-segment descriptor base-address and limit fields are ignored. The target virtual-address is the 64-bit offset field in the expanded call-gate descriptor.

Privilege Checks. Before loading the CS register with the code-segment selector located in the call gate, the processor performs three privilege checks. The following checks are performed when either conforming or nonconforming code segments are referenced:

1. The processor compares the CPL with the call-gate DPL from the call-gate descriptor (DPL_G). The CPL must be numerically *less than or equal to* DPL_G for this check to pass. In other words, the following expression must be true: $CPL \leq DPL_G$.

2. The processor compares the RPL in the call-gate selector with DPL_G. The RPL must be numerically *less than or equal to* DPL_G for this check to pass. In other words, the following expression must be true: $RPL \leq DPL_G$.
3. The processor compares the CPL with the target code-segment DPL from the code-segment descriptor (DPL_S). The type of comparison varies depending on the type of control transfer.
   - When a call—or a jump to a *conforming* code segment—is used to transfer control through a call gate, the CPL must be numerically *greater than or equal to* DPL_S for this check to pass. (This check prevents control transfers to less-privileged programs.) In other words, the following expression must be true: CPL DPL_S.
   - When a JMP instruction is used to transfer control through a call gate to a *nonconforming* code segment, the CPL must be numerically *equal to* DPL_S for this check to pass. (JMP instructions cannot change CPL.) In other words, the following expression must be true: $CPL = DPL_S$.

Figure 4-32 on page 116 shows two examples of call-gate privilege checks. In Example 1, all privilege checks pass as follows:

- The call-gate DPL (DPL_G) is at the lowest privilege (3), specifying that software running at any privilege level (CPL) can access the gate.
- The selector referencing the call gate passes its privilege check because the RPL is numerically less than or equal to DPL_G.
- The target code segment is at the highest privilege level ($DPL_S = 0$). This means software running at any privilege level can access the target code segment through the call gate.

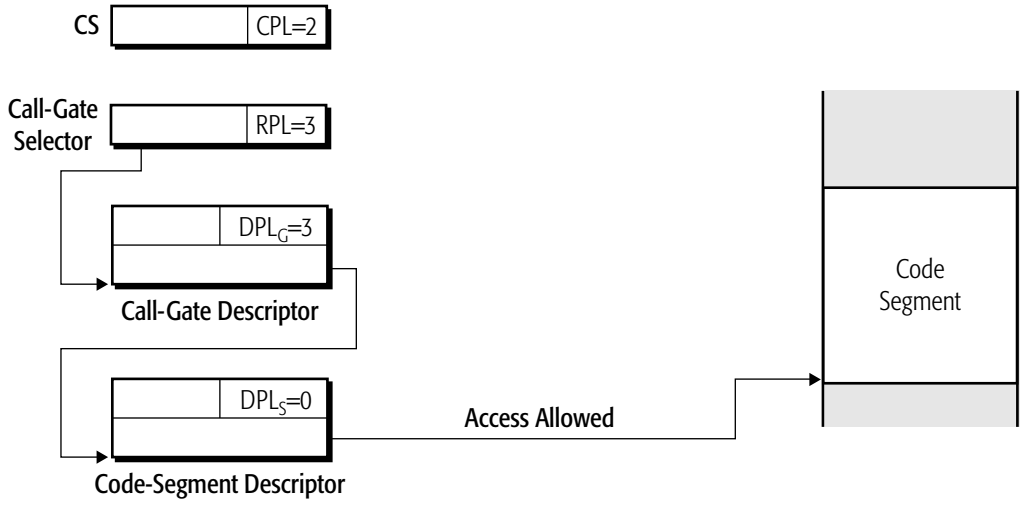


Figure 4-32. Privilege-Check Examples for Call Gates

In Example 2, all privilege checks fail as follows:

- The call-gate DPL (DPL_G) specifies that only software at privilege-level 0 can access the gate. The current program does not have enough privilege to access the call gate because its CPL is 2.
- The selector referencing the call-gate descriptor does not have enough privilege to complete the reference. Its RPL is numerically greater than DPL_G.

- The target code segment is at a lower privilege (DPL_S = 3) than the currently running software (CPL = 2). Transitions from more-privileged software to less-privileged software are not allowed, so this privilege check fails as well.

Although all three privilege checks failed in Example 2, failing only one check is sufficient to deny access into the target code segment.

Stack Switching. The processor performs an automatic stack switch when a control transfer causes a change in privilege levels to occur. Switching stacks isolates more-privileged software stacks from less-privileged software stacks and provides a mechanism for saving the return pointer back to the program that initiated the call.

When switching to more-privileged software, as is done when transferring control using a call gate, the processor uses the corresponding stack pointer (privilege-level 0, 1, or 2) stored in the task-state segment (TSS). The format of the stack pointer stored in the TSS depends on the system-software operating mode:

- Legacy-mode system software stores a 32-bit ESP value (stack offset) and 16-bit SS selector register value in the TSS for each of three privilege levels 0, 1, and 2.
- Long-mode system software stores a 64-bit RSP value in the TSS for privilege levels 0, 1, and 2. No SS register value is stored in the TSS because in long mode a call gate *must* reference a 64-bit code-segment descriptor. 64-bit mode does not use segmentation, and the stack pointer consists solely of the 64-bit RSP. Any value loaded in the SS register is ignored.

See “Task-Management Resources” on page 362 for more information on the legacy-mode and long-mode TSS formats.

Figure 4-33 on page 118 shows a 32-bit stack in legacy mode before and after the automatic stack switch. This particular example assumes that parameters are passed from the current program to the target program. The process followed by legacy mode in switching stacks and copying parameters is:

1. The target code-segment DPL is read by the processor and used as an index into the TSS for selecting the new stack pointer (SS:ESP). For example, if DPL=1 the processor selects the SS:ESP for privilege-level 1 from the TSS.
2. The SS and ESP registers are loaded with the new SS:ESP values read from the TSS.
3. The old values of the SS and ESP registers are pushed onto the stack pointed to by the new SS:ESP.
4. The 5-bit count field is read from the call-gate descriptor.
5. The number of parameters specified in the count field (up to 31) are copied from the old stack to the new stack. The size of the parameters copied by the processor depends on the call-gate size: 32-bit call gates copy 4-byte parameters and 16-bit call gates copy 2-byte parameters.
6. The return pointer is pushed onto the stack. The return pointer consists of the current CS-register value and the EIP of the instruction following the calling instruction.

7. The CS register is loaded from the segment-selector field in the call-gate descriptor, and the EIP is loaded from the offset field in the call-gate descriptor.
8. The target program begins executing with the instruction referenced by new CS:EIP.

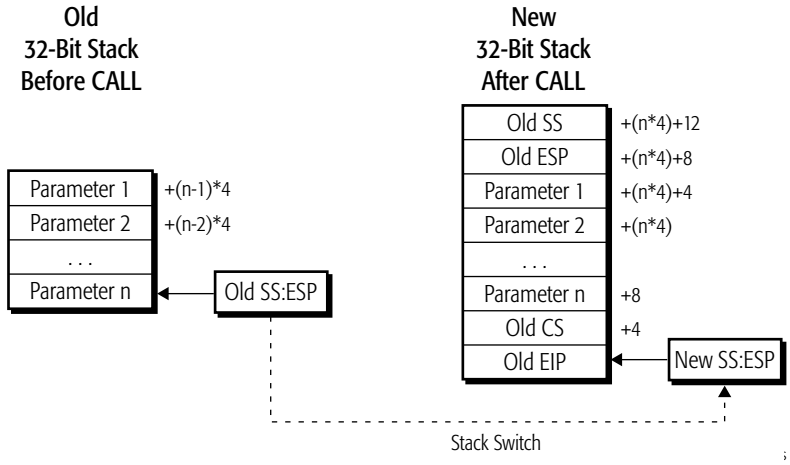


Figure 4-33. Legacy-Mode 32-Bit Stack Switch, with Parameters

Figure 4-34 shows a 32-bit stack in legacy mode before and after the automatic stack switch when no parameters are passed (count=0). Most software does not use the call-gate descriptor count-field to pass parameters. System software typically defines linkage mechanisms that do not rely on automatic parameter copying.

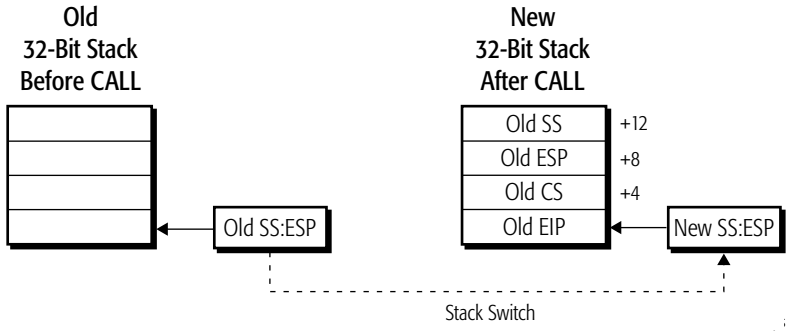


Figure 4-34. 32-Bit Stack Switch, No Parameters—Legacy Mode

Figure 4-35 on page 119 shows a long-mode stack switch. In long mode, all call gates *must* reference 64-bit code-segment descriptors, so a long-mode stack switch uses a 64-bit stack. The process of

switching stacks in long mode is similar to switching in legacy mode when no parameters are passed. The process is as follows:

1. The target code-segment DPL is read by the processor and used as an index into the 64-bit TSS for selecting the new stack pointer (RSP).
2. The RSP register is loaded with the new RSP value read from the TSS. The SS register is loaded with a null selector (SS=0). Setting the new SS selector to null allows proper handling of nested control transfers in 64-bit mode. See “Nested Returns to 64-Bit Mode Procedures” on page 121 for additional information.

   As in legacy mode, it is desirable to keep the stack-segment requestor privilege-level (SS.RPL) equal to the current privilege-level (CPL). When using a call gate to change privilege levels, the SS.RPL is updated to reflect the new CPL. The SS.RPL is restored from the return-target CS.RPL on the subsequent privilege-level-changing far return.
3. The old values of the SS and RSP registers are pushed onto the stack pointed to by the new RSP. The old SS value is popped on a subsequent far return. This allows system software to set up the SS selector for a compatibility-mode process by executing a RET (or IRET) that changes the privilege level.
4. The return pointer is pushed onto the stack. The return pointer consists of the current CS-register value and the RIP of the instruction following the calling instruction.
5. The CS register is loaded from the segment-selector field in the long-mode call-gate descriptor, and the RIP is loaded from the offset field in the long-mode call-gate descriptor.

The target program begins execution with the instruction referenced by the new RIP.

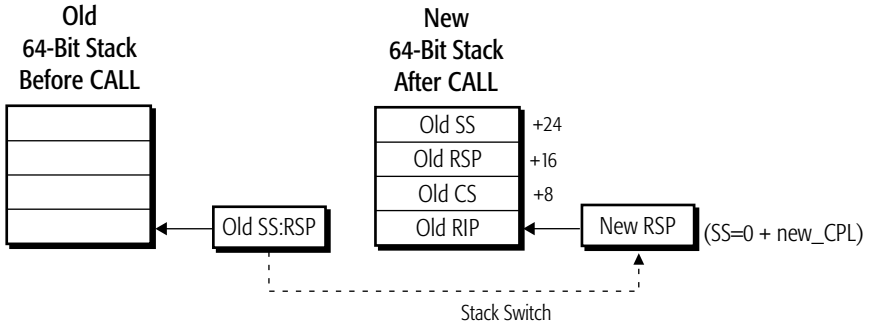


Figure 4-35. Stack Switch—Long Mode

All long-mode stack pushes resulting from a privilege-level-changing far call are eight-bytes wide and increment the RSP by eight. Long mode ignores the call-gate count field and does not support the automatic parameter-copy feature found in legacy mode. Software can access parameters on the old stack, if necessary, by referencing the old stack segment selector and stack pointer saved on the new process stack.

4.11.3 Return Control Transfers

Returns to calling programs can be performed by using the RET instruction. The following types of returns are possible:

- *Near Return*—Near returns perform control transfers within the same code segment, so the CS register is unchanged. The new offset is popped off the stack and into the rIP register. No privilege checks are performed.
- *Far Return, Same Privilege*—A far return transfers control from one code segment to another. When the original code segment is at the same privilege level as the target code segment, a far pointer (CS:rIP) is popped off the stack and the RPL of the new code segment (CS) is checked. If the requested privilege level (RPL) matches the current privilege level (CPL), then a return is made to the same privilege level. This prevents software from changing the CS value on the stack in an attempt to return to higher-privilege software.
- *Far Return, Less Privilege*—Far returns can change privilege levels, but only to a *lower*-privilege level. In this case a stack switch is performed between the current, higher-privilege program and the lower-privilege return program. The CS-register and rIP-register values are popped off the stack. The lower-privilege stack pointer is also popped off the stack and into the SS register and rSP register. The processor checks both the CS and SS privilege levels to ensure they are equal and at a lesser privilege than the current CS.

  In the case of nested returns to 64-bit mode, a null selector can be popped into the SS register. See “Nested Returns to 64-Bit Mode Procedures” on page 121.

  Far returns also check the privilege levels of the DS, ES, FS and GS selector registers. If any of these segment registers have a selector with a higher privilege than the return program, the segment register is loaded with the null selector.

Stack Switching. The stack switch performed by a far return to a lower-privilege level reverses the stack switch of a call gate to a higher-privilege level, except that parameters are never automatically copied as part of a return. The process followed by a far-return stack switch in long mode and legacy mode is:

1. The return code-segment RPL is read by the processor from the CS value stored on the stack to determine that a lower-privilege control transfer is occurring.
2. The return-program instruction pointer is popped off the current-program (higher privilege) stack and loaded into the CS and rIP registers.
3. The return instruction can include an immediate operand that specifies the number of additional bytes to be popped off of the stack. These bytes may correspond to the parameters pushed onto the stack previously by a call through a call gate containing a non-zero parameter-count field. If the return includes the immediate operand, then the stack pointer is adjusted upward by adding the specified number of bytes to the rSP.
4. The return-program stack pointer is popped off the current-program (higher privilege) stack and loaded into the SS and rSP registers. In the case of nested returns to 64-bit mode, a null selector can be popped into the SS register.

The operand size of a far return determines the size of stack pops when switching stacks. If a far return is used in 64-bit mode to return from a prior call through a long-mode call gate, the far return must use a 64-bit operand size. The 64-bit operand size allows the far return to properly read the stack established previously by the far call.

Nested Returns to 64-Bit Mode Procedures. In long mode, a far call that changes privilege levels causes the SS register to be loaded with a null selector (this is the same action taken by an interrupt in long mode). If the called procedure performs another far call to a higher-privileged procedure, or is interrupted, the null SS selector is pushed onto the stack frame, and another null selector is loaded into the SS register. Using a null selector in this way allows the processor to properly handle returns nested within 64-bit-mode procedures and interrupt handlers.

Normally, a RET that pops a null selector into the SS register causes a general-protection exception (#GP) to occur. However, in long mode, the null selector acts as a flag indicating the existence of nested interrupt handlers or other privileged software in 64-bit mode. Long mode allows RET to pop a null selector into SS from the stack under the following conditions:

- The target mode is 64-bit mode.
- The target CPL is less than 3.

In this case, the processor does not load an SS descriptor, and the null selector is loaded into SS without causing a #GP exception.

4.12 Limit Checks

Except in 64-bit mode, limit checks are performed by all instructions that reference memory. Limit checks detect attempts to access memory outside the current segment boundary, attempts at executing instructions outside the current code segment, and indexing outside the current descriptor table. If an instruction fails a limit check, either (1) a general-protection exception occurs for all other segment-limit violations or (2) a stack-fault exception occurs for stack-segment limit violations.

In 64-bit mode, segment limits are *not checked* during accesses to any segment referenced by the CS, DS, ES, FS, GS, and SS selector registers. Instead, the processor checks that the virtual addresses used to reference memory are in canonical-address form. In 64-bit mode, as with legacy mode and compatibility mode, descriptor-table limits *are checked*.

4.12.1 Determining Limit Violations

To determine segment-limit violations, the processor checks a virtual (linear) address to see if it falls outside the valid range of segment offsets determined by the segment-limit field in the descriptor. If any part of an operand or instruction falls outside the segment-offset range, a limit violation occurs. For example, a doubleword access, two bytes from an upper segment boundary, causes a segment violation because half of the doubleword is outside the segment.

Three bits from the descriptor entry are used to control how the segment-limit field is interpreted: the granularity (G) bit, the default operand-size (D) bit, and for data segments, the expand-down (E) bit. See “Legacy Segment Descriptors” on page 88 for a detailed description of each bit.

For all segments other than expand-down segments, the minimum segment-offset is 0. The maximum segment-offset depends on the value of the G bit:

- If G=0 (byte granularity), the maximum allowable segment-offset is equal to the value of the segment-limit field.
- If G=1 (4096-byte granularity), the segment-limit field is first scaled by 4096 (1000h). Then 4095 (0FFFh) is added to the scaled value to arrive at the maximum allowable segment-offset, as shown in the following equation:

  maximum segment-offset = (limit × 1000h) + 0FFFh

  For example, if the segment-limit field is 0100h, then the maximum allowable segment-offset is

  (0100h × 1000h) + 0FFFh = 10_1FFFh.

In both cases, the maximum segment-size is specified when the descriptor segment-limit field is 0F_FFFFh.

Expand-Down Segments. Expand-down data segments are supported in legacy mode and compatibility mode but not in 64-bit mode. With expand-down data segments, the maximum segment offset depends on the value of the D bit in the data-segment descriptor:

- If D=0 the maximum segment-offset is 0_FFFFh.
- If D=1 the maximum segment-offset is 0_FFFF_FFFFh.

The minimum allowable segment offset in expand-down segments depends on the value of the G bit:

- If G=0 (byte granularity), the minimum allowable segment offset is the segment-limit value plus 1.

  For example, if the segment-limit field is 0100h, then the minimum allowable segment-offset is 0101h.
- If G=1 (4096-byte granularity), the segment-limit value in the descriptor is first scaled by 4096 (1000h), and then 4095 (0FFFh) is added to the scaled value to arrive at a scaled segment-limit value. The minimum allowable segment-offset is this scaled segment-limit value plus 1, as shown in the following equation:

  minimum segment-offset = (limit × 1000) + 0FFFh + 1

  For example, if the segment-limit field is 0100h, then the minimum allowable segment-offset is

  (0100h × 1000h) + 0FFFh + 1 = 10_1000h.

For expand-down segments, the maximum segment size is specified when the segment-limit value is 0.

4.12.2 Data Limit Checks in 64-bit Mode

In 64-bit mode, data reads and writes are not normally checked for segment-limit violations. When EFER.LMSLE = 1, reads and writes in 64-bit mode at CPL > 0, using the DS, ES, FS, or SS segments, have a segment-limit check applied.

This limit-check uses the 32-bit segment-limit to find the maximum allowable address in the top 4GB of the 64-bit virtual (linear) address space.

Table 4-8. Segment Limit Checks in 64-Bit Mode

Memory Address	Effect of Limit Check
Linear Address £ (0FFFFFFFF_00000000h + 32-bit Limit)	Access OK.
Linear Address > (0FFFFFFFF_00000000h + 32-bit Limit)	Exception (#GP or #SS)

This segment-limit check does not apply to accesses through the GS segment, or to code reads. If the DS, ES, FS, or SS segment is null or expand-down, the effect of the limit check is undefined. Data segment limit checking in 64-bit mode is not supported by all processor implementations and has been deprecated. If CPUID Fn8000_0008_EBX[EferLmlseUnsupported](bit 20) = 1, 64-bit mode segment limit checking is not supported and attempting to enable this feature by setting EFER.LMSLE =1 will result in a #GP exception.

4.13 Type Checks

Type checks prevent software from using descriptors in invalid ways. Failing a type check results in an exception. Type checks are performed using five bits from the descriptor entry: the S bit and the 4-bit Type field. Together, these five bits are used to specify the descriptor type (code, data, segment, or gate) and its access characteristics. See “Legacy Segment Descriptors” on page 88 for a detailed description of the S bit and Type-field encodings. Type checks are performed by the processor in compatibility mode as well as legacy mode. Limited type checks are performed in 64-bit mode.

4.13.1 Type Checks in Legacy and Compatibility Modes

The type checks performed in legacy mode and compatibility mode are listed in the following sections.

Descriptor-Table Register Loads. Loads into the LDTR and TR descriptor-table registers are checked for the appropriate system-segment type. The LDTR can only be loaded with an LDT descriptor, and the TR only with a TSS descriptor. The checks are performed during any action that causes these registers to be loaded. This includes execution of the LLDT and LTR instructions and during task switches.

Segment Register Loads. The following restrictions are placed on the segment-descriptor types that can be loaded into the six user segment registers:

- Only code segments can be loaded into the CS register.
- Only writable data segments can be loaded into the SS register.

- Only the following segment types can be loaded into the DS, ES, FS, or GS registers:
  - Read-only or read/write data segments.
  - Readable code segments.

These checks are performed during any action that causes the segment registers to be loaded. This includes execution of the MOV segment-register instructions, control transfers, and task switches.

Control Transfers. Control transfers (branches and interrupts) place additional restrictions on the segment types that can be referenced during the transfer:

- The segment-descriptor type referenced by far CALLs and far JMPs must be one of the following:
  - A code segment
  - A call gate or a task gate
  - An available TSS (only allowed in legacy mode)
  - A task gate (only allowed in legacy mode)
- Only code-segment descriptors can be referenced by call-gate, interrupt-gate, and trap-gate descriptors.
- Only TSS descriptors can be referenced by task-gate descriptors.
- The link field (selector) in the TSS can only point to a TSS descriptor. This is checked during an IRET control transfer to a task.
- The far RET and far IRET instructions can only reference code-segment descriptors.
- The interrupt-descriptor table (IDT), which is referenced during interrupt control transfers, can only contain interrupt gates, trap gates, and task gates.

Segment Access. After a segment descriptor is successfully loaded into one of the segment registers, reads and writes into the segments are restricted in the following ways:

- Writes are not allowed into read-only data-segment types.
- Writes are not allowed into code-segment types (executable segments).
- Reads from code-segment types are not allowed if the readable (R) type bit is cleared to 0.

These checks are generally performed during execution of instructions that access memory.

4.13.2 Long Mode Type Check Differences

Compatibility Mode and 64-Bit Mode. The following type checks differ in long mode (64-bit mode and compatibility mode) as compared to legacy mode:

- *System Segments*—System-segment types are checked, but the following types that are valid in legacy mode are illegal in long mode:
  - 16-bit available TSS.
  - 16-bit busy TSS.

  - Type-field encoding of 00h in the upper half of a system-segment descriptor to indicate an illegal type and prevent access as a legacy descriptor.
- *Gates*—Gate-descriptor types are checked, but the following types that are valid in legacy mode are illegal in long mode:
  - 16-bit call gate.
  - 16-bit interrupt gate.
  - 16-bit trap gate.
  - Task gate.

64-Bit Mode. 64-bit mode disables segmentation, and most of the segment-descriptor fields are ignored. The following list identifies situations where type checks in 64-bit mode differ from those in compatibility mode and legacy mode:

- *Code Segments*—The readable (R) type bit is ignored in 64-bit mode. None of the legacy type-checks that prevent reads from or writes into code segments are performed in 64-bit mode.
- *Data Segments*—Data-segment type attributes are ignored in 64-bit mode. The writable (W) and expand-down (E) type bits are ignored. All data segments are treated as writable.

5 Page Translation and Protection

The x86 page-translation mechanism (or simply *paging mechanism*) enables system software to create separate address spaces for each process or application. These address spaces are known as *virtual-address* spaces. System software uses the paging mechanism to selectively map individual pages of physical memory into the virtual-address space using a set of hierarchical address-translation tables known collectively as *page tables*.

The paging mechanism and the page tables are used to provide each process with its own private region of physical memory for storing its code and data. Processes can be protected from each other by isolating them within the virtual-address space. A process cannot access physical memory that is not mapped into its virtual-address space by system software.

System software can use the paging mechanism to selectively map physical-memory pages into multiple virtual-address spaces. Mapping physical pages in this manner allows them to be shared by multiple processes and applications. The physical pages can be configured by the page tables to allow read-only access. This prevents applications from altering the pages and ensures their integrity for use by all applications.

Shared mapping is typically used to allow access of shared-library routines by multiple applications. A read-only copy of the library routine is mapped to each application virtual-address space, but only a single copy of the library routine is present in physical memory. This capability also allows a copy of the operating-system kernel and various device drivers to reside within the application address space. Applications are provided with efficient access to system services without requiring costly address-space switches.

The system-software portion of the address space necessarily includes system-only data areas that must be protected from accesses by applications. System software uses the page tables to protect this memory by designating the pages as *supervisor* pages. Such pages are only accessible by system software.

When the supervisor mode execution prevention (SMEP) feature is supported and enabled, attempted instruction fetches from user-mode accessible pages while in supervisor-mode triggers a page fault (#PF). This protects the integrity of system software by preventing the execution of instructions at a supervisor privilege level (CPL < 3) when these instructions could have been written or modified by user-mode code.

When the supervisor mode access prevention (SMAP) feature is supported and enabled and RFLAGS.AC=0, some attempted explicit data accesses from user-mode accessible pages while in supervisor-mode trigger a page fault (#PF). This protects the integrity of system software by preventing the use of data at a supervisor privilege level (CPL < 3) that could have been modified by user-mode code. Supervisor software that requires access to data which is marked as user-mode accessible may temporarily suppress SMAP checks by modifying RFLAGS.AC, such as with the CLAC/STAC instructions.

Finally, system software can use the paging mechanism to map multiple, large virtual-address spaces into a much smaller amount of physical memory. Each application can use the entire 32-bit or 64-bit virtual-address space. System software actively maps the most-frequently-used virtual-memory pages into the available pool of physical-memory pages. The least-frequently-used virtual-memory pages are swapped out to the hard drive. This process is known as *demand-paged virtual memory*.

5.1 Page Translation Overview

The legacy x86 architecture provides support for translating 32-bit virtual addresses into 32-bit physical addresses (larger physical addresses, such as 36-bit or 40-bit addresses, are supported as a special mode). The AMD64 architecture enhances this support to allow translation of 64-bit virtual addresses into 52-bit physical addresses, although processor implementations can support smaller virtual-address and physical-address spaces.

Virtual addresses are translated to physical addresses through hierarchical translation tables created and managed by system software. Each table contains a set of entries that point to the next-lower table in the translation hierarchy. A single table at one level of the hierarchy can have hundreds of entries, each of which points to a unique table at the next-lower hierarchical level. Each lower-level table can in turn have hundreds of entries pointing to tables further down the hierarchy. The lowest-level table in the hierarchy points to the translated physical page.

Figure 5-1 on page 130 shows an overview of the page-translation hierarchy used in long mode. Legacy mode paging uses a subset of this translation hierarchy (the page-map level-4 table does not exist in legacy mode and the PDP table may or may not be used, depending on which paging mode is enabled). As this figure shows, a virtual address is divided into fields, each of which is used as an offset into a translation table. The complete translation chain is made up of all table entries referenced by the virtual-address fields. The lowest-order virtual-address bits are used as the byte offset into the physical page.

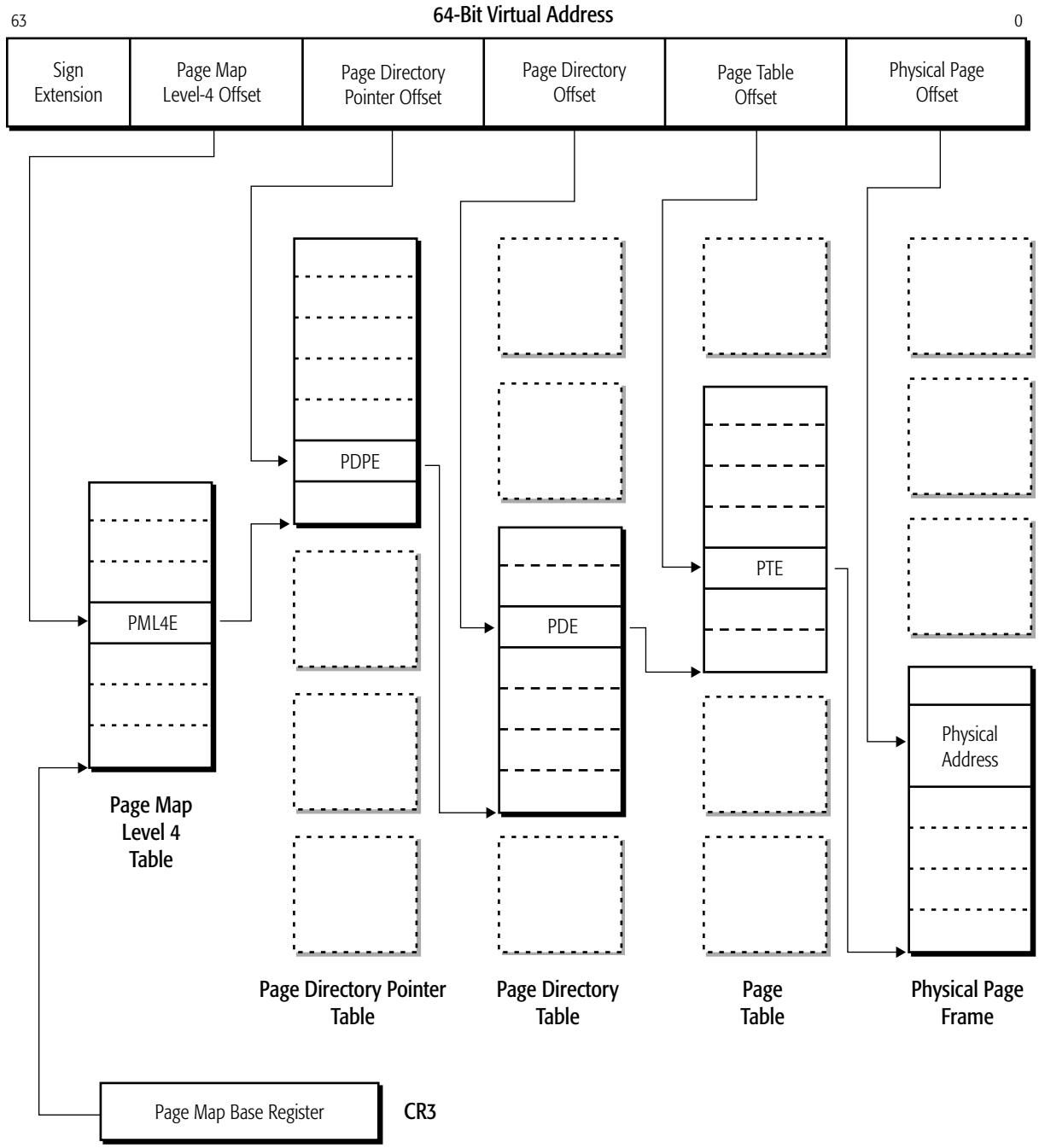


Figure 5-1. Virtual to Physical Address Translation—Long Mode

The following physical-page sizes are supported: 4 Kbytes, 2 Mbytes, 4 Mbytes, and 1 Gbytes. In long mode 4-Kbyte, 2-MByte, and 1-GByte sizes are available. In legacy mode 4-Kbyte, 2-MByte, and 4-MByte sizes are available.

Virtual addresses are 32 bits long, and physical addresses up to the supported physical-address size can be used. The AMD64 architecture enhances the legacy translation support by allowing virtual addresses of up to 64 bits long to be translated into physical addresses of up to 52 bits long.

Currently, the AMD64 architecture defines a mechanism for translating 48-bit virtual addresses to 52-bit physical addresses. The mechanism used to translate a full 64-bit virtual address is reserved and will be described in a future AMD64 architectural specification.

5.1.1 Page-Translation Options

The form of page-translation support available to software depends on which paging features are enabled. Four controls are available for selecting the various paging alternatives:

- Page-Translation Enable (CR0.PG)
- Physical-Address Extensions (CR4.PAE)
- Page-Size Extensions (CR4.PSE)
- Long-Mode Active (EFER.LMA)

Not all paging alternatives are available in all modes. Table 5-1 summarizes the paging support available in each mode.

Table 5-1. Supported Paging Alternatives (CR0.PG=1)

Mode	Physical-Address Extensions (CR4.PAE)	Page-Size Extensions (CR4.PSE)	Page-Directory Pointer Offset	Page-Directory Page Size	Resulting Physical-Page Size	Maximum Virtual Address	Maximum Physical Address
Long Mode (64-bit and compatability modes)	Enabled	–	PDPE.PS=0	PDE.PS=0	4 Kbyte	64-bit	52-bit
				PDE.PS=1	2 Mbyte		
			PDPE.PS=1	–	1 Gbyte		
Legacy Mode	Enabled	–	PDPE.PS=0	PDE.PS=0	4 Kbyte	32-bit	52-bit
				PDE.PS=1	2 Mbyte		52-bit
	Disabled	Disabled		–	4 Kbyte		32-bit
		Enabled		PDE.PS=0	4 Kbyte		32-bit
				PDE.PS=1	4 Mbyte		40-bit

5.1.2 Page-Translation Enable (PG) Bit

Page translation is controlled by the PG bit in CR0 (bit 31). When CR0.PG is set to 1, page translation is enabled. When CR0.PG is cleared to 0, page translation is disabled.

The AMD64 architecture uses CR0.PG to activate and deactivate long mode when long mode is enabled. See “Enabling and Activating Long Mode” on page 474 for more information.

5.1.3 Physical-Address Extensions (PAE) Bit

Physical-address extensions are controlled by the PAE bit in CR4 (bit 5). When CR4.PAE is set to 1, physical-address extensions are enabled. When CR4.PAE is cleared to 0, physical-address extensions are disabled.

Setting CR4.PAE = 1 enables virtual addresses to be translated into physical addresses up to 52 bits long. This is accomplished by doubling the size of paging data-structure entries from 32 bits to 64 bits to accommodate the larger physical base-addresses for physical-pages.

PAE must be enabled before activating long mode. See “Enabling and Activating Long Mode” on page 474.

5.1.4 Page-Size Extensions (PSE) Bit

Page-size extensions are controlled by the PSE bit in CR4 (bit 4). Setting CR4.PSE to 1 allows operating-system software to use 4-Mbyte physical pages in the translation process. The 4-Mbyte physical pages can be mixed with standard 4-Kbyte physical pages or replace them entirely. The selection of physical-page size is made on a page-directory-entry basis. See “Page Size (PS) Bit” on page 152 for more information on physical-page size selection. When CR4.PSE is cleared to 0, page-size extensions are disabled.

The choice of 2 Mbyte or 4 Mbyte as the large physical-page size depends on the value of CR4.PSE and CR4.PAE, as follows:

- If physical-address extensions are enabled (CR4.PAE=1), the large physical-page size is 2 Mbytes, regardless of the value of CR4.PSE.
- If physical-address extensions are disabled (CR4.PAE=0) *and* CR4.PSE=1, the large physical-page size is 4 Mbytes.
- If both CR4.PAE=0 and CR4.PSE=0, the only available page size is 4 Kbytes.

The value of CR4.PSE is ignored when long mode is active. This is because physical-address extensions must be enabled in long mode, and the only available page sizes are 4 Kbytes and 2 Mbytes.

In legacy mode, physical addresses up to 40 bits long can be translated from 32-bit virtual addresses using 32-bit paging data-structure entries when 4-Mbyte physical-page sizes are selected. In this special case, CR4.PSE=1 and CR4.PAE=0. See “4-Mbyte Page Translation” on page 136 for a description of the 4-Mbyte PDE that supports 40-bit physical-address translation. The 40-bit physical-address capability is an AMD64 architecture enhancement over the similar capability available in the legacy x86 architecture.

5.1.5 Page-Directory Page Size (PS) Bit

The page directory offset entry (PDE) and page directory pointer offset entry (PDPE) are data structures used in page translation (see Figure 5-1 on page 130). The page-size (PS) bit in the PDE (bit 7, referred to as PDE.PS) selects between standard 4-Kbyte physical-page sizes and larger (2-Mbyte or 4-Mbyte) physical-page sizes. The page-size (also PS) bit in the PDPE (bit 7, referred to as PDPE.PS) selects between 2-Mbyte and 1-Gbyte physical-page sizes in long mode.

When PDE.PS is set to 1, large physical pages are used, and the PDE becomes the lowest level of the translation hierarchy. The size of the large page is determined by the values of CR4.PAE and CR4.PSE, as shown in Figure 5-1 on page 131. When PDE.PS is cleared to 0, standard 4-Kbyte physical pages are used, and the PTE is the lowest level of the translation hierarchy.

When PDPE.PS is set to 1, 1-Gbyte physical pages are used, and the PDPE becomes the lowest level of the translation hierarchy. Neither the PDE nor PTE are used for 1-Gbyte paging.

5.2 Legacy-Mode Page Translation

Legacy mode supports two forms of translation:

- *Normal* (non-PAE) *Paging*—This is used when physical-address extensions are disabled (CR4.PAE=0). Entries in the page translation table are 32 bits and are used to translate 32-bit virtual addresses into physical addresses as large as 40 bits.
- *PAE Paging*—This is used when physical-address extensions are enabled (CR4.PAE=1). Entries in the page translation table are 64 bits and are used to translate 32-bit virtual addresses into physical addresses as large as 52 bits.

Legacy paging uses up to three levels of page-translation tables, depending on the paging form used and the physical-page size. Entries within each table are selected using virtual-address bit fields. The legacy page-translation tables are:

- *Page Table*—Each page-table entry (PTE) points to a physical page. If 4-Kbyte pages are used, the page table is the lowest level of the page-translation hierarchy. PTEs are not used when translating 2-Mbyte or 4-Mbyte pages.
- *Page Directory*—If 4-Kbyte pages are used, each page-directory entry (PDE) points to a page table. If 2-Mbyte or 4-Mbyte pages are used, a PDE is the lowest level of the page-translation hierarchy and points to a physical page. In non-PAE paging, the page directory is the highest level of the translation hierarchy.
- *Page-Directory Pointer*—Each page-directory pointer entry (PDPE) points to a page directory. Page-directory pointers are only used in PAE paging (CR4.PAE=1), and are the highest level in the legacy page-translation hierarchy.

The translation-table-entry formats and how they are used in the various forms of legacy page translation are described beginning on page 135.

5.2.1 CR3 Register

The CR3 register is used to point to the base address of the highest-level page-translation table. The base address is either the page-directory pointer table or the page directory table. The CR3 register format depends on the form of paging being used. Figure 5-2 on page 134 shows the CR3 format when normal (non-PAE) paging is used (CR4.PAE=0). Figure 5-3 shows the CR3 format when PAE paging is used (CR4.PAE=1).

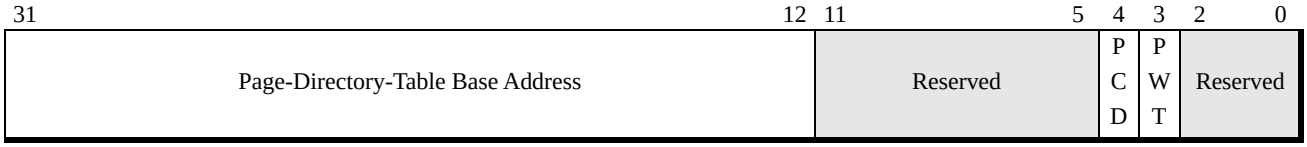


Figure 5-2. Control Register 3 (CR3)—Non-PAE Paging Legacy-Mode

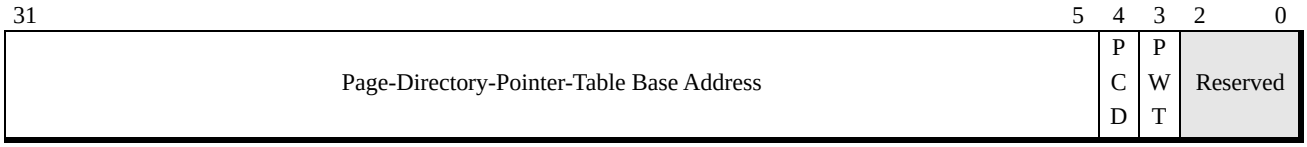


Figure 5-3. Control Register 3 (CR3)—PAE Paging Legacy-Mode

The CR3 register fields for legacy-mode paging are:

Table Base Address Field. This field points to the starting physical address of the highest-level page-translation table. The size of this field depends on the form of paging used:

- *Normal (Non-PAE) Paging (CR4.PAE=0)*—This 20-bit field occupies bits 31:12, and points to the base address of the page-directory table. The page-directory table is aligned on a 4-Kbyte boundary, with the low-order 12 address bits 11:0 assumed to be 0. This yields a total base-address size of 32 bits.
- *PAE Paging (CR4.PAE=1)*—This field is 27 bits and occupies bits 31:5. The CR3 register points to the base address of the page-directory-pointer table. The page-directory-pointer table is aligned on a 32-byte boundary, with the low 5 address bits 4:0 assumed to be 0.

Page-Level Writethrough (PWT) Bit. Bit 3. Page-level writethrough indicates whether the highest-level page-translation table has a writeback or writethrough caching policy. When PWT=0, the table has a writeback caching policy. When PWT=1, the table has a writethrough caching policy.

Page-Level Cache Disable (PCD) Bit. Bit 4. Page-level cache disable indicates whether the highest-level page-translation table is cacheable. When PCD=0, the table is cacheable. When PCD=1, the table is not cacheable.

Reserved Bits. Reserved fields should be cleared to 0 by software when writing CR3.

5.2.2 Normal (Non-PAE) Paging

Non-PAE paging (CR4.PAE=0) supports 4-Kbyte and 4-Mbyte physical pages, as described in the following sections.

4-Kbyte Page Translation. 4-Kbyte physical-page translation is performed by dividing the 32-bit virtual address into three fields. Each of the upper two fields is used as an index into a two-level page-translation hierarchy. The virtual-address fields are used as follows, and are shown in Figure 5-4:

- Bits 31:22 index into the 1024-entry page-directory table.
- Bits 21:12 index into the 1024-entry page table.
- Bits 11:0 provide the byte offset into the physical page.

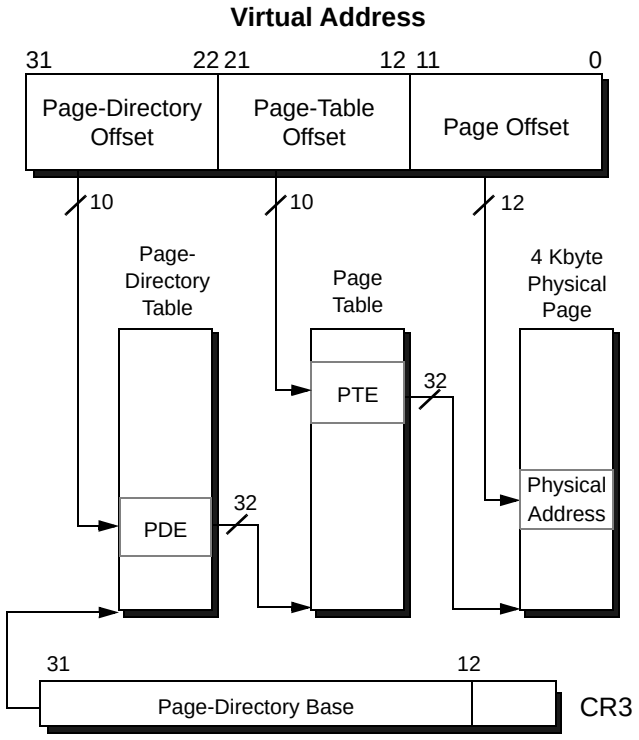


Figure 5-4. 4-Kbyte Non-PAE Page Translation—Legacy Mode

Figure 5-5 on page 136 shows the format of the PDE (page-directory entry), and Figure 5-6 on page 136 shows the format of the PTE (page-table entry). Each table occupies 4 Kbytes and can hold 1024 of the 32-bit table entries. The fields within these table entries are described in "Page-Translation-Table Entry Fields" on page 150.

Figure 5-5 shows bit 7 cleared to 0. This bit is the *page-size* bit (PS), and specifies a 4-Kbyte physical-page translation.

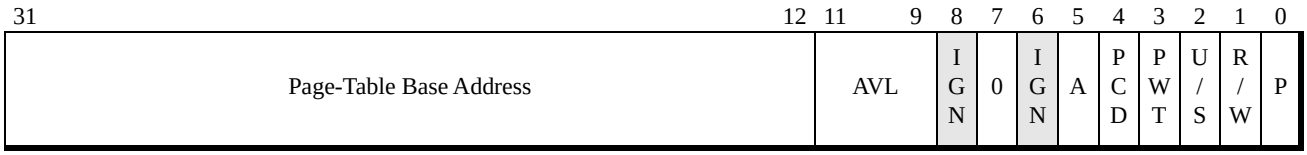


Figure 5-5. 4-Kbyte PDE—Non-PAE Paging Legacy-Mode

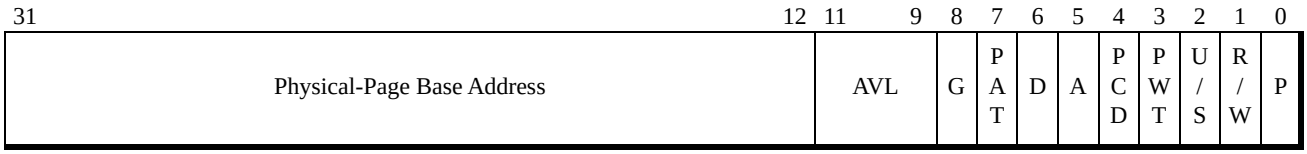


Figure 5-6. 4-Kbyte PTE—Non-PAE Paging Legacy-Mode

4-Mbyte Page Translation. 4-Mbyte page translation is only supported when page-size extensions are enabled (CR4.PSE=1) and physical-address extensions are disabled (CR4.PAE=0).

PSE defines a page-size bit in the 32-bit PDE format (PDE.PS). This bit is used by the processor during page translation to support both 4-Mbyte and 4-Kbyte pages. 4-Mbyte pages are selected when PDE.PS is set to 1, and the PDE points directly to a 4-Mbyte physical page. PTEs are not used in a 4-Mbyte page translation. If PDE.PS is cleared to 0, or if 4-Mbyte page translation is disabled, the PDE points to a PTE.

4-Mbyte page translation is performed by dividing the 32-bit virtual address into two fields. Each field is used as an index into a single-level page-translation hierarchy. The virtual-address fields are used as follows, and are shown in Figure 5-7 on page 137:

- Bits 31:22 index into the 1024-entry page-directory table.
- Bits 21:0 provide the byte offset into the physical page.

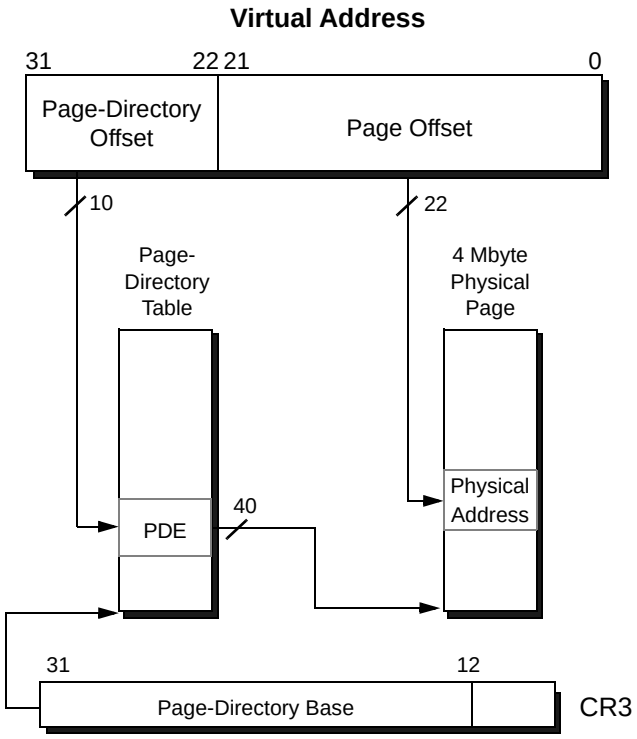


Figure 5-7. 4-Mbyte Page Translation—Non-PAE Paging Legacy-Mode

The AMD64 architecture modifies the legacy 32-bit PDE format in PSE mode to increase physical-address size support to 40 bits. This increase in address size is accomplished by using bits 20:13 to hold eight additional high-order physical-address bits. Bit 21 is reserved and must be cleared to 0.

Figure 5-8 shows the format of the PDE when PSE mode is enabled. The physical-page base-address bits are contained in a split field. The high-order, physical-page base-address bits 39:32 are located in PDE[20:13], and physical-page base-address bits 31:22 are located in PDE[31:22].

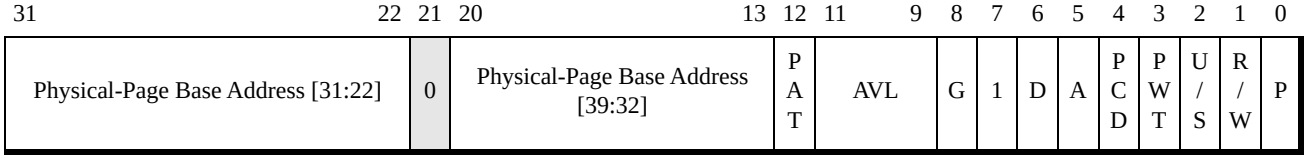


Figure 5-8. 4-Mbyte PDE—Non-PAE Paging Legacy-Mode

5.2.3 PAE Paging

PAE paging is used when physical-address extensions are enabled (CR4.PAE=1). PAE paging doubles the size of page-translation table entries to 64 bits so that the table entries can hold larger physical

addresses (up to 52 bits). The size of each table remains 4 Kbytes, which means each table can hold 512 of the 64-bit entries. PAE paging also introduces a third-level page-translation table, known as the page-directory-pointer table (PDPT).

The size of large pages in PAE-paging mode is 2 Mbytes rather than 4 Mbytes. PAE uses the page-directory page-size bit (PDE.PS) to allow selection between 4-Kbyte and 2-Mbyte page sizes. PAE automatically uses the page-size bit, so the value of CR4.PSE is ignored by PAE paging.

4-Kbyte Page Translation. With PAE paging, 4-Kbyte physical-page translation is performed by dividing the 32-bit virtual address into four fields, each of the upper three fields is used as an index into a 3-level page-translation hierarchy. The virtual-address fields are described as follows and are shown in Figure 5-9:

- Bits 31:30 index into a 4-entry page-directory-pointer table.
- Bits 29:21 index into the 512-entry page-directory table.
- Bits 20:12 index into the 512-entry page table.
- Bits 11:0 provide the byte offset into the physical page.

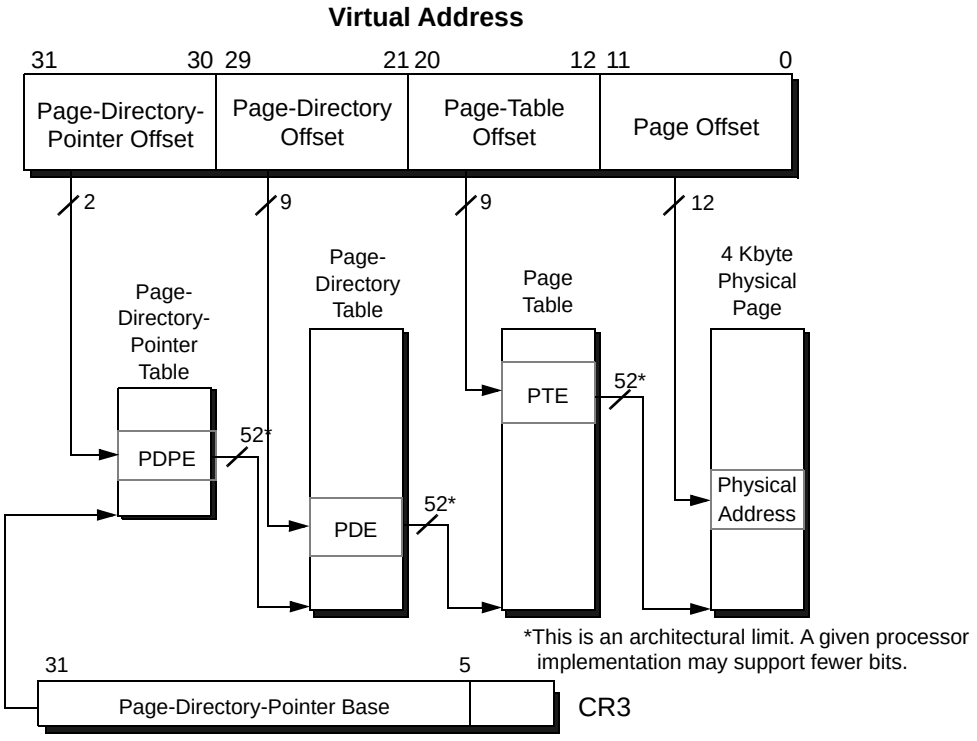


Figure 5-9. 4-Kbyte PAE Page Translation—Legacy Mode

Figures 5-10 through 5-12 show the legacy-mode 4-Kbyte translation-table formats:

- Figure 5-10 shows the PDPE (page-directory-pointer entry) format.
- Figure 5-11 shows the PDE (page-directory entry) format.
- Figure 5-12 shows the PTE (page-table entry) format.

The fields within these table entries are described in “Page-Translation-Table Entry Fields” on page 150.

Figure 5-11 shows the PDE.PS bit cleared to 0 (bit 7), specifying a 4-Kbyte physical-page translation.

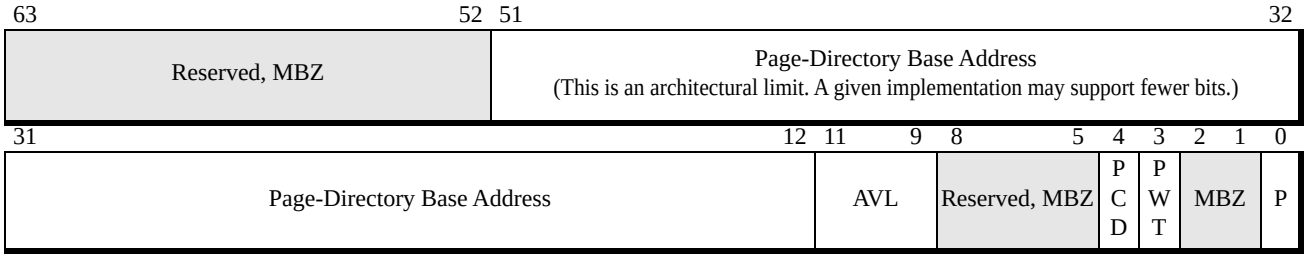


Figure 5-10. 4-Kbyte PDPE—PAE Paging Legacy-Mode

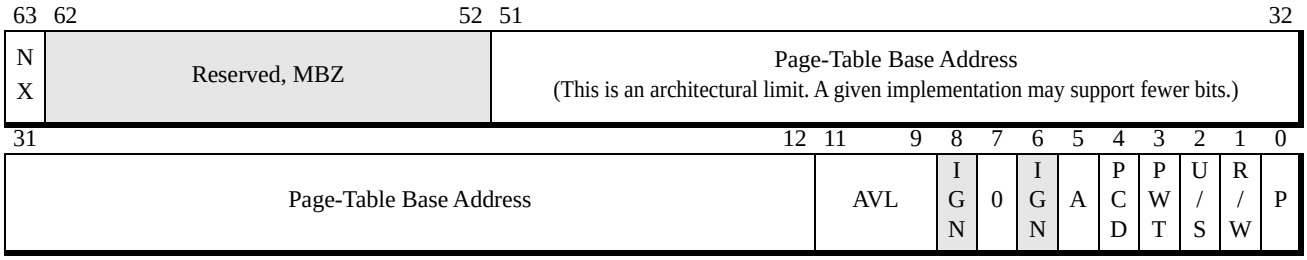


Figure 5-11. 4-Kbyte PDE—PAE Paging Legacy-Mode

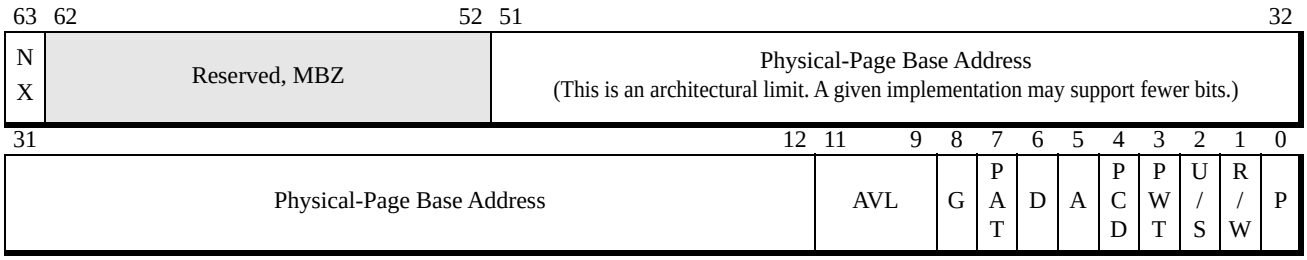


Figure 5-12. 4-Kbyte PTE—PAE Paging Legacy-Mode

2-Mbyte Page Translation. 2-Mbyte page translation is performed by dividing the 32-bit virtual address into three fields. Each field is used as an index into a 2-level page-translation hierarchy. The virtual-address fields are described as follows and are shown in Figure 5-13 on page 140:

- Bits 31:30 index into the 4-entry page-directory-pointer table.

- Bits 29:21 index into the 512-entry page-directory table.
- Bits 20:0 provide the byte offset into the physical page.

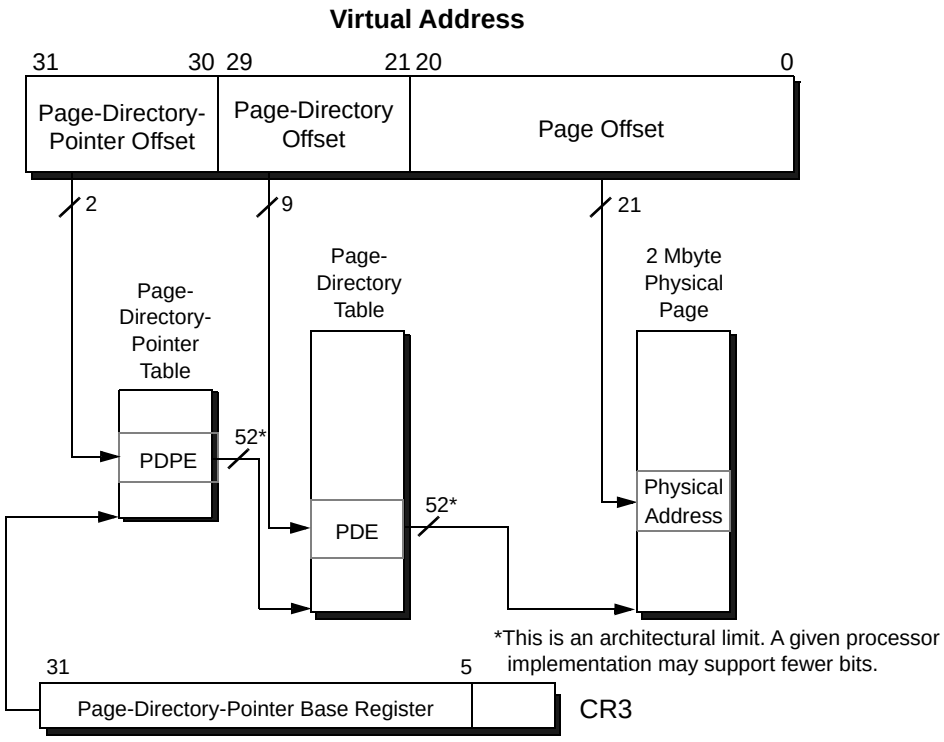


Figure 5-13. 2-Mbyte PAE Page Translation—Legacy Mode

Figure 5-14 shows the format of the PDPE (page-directory-pointer entry) and Figure 5-15 on page 141 shows the format of the PDE (page-directory entry). PTEs are not used in 2-Mbyte page translations.

Figure 5-15 on page 141 shows the PDE.PS bit set to 1 (bit 7), specifying a 2-Mbyte physical-page translation.

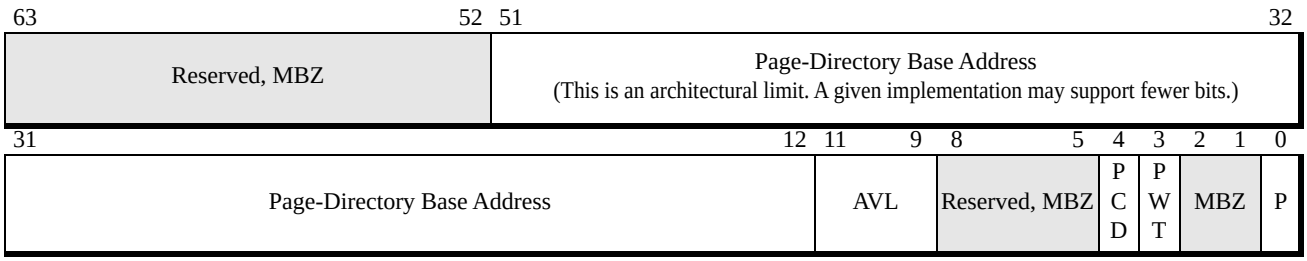


Figure 5-14. 2-Mbyte PDPE—PAE Paging Legacy-Mode

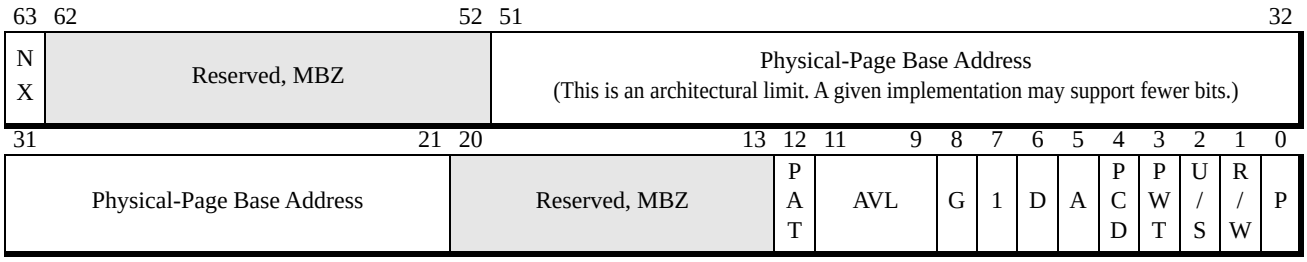


Figure 5-15. 2-Mbyte PDE—PAE Paging Legacy-Mode

5.3 Long-Mode Page Translation

Long-mode page translation requires the use of physical-address extensions (PAE). Before activating long mode, PAE must be enabled by setting CR4.PAE to 1. Activating long mode before enabling PAE causes a general-protection exception (#GP) to occur.

The PAE-paging data structures support mapping of 64-bit virtual addresses into 52-bit physical addresses. PAE expands the size of legacy page-directory entries (PDEs) and page-table entries (PTEs) from 32 bits to 64 bits, allowing physical-address sizes of greater than 32 bits.

The AMD64 architecture enhances the page-directory-pointer entry (PDPE) by defining previously reserved bits for access and protection control. A new translation table is added to PAE paging, called the page-map level-4 (PML4). The PML4 table precedes the PDP table in the page-translation hierarchy.

Because PAE is always enabled in long mode, the PS bit in the page directory entry (PDE.PS) selects between 4-Kbyte and 2-Mbyte page sizes, and the CR4.PSE bit is ignored. When 1-Gbyte pages are supported, the PDPE. PS bit selects the 1-Gbyte page size.

5.3.1 Canonical Address Form

The AMD64 architecture requires implementations supporting fewer than the full 64-bit virtual address to ensure that those addresses are in canonical form. An address is in canonical form if the address bits from the most-significant implemented bit up to bit 63 are all ones or all zeros. If the addresses of all bytes in a virtual-memory reference are not in canonical form, the processor generates a general-protection exception (#GP) or a stack fault (#SS) as appropriate.

5.3.2 CR3

In long mode, the CR3 register is used to point to the PML4 base address. CR3 is expanded to 64 bits in long mode, allowing the PML4 table to be located anywhere in the 52-bit physical-address space. Figure on page 142 shows the long-mode CR3 format.

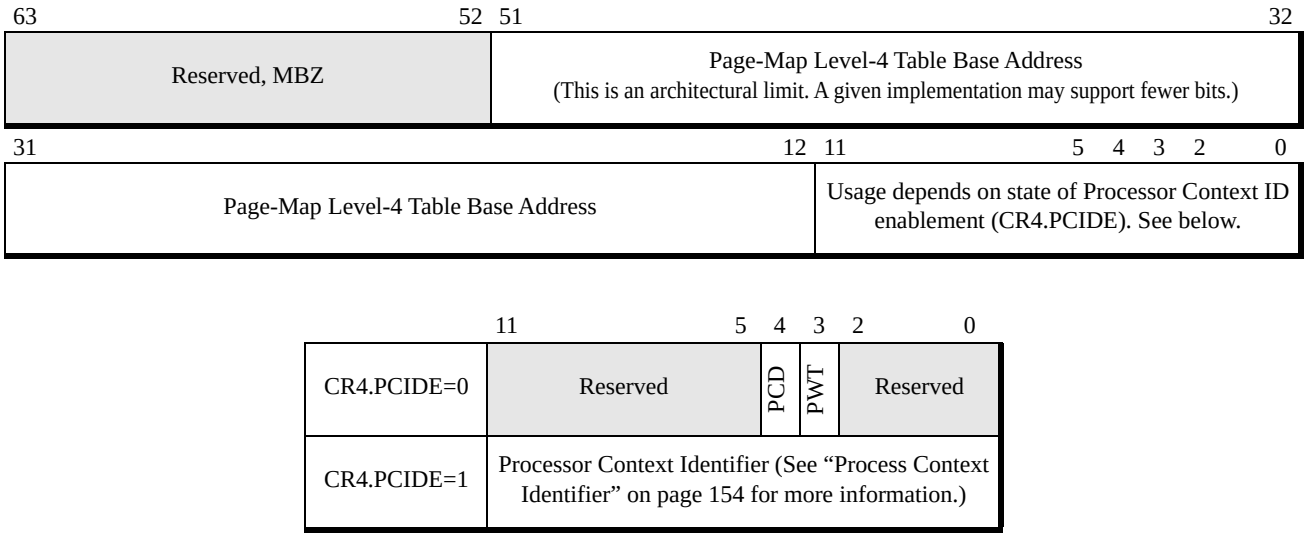


Figure 5-16. Control Register 3 (CR3)—Long Mode

The CR3 register fields for long mode are:

Table Base Address Field. Bits 51:12. This 40-bit field points to the PML4 base address. The PML4 table is aligned on a 4-Kbyte boundary with the low-order 12 address bits (11:0) assumed to be 0. This yields a total base-address size of 52 bits. System software running on processor implementations supporting less than the full 52-bit physical-address space must clear the unimplemented upper base-address bits to 0.

Page-Level Writethrough (PWT) Bit. Bit 3. Page-level writethrough indicates whether the highest-level page-translation table has a writeback or writethrough caching policy. When PWT=0, the table has a writeback caching policy. When PWT=1, the table has a writethrough caching policy.

Page-Level Cache Disable (PCD) Bit. Bit 4. Page-level cache disable indicates whether the highest-level page-translation table is cacheable. When PCD=0, the table is cacheable. When PCD=1, the table is not cacheable.

Process Context Identifier. Bits 11:0. This 12-bit field determines the current Processor Context Identifier (PCID) when CR4.PCIDE=1.

Reserved Bits. Reserved fields should be cleared to 0 by software when writing CR3.

5.3.3 4-Kbyte Page Translation

In long mode, 4-Kbyte physical-page translation is performed by dividing the virtual address into six fields. Four of the fields are used as indices into the level page-translation hierarchy. The virtual-address fields are described as follows, and are shown in Figure 5-17 on page 143:

- Bits 63:48 are a sign extension of bit 47, as required for canonical-address forms.

- Bits 47:39 index into the 512-entry page-map level-4 table.
- Bits 38:30 index into the 512-entry page-directory pointer table.
- Bits 29:21 index into the 512-entry page-directory table.
- Bits 20:12 index into the 512-entry page table.
- Bits 11:0 provide the byte offset into the physical page.

Note: *The sizes of the sign extension and the PML4 fields depend on the number of virtual address bits supported by the implementation.*

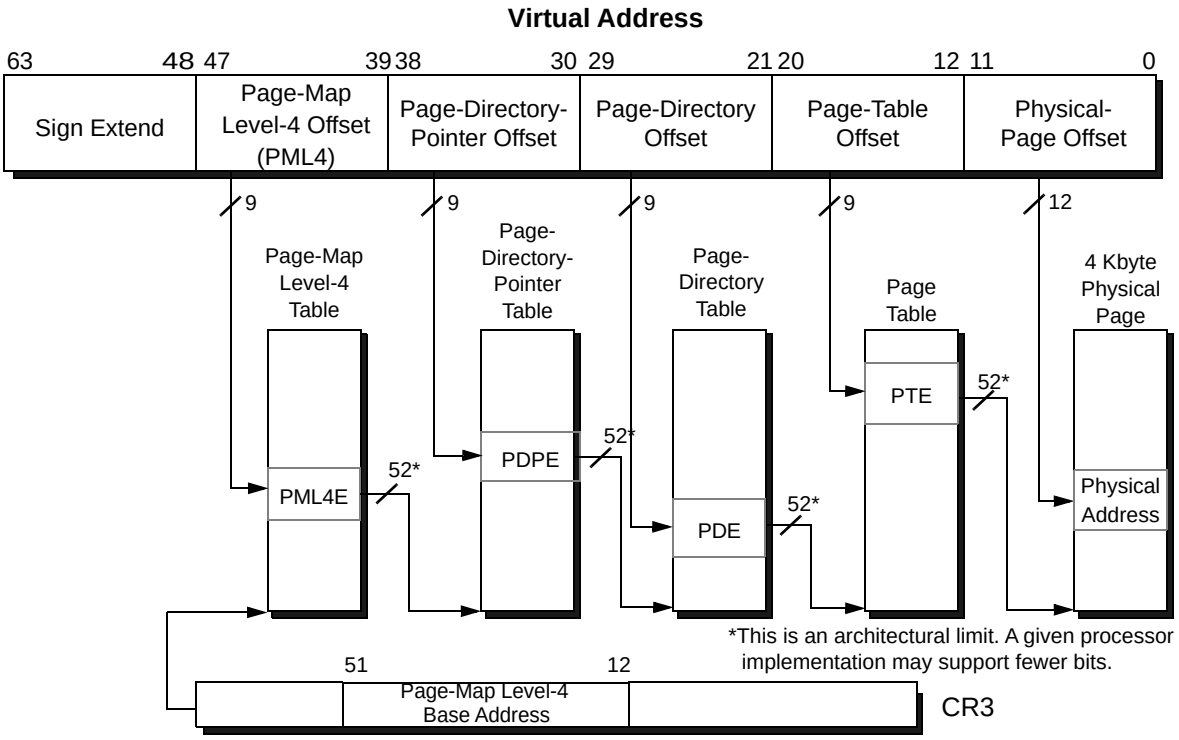


Figure 5-17. 4-Kbyte Page Translation—Long Mode

Figures 5-18 through 5-20 on page 144 and Figure 5-21 on page 145 show the long-mode 4-Kbyte translation-table formats:

- Figure 5-18 on page 144 shows the PML4E (page-map level-4 entry) format.
- Figure 5-19 on page 144 shows the PDPE (page-directory-pointer entry) format.
- Figure 5-20 on page 144 shows the PDE (page-directory entry) format.
- Figure 5-21 on page 145 shows the PTE (page-table entry) format.

The fields within these table entries are described in “Page-Translation-Table Entry Fields” on page 150.

Figure 5-20 on page 144 shows the PDE.PS bit (bit 7) cleared to 0, indicating a 4-Kbyte physical-page translation.

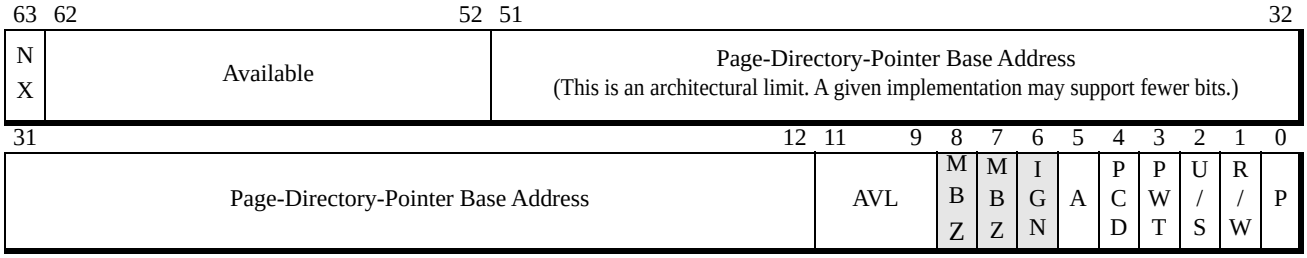


Figure 5-18. 4-Kbyte PML4E—Long Mode

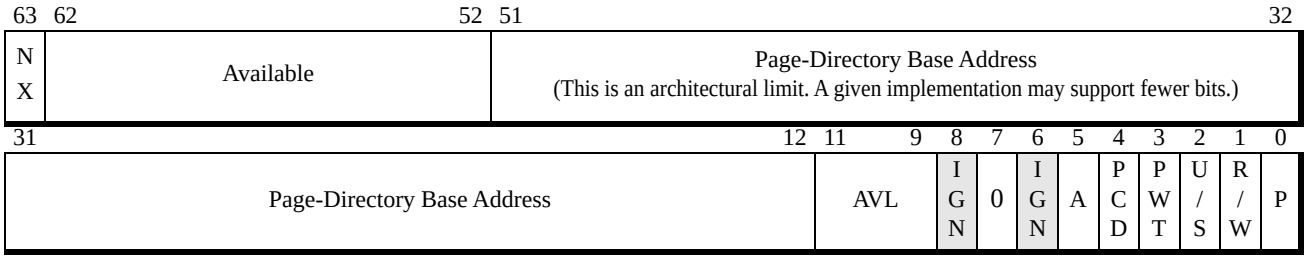


Figure 5-19. 4-Kbyte PDPE—Long Mode

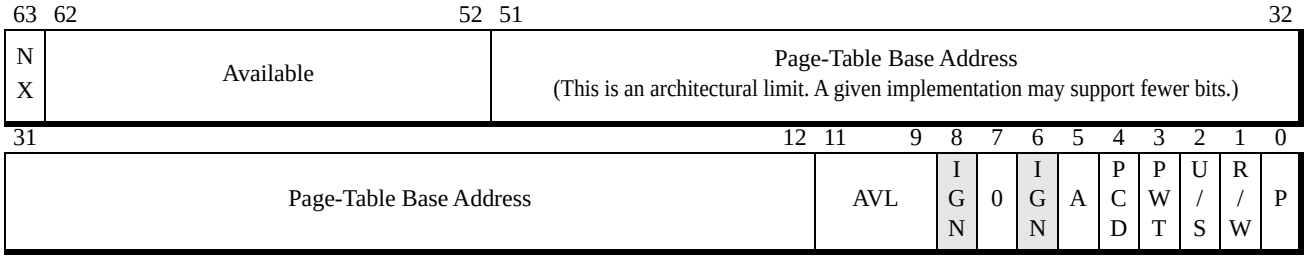


Figure 5-20. 4-Kbyte PDE—Long Mode

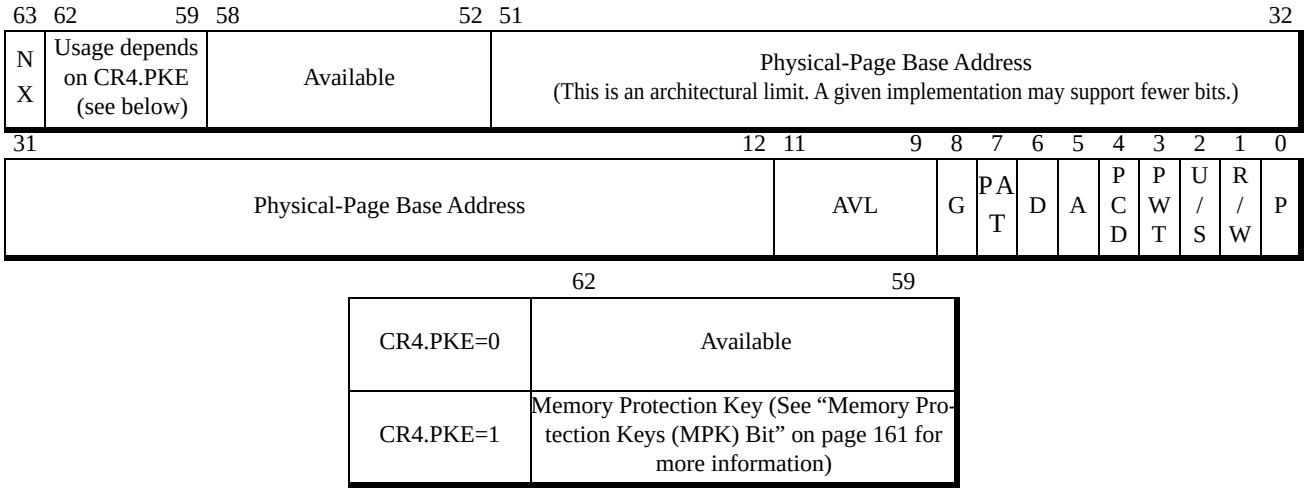


Figure 5-21. 4-Kbyte PTE—Long Mode

5.3.4 2-Mbyte Page Translation

In long mode, 2-Mbyte physical-page translation is performed by dividing the virtual address into five fields. Three of the fields are used as indices into the level page-translation hierarchy. The virtual-address fields are described as follows, and are shown in Figure 5-22:

- Bits 63:48 are a sign extension of bit 47 as required for canonical address forms.
- Bits 47:39 index into the 512-entry page-map level-4 table.
- Bits 38:30 index into the 512-entry page-directory-pointer table.
- Bits 29:21 index into the 512-entry page-directory table.
- Bits 20:0 provide the byte offset into the physical page.

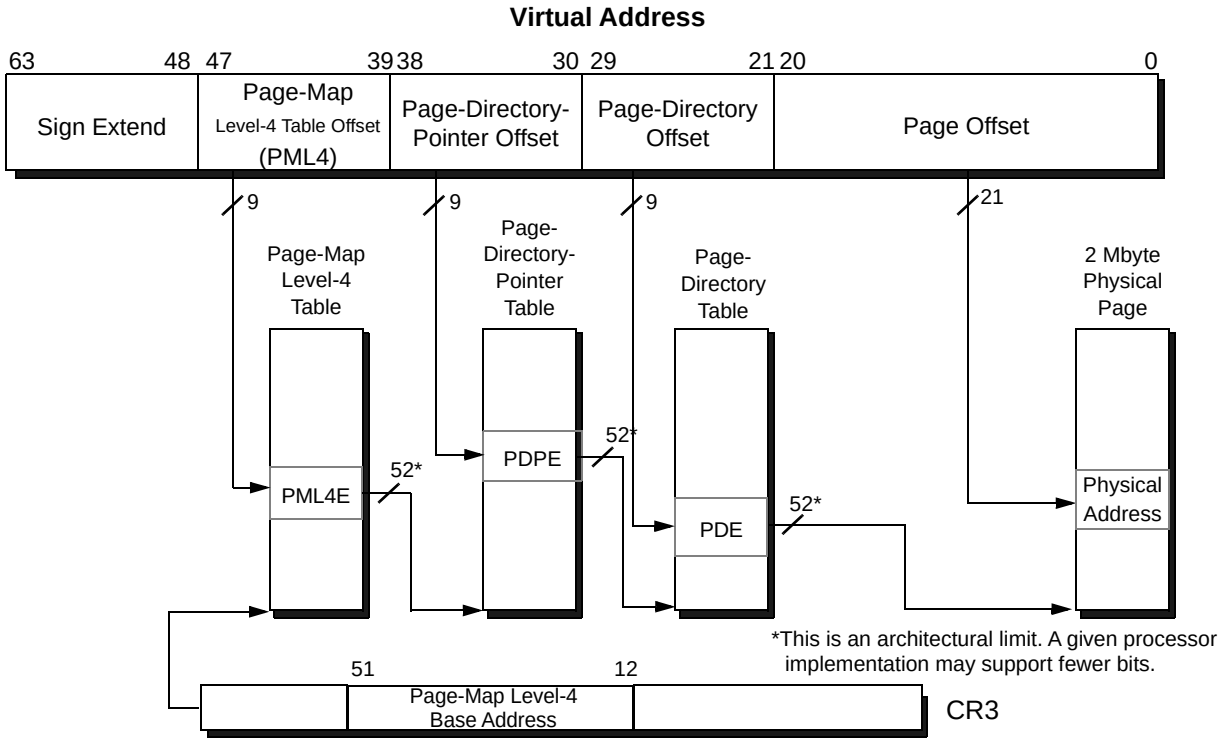


Figure 5-22. 2-Mbyte Page Translation—Long Mode

Figures 5-23 through 5-25 on page 147 show the long-mode 2-Mbyte translation-table formats (the PML4 and PDPT formats are identical to those used for 4-Kbyte page translations and are repeated here for clarity):

- Figure 5-23 on page 147 shows the PML4E (page-map level-4 entry) format.
- Figure 5-24 on page 147 shows the PDPE (page-directory-pointer entry) format.
- Figure 5-25 on page 147 shows the PDE (page-directory entry) format.

The fields within these table entries are described in “Page-Translation-Table Entry Fields” on page 150. PTEs are not used in 2-Mbyte page translations.

Figure 5-25 shows the PDE.PS bit (bit 7) set to 1, indicating a 2-Mbyte physical-page translation.

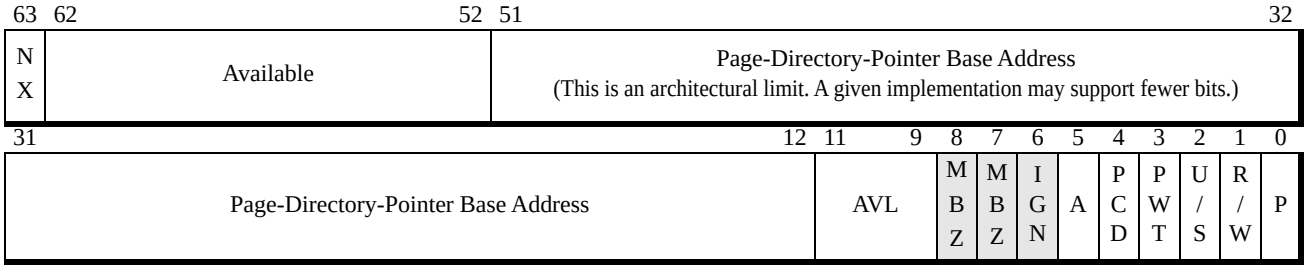


Figure 5-23. 2-Mbyte PML4E—Long Mode

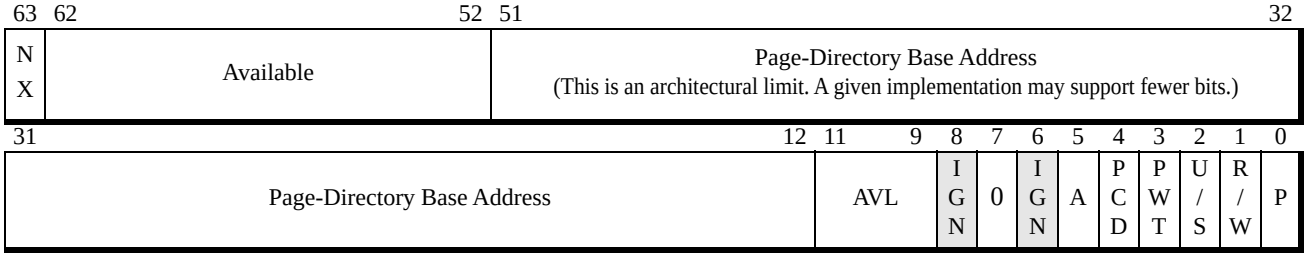


Figure 5-24. 2-Mbyte PDPE—Long Mode

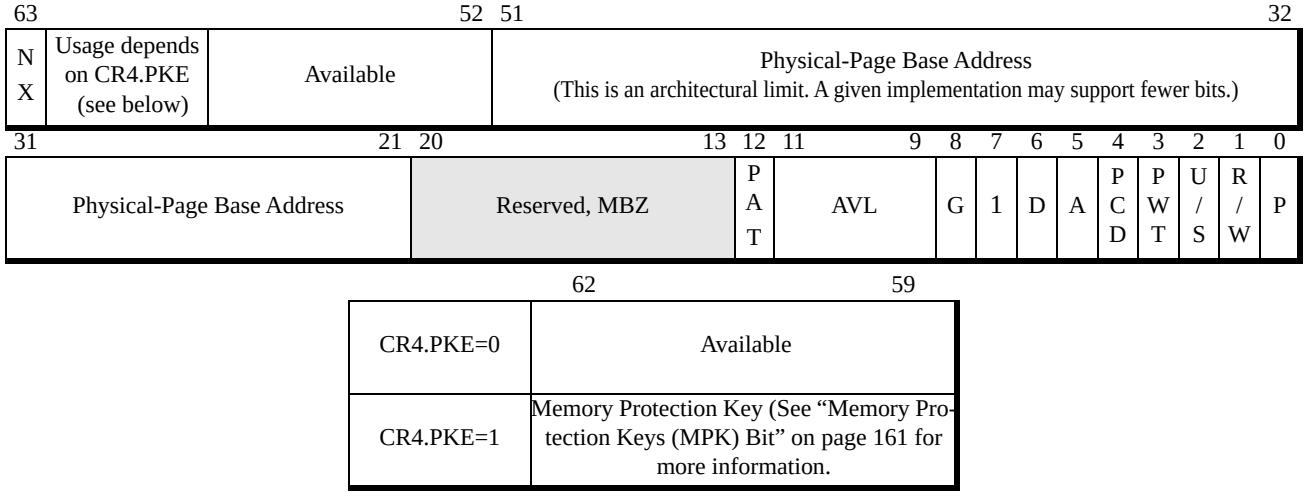


Figure 5-25. 2-Mbyte PDE—Long Mode

5.3.5 1-Gbyte Page Translation

In long mode, 1-Gbyte physical-page translation is performed by dividing the virtual address into four fields. Two of the fields are used as indices into the level page-translation hierarchy. The virtual-address fields are described as follows, and are shown in Figure 5-26 on page 148:

- Bits 63:48 are a sign extension of bit 47 as required for canonical address forms.
- Bits 47:39 index into the 512-entry page-map level-4 table.

- Bits 38:30 index into the 512-entry page-directory-pointer table.
- Bits 29:0 provide the byte offset into the physical page.

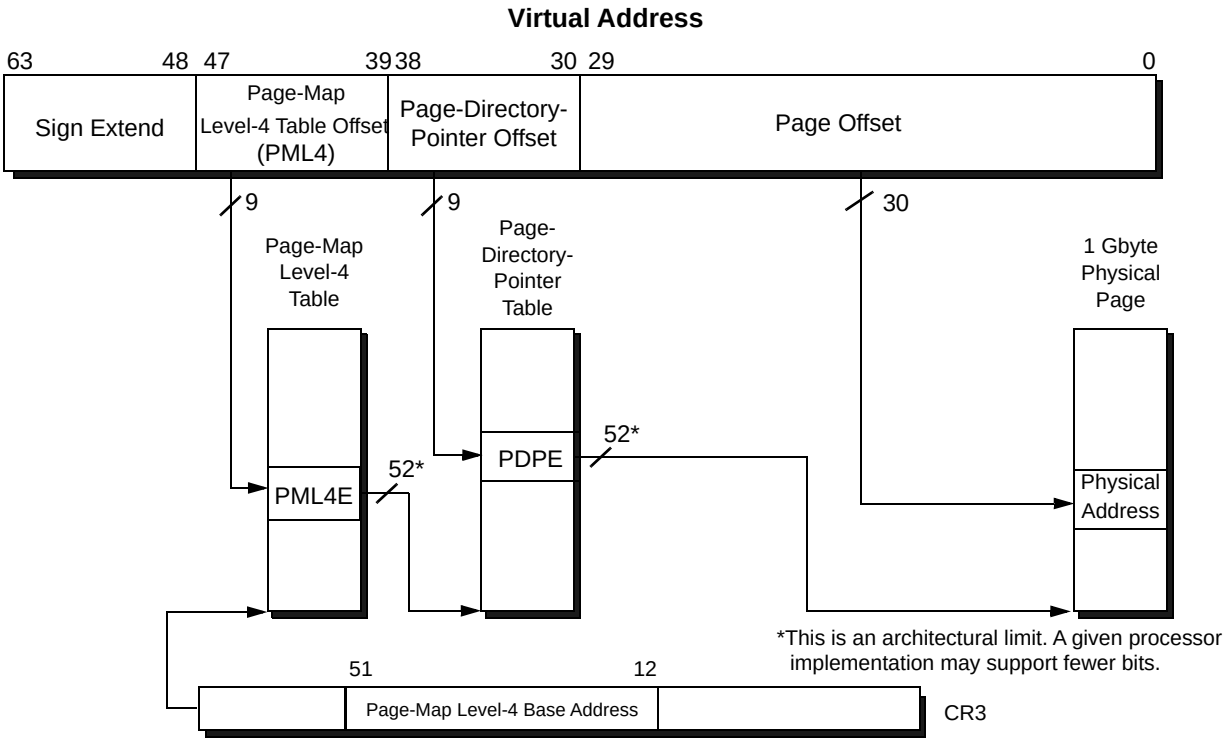


Figure 5-26. 1-Gbyte Page Translation—Long Mode

Figure 5-27 and Figure 5-28 on page 149 show the long mode 1-Gbyte translation-table formats (the PML4 format is identical to the one used for 4-Kbyte page translations and is repeated here for clarity):

- Figure 5-27 shows the PML4E (page-map level-4 entry) format.
- Figure 5-28 shows the PDPE (page-directory-pointer entry) format.

The fields within these table entries are described in “Page-Translation-Table Entry Fields” on page 150 in the current volume. PTEs and PDEs are not used in 1-Gbyte page translations.

Figure 5-28 on page 149 shows the PDPE.PS bit (bit 7) set to 1, indicating a 1-Gbyte physical-page translation.

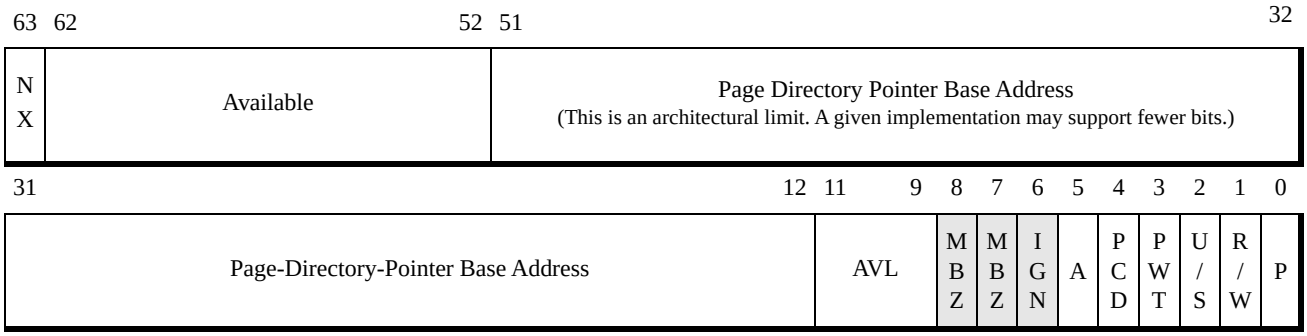


Figure 5-27. 1-Gbyte PML4E—Long Mode

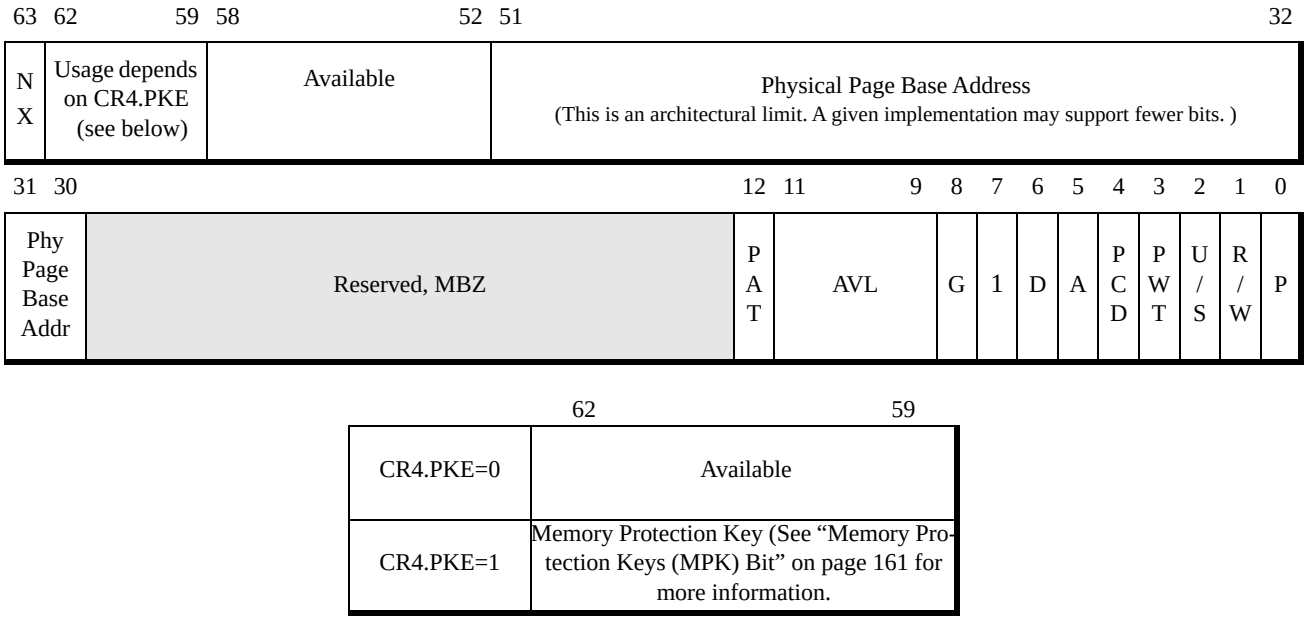


Figure 5-28. 1-Gbyte PDPE—Long Mode

1-Gbyte Paging Feature Identification. EDX bit 26 as returned by CPUID function 8000_0001h indicates 1-Gbyte page support. The EAX register as returned by CPUID function 8000_0019h reports the number of 1-Gbyte L1 TLB entries supported and EBX reports the number of 1-Gbyte L2 TLB entries. For more information using the CPUID instruction see Section 3.3 “Processor Feature Identification” on page 71.

5.4 Page-Translation-Table Entry Fields

The page-translation-table entries contain control and informational fields used in the management of the virtual-memory environment. Most fields are common across all translation table entries and modes and occupy the same bit locations. However, some fields are located in different bit positions depending on the page translation hierarchical level, and other fields have different sizes depending on which physical-page size, physical-address size, and operating mode are selected. Although these fields can differ in bit position or size, their meaning is consistent across all levels of the page translation hierarchy and in all operating modes.

5.4.1 Field Definitions

The following sections describe each field within the page-translation table entries.

Translation-Table Base Address Field. The translation-table base-address field points to the physical base address of the next-lower-level table in the page-translation hierarchy. Page data-structure tables are always aligned on 4-Kbyte boundaries, so only the address bits above bit 11 are stored in the translation-table base-address field. Bits 11:0 are assumed to be 0. The size of the field depends on the mode:

- In normal (non-PAE) paging (CR4.PAE=0), this field specifies a 32-bit physical address.
- In PAE paging (CR4.PAE=1), this field specifies a 52-bit physical address.

52 bits correspond to the maximum physical-address size allowed by the AMD64 architecture. If a processor implementation supports fewer than the full 52-bit physical address, software must clear the unimplemented high-order translation-table base-address bits to 0. For example, if a processor implementation supports a 40-bit physical-address size, software must clear bits 51:40 when writing a translation-table base-address field in a page data-structure entry.

Physical-Page Base Address Field. The physical-page base-address field points to the base address of the translated physical page. This field is found only in the lowest level of the page-translation hierarchy. The size of the field depends on the mode:

- In normal (non-PAE) paging (CR4.PAE=0), this field specifies a 32-bit base address for a physical page.
- In PAE paging (CR4.PAE=1), this field specifies a 52-bit base address for a physical page.

Physical pages can be 4 Kbytes, 2 Mbytes, 4 Mbytes, or 1-Gbyte and they are always aligned on an address boundary corresponding to the physical-page length. For example, a 2-Mbyte physical page is always aligned on a 2-Mbyte address boundary. Because of this alignment, the low-order address bits are assumed to be 0, as follows:

- 4-Kbyte pages, bits 11:0 are assumed 0.
- 2-Mbyte pages, bits 20:0 are assumed 0.
- 4-Mbyte pages, bits 21:0 are assumed 0.
- 1-Gbyte pages, bits 29:0 are assumed 0.

Present (P) Bit. Bit 0. This bit indicates whether the page-translation table or physical page is loaded in physical memory. When the P bit is cleared to 0, the table or physical page is not loaded in physical memory. When the P bit is set to 1, the table or physical page is loaded in physical memory.

Software clears this bit to 0 to indicate a page table or physical page is not loaded in physical memory. A page-fault exception (#PF) occurs if an attempt is made to access a table or page when the P bit is 0. System software is responsible for loading the missing table or page into memory and setting the P bit to 1.

When the P bit is 0, indicating a not-present page, all remaining bits in the page data-structure entry are available to software.

Entries with P cleared to 0 are never cached in TLB nor will the processor set the Accessed or Dirty bit for the table entry.

Read/Write (R/W) Bit. Bit 1. This bit controls read/write access to all physical pages mapped by the table entry. For example, a page-map level-4 R/W bit controls read/write access to all 128M ($512 \times 512 \times 512$) physical pages it maps through the lower-level translation tables. When the R/W bit is cleared to 0, access is restricted to read-only. When the R/W bit is set to 1, both read and write access is allowed. See “Page-Protection Checks” on page 158 for a description of the paging read/write protection mechanism.

User/Supervisor (U/S) Bit. Bit 2. This bit controls user (CPL 3) access to all physical pages mapped by the table entry. For example, a page-map level-4 U/S bit controls the access allowed to all 128M ($512 \times 512 \times 512$) physical pages it maps through the lower-level translation tables. When the U/S bit is cleared to 0, access is restricted to supervisor level (CPL 0, 1, 2). When the U/S bit is set to 1, both user and supervisor access is allowed. See “Page-Protection Checks” on page 158 for a description of the paging user/supervisor protection mechanism.

Page-Level Writethrough (PWT) Bit. Bit 3. This bit indicates whether the page-translation table or physical page to which this entry points has a writeback or writethrough caching policy. When the PWT bit is cleared to 0, the table or physical page has a writeback caching policy. When the PWT bit is set to 1, the table or physical page has a writethrough caching policy. See “Memory Caches” on page 199 for additional information on caching.

Page-Level Cache Disable (PCD) Bit. Bit 4. This bit indicates whether the page-translation table or physical page to which this entry points is cacheable. When the PCD bit is cleared to 0, the table or physical page is cacheable. When the PCD bit is set to 1, the table or physical page is not cacheable. See “Memory Caches” on page 199 for additional information on caching.

Accessed (A) Bit. Bit 5. This bit indicates whether the page-translation table or physical page to which this entry points has been accessed. The A bit is set to 1 by the processor the first time the table or physical page is either read from or written to. The A bit is never cleared by the processor. Instead, software must clear this bit to 0 when it needs to track the frequency of table or physical-page accesses.

Dirty (D) Bit. Bit 6. This bit is only present in the lowest level of the page-translation hierarchy. It indicates whether the physical page to which this entry points has been written. The D bit is set to 1 by

the processor the first time there is a write to the physical page. The D bit is never cleared by the processor. Instead, software must clear this bit to 0 when it needs to track the frequency of physical-page writes.

Page Size (PS) Bit. Bit 7. This bit is present in page-directory entries and long-mode page-directory-pointer entries. When the PS bit is set in the page-directory-pointer entry (PDPE) or page-directory entry (PDE), that entry is the lowest level of the page-translation hierarchy. When the PS bit is cleared to 0 in all levels above PTE, the lowest level of the page-translation hierarchy is the page-table entry (PTE), and the physical-page size is 4 Kbytes. The physical-page size is determined as follows:

- If EFER.LMA=1 and PDPE.PS=1, the physical-page size is 1 Gbyte.
- If CR4.PAE=0 and PDE.PS=1, the physical-page size is 4 Mbytes.
- If CR4.PAE=1 and PDE.PS=1, the physical-page size is 2 Mbytes.

See Table 5-1 on page 131 for a description of the relationship between the PS bit, PAE, physical-page sizes, and page-translation hierarchy.

Global Page (G) Bit. Bit 8. This bit is only present in the lowest level of the page-translation hierarchy. It indicates the physical page is a global page. The TLB entry for a global page (G=1) is not invalidated when CR3 is loaded either explicitly by a MOV CRn instruction or implicitly during a task switch. Use of the G bit requires the page-global enable bit in CR4 to be set to 1 (CR4.PGE=1). See “Global Pages” on page 155 for more information on the global-page mechanism.

Available to Software (AVL) Bit. These bits are not interpreted by the processor and are available for use by system software.

Page-Attribute Table (PAT) Bit. This bit is only present in the lowest level of the page-translation hierarchy, as follows:

- If the lowest level is a PTE (PDE.PS=0), PAT occupies bit 7.
- If the lowest level is a PDE (PDE.PS=1) or PDPE (PDPE.PS=1), PAT occupies bit 12.

The PAT bit is the high-order bit of a 3-bit index into the PAT register (Figure 7-10 on page 218). The other two bits involved in forming the index are the PCD and PWT bits. Not all processors support the PAT bit by implementing the PAT registers. See “Page-Attribute Table Mechanism” on page 218 for a description of the PAT mechanism and how it is used.

Memory Protection Key (MPK) Bits. Bits 62:59. When Memory Protection Keys are enabled (CR4.PKE=1), this 4-bit field selects the memory protection key for the physical page mapped by this entry. Ignored if memory protection keys are disabled (CR4.PKE=0). (See “Memory Protection Keys (MPK) Bit” on page 161 for a description of this mechanism.)

No Execute (NX) Bit. Bit 63. This bit is present in the translation-table entries defined for PAE paging, with the exception that the legacy-mode PDPE does not contain this bit. This bit is not supported by non-PAE paging.

The NX bit can only be set when the no-execute page-protection feature is enabled by setting EFER.NXE to 1 (see “Extended Feature Enable Register (EFER)” on page 56). If EFER.NXE=0, the NX bit is treated as reserved. In this case, a page-fault exception (#PF) occurs if the NX bit is not cleared to 0.

This bit controls the ability to execute code from all physical pages mapped by the table entry. For example, a page-map level-4 NX bit controls the ability to execute code from all 128M ($512 \times 512 \times 512$) physical pages it maps through the lower-level translation tables. When the NX bit is cleared to 0, code can be executed from the mapped physical pages. When the NX bit is set to 1, code cannot be executed from the mapped physical pages. See “No Execute (NX) Bit” on page 152 for a description of the no-execute page-protection mechanism.

Reserved Bits. Software should clear all reserved bits to 0. If the processor is in long mode, or if page-size and physical-address extensions are enabled in legacy mode, a page-fault exception (#PF) occurs if reserved bits are not cleared to 0.

5.4.2 Notes on Accessed and Dirty Bits

The processor never sets the Accessed bit or the Dirty bit for a not present page (P = 0). The ordering of Accessed and Dirty bit updates with respect to surrounding loads and stores is discussed below.

Accessed (A) Bit. The Accessed bit can be set for instructions that are speculatively executed by the processor.

For example, the Accessed bit may be set by instructions in a mispredicted branch path even though those instructions are never retired. Thus, software must not assume that the TLB entry has not been cached in the TLB, just because no instruction that accessed the page was successfully retired. Nevertheless, a table entry is never cached in the TLB without its Accessed bit being set at the same time.

The processor does not order Accessed bit updates with respect to loads done by other instructions.

Dirty (D) Bit. The Dirty bit is not updated speculatively. For instructions with multiple writes, the D bit may be set for any writes completed up to the point of a fault. In rare cases, the Dirty bit may be set even if a write was not actually performed, including MASKMOVQ with a mask of zero and certain x87 floating point instructions that cause an exception. Thus software can not assume that the page has actually been written even where PTE[D] is set to 1.

If PTE[D] is cleared to 0, software can rely on the fact that the page has not been written.

In general, Dirty bit updates are ordered with respect to other loads and stores, although not necessarily with respect to accesses to WC memory; in particular, they may not cause WC buffers to be flushed. However, to ensure compatibility with future processors, a serializing operation should be inserted before reading the D bit.

5.5 Translation-Lookaside Buffer (TLB)

When paging is enabled, every memory access has its virtual address automatically translated into a physical address using the page-translation hierarchy. *Translation-lookaside buffers* (TLBs), also known as *page-translation caches*, nearly eliminate the performance penalty associated with page translation. TLBs are special on-chip caches that hold the most-recently used virtual-to-physical address translations. Each memory reference (instruction and data) is checked by the TLB. If the translation is present in the TLB, it is immediately provided to the processor, thus avoiding external memory references for accessing page tables.

TLBs take advantage of the *principle of locality.* That is, if a memory address is referenced, it is likely that nearby memory addresses will be referenced in the near future. In the context of paging, the proximity of memory addresses required for locality can be broad—it is equal to the page size. Thus, it is possible for a large number of addresses to be translated by a small number of page translations. This high degree of locality means that almost all translations are performed using the on-chip TLBs.

System software is responsible for managing the TLBs when updates are made to the linear-to-physical mapping of addresses. A change to any paging data-structure entry is not automatically reflected in the TLB, and hardware snooping of TLBs during memory-reference cycles is not performed. Software must invalidate the TLB entry of a modified translation-table entry so that the change is reflected in subsequent address translations. TLB invalidation is described in “TLB Management” on page 155. Only privileged software running at CPL=0 can manage the TLBs.

5.5.1 Process Context Identifier

The Process Context Identifier (PCID) feature allows a logical processor to cache TLB mappings concurrently for multiple virtual address spaces. When enabled (by setting CR4.PCIDE=1), the processor associates the current 12-bit PCID with each TLB mapping it creates. Only entries matching the current PCID are used when performing address translations. In this way, the processor may retain cached TLB mappings for multiple contexts.

The current PCID is the value in CR3[11:0]. When PCIDs are enabled the system software can store 12-bit Process Context Identifiers in CR3 for different address spaces. Subsequently, when system software switches address spaces (by writing the page table base pointer in CR3[62:12]), the processor may use TLB mappings previously stored for that address space and PCID, providing that bit 63 of the source operand is set to 1. If bit 63 is set to 0, the legacy behavior of a move to CR3 is maintained, invalidating TLB entries but only non-global entries for the specified PCID. Note that this bit is not stored in the CR3 register itself. PCID-tagged TLB contents may also be managed using the INVPCID instruction; see the INVPCID description in volume 3 for details. A MOV to CR4 that clears CR4.PCIDE causes all cached entries in the TLB for the logical processor to be invalidated. When PCIDs are not enabled (CR4.PCIDE=0) the current PCID is always zero and all TLB mappings are associated with PCID=0.

Attempting to set CR4.PCIDE with a MOV to CR4 if EFER.LMA = 0 or CR3[11:0] <> 0 causes a #GP exception. Attempting to clear CR0.PG with a MOV to CR0 if CR4.PCIDE is set also causes a

#GP exception. The presence of PCID functionality is indicated by CPUID Function 1, ECX[PCID]=1.

5.5.2 Global Pages

The processor invalidates the TLB whenever CR3 is loaded either explicitly or implicitly. After the TLB is invalidated, subsequent address references can consume many clock cycles until their translations are cached as new entries in the TLB. Invalidation of TLB entries for frequently-used or critical pages can be avoided by specifying the translations for those pages as *global*. TLB entries for global pages are not invalidated as a result of a CR3 load. Global pages are invalidated using the INVLPG instruction.

Global-page extensions are controlled by setting and clearing the PGE bit in CR4 (bit 7). When CR4.PGE is set to 1, global-page extensions are enabled. When CR4.PGE is cleared to 0, global-page extensions are disabled. When CR4.PGE=1, setting the global (G) bit in the translation-table entry marks the page as global.

The INVLPG instruction ignores the G bit and can be used to invalidate individual global-page entries in the TLB. To invalidate all entries, including global-page entries, disable global-page extensions (CR4.PGE=0).

5.5.3 TLB Management

Generally, unless system software modifies the linear-to-physical address mapping, the processor manages the TLB transparently to software. This includes allocating entries and replacing old entries with new entries. In general, software changes made to paging-data structures are not automatically reflected in the TLB. In these situations, it is necessary for software to invalidate TLB entries so that these changes will be propagated to the page-translation mechanism.

TLB entries can be explicitly invalidated using operations intended for that purpose or implicitly invalidated as a result of another operation. TLB invalidation has no effect on the associated page-translation tables in memory.

Explicit Invalidations. Several mechanisms are provided to explicitly invalidate the TLB:

- The *Invalidate TLB Entry* instruction (INVLPG) can be used to invalidate a specific entry within the TLB. This instruction invalidates an entry regardless of whether it is marked as global or not.
- The *Invalidate TLB entry in a Specified ASID* instruction (INVLPGA) operates similarly, but operates only on entries associated with the specified ASID. See “Invalidate Page, Alternate ASID” on page 514.
- The *Invalidate TLB with Broadcast* instruction (INVLPGB) can be used to invalidate a specified range of TLB entries on the local processor and broadcast the invalidation operation to remote processors. See INVLPGB in Volume 3.
- The *Invalidate TLB entries in Specified PCID* instruction (INVPCID) can be used to invalidate TLB entries of the specified Processor Context ID. See INVPCID in Volume 3.

- Updates to the CR3 register cause the entire TLB to be invalidated *except* for global pages. The CR3 register can be updated with the MOV CR3 instruction. CR3 is also updated during a task switch, with the updated CR3 value read from the TSS of the new task. This behavior for CR3 is modified by the PCID extension; See “Process Context Identifier” on page 154 for details.
- The TLB_CONTROL field of a VMCB can request specific flushes of the TLB to occur when the VMRUN instruction is executed on that VMCB. See “TLB Flush” on page 513.

Implicit Invalidations. The following operations cause the entire TLB to be invalidated, including global pages:

- Modifying the CR0.PG bit (paging enable).
- Modifying the CR4.PAE bit (physical-address extensions), the CR4.PSE bit (page-size extensions), or the CR4.PGE bit (page-global enable).
- Entering SMM as a result of an SMI interrupt.
- Executing the RSM instruction to return from SMM.
- Updating a memory-type range register (MTRR) with the WRMSR instruction.
- External initialization of the processor.
- External masking of the A20 address bit (asserting the A20M# input signal).
- Writes to certain model-specific registers with the WRMSR instruction; see the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product for more information.
- A MOV to CR4 that changes CR4.PKE from 0 to 1.
- A MOV to CR4 that clears CR4.PCIDE from 1 to 0.

Invalidation of Table Entry Upgrades. If a table entry is updated to remove a page access constraint, such as removing supervisor, read-only, and/or no-execute restrictions, an invalidation is not required because the hardware will automatically detect the changes. If a table entry is updated and does not remove a permission violation, it is unpredictable whether the old or updated entry will be used until an invalidation is performed.

Speculative Caching of Address Translations. For performance reasons, AMD64 processors may speculatively load valid address translations into the TLB on false execution paths. Such translations are not based on references that a program makes from an “architectural state” perspective, but which the processor may make in speculatively following an instruction path which turns out to be mispredicted. In general, the processor may create a TLB entry for any linear address for which valid entries exist in the page table structure currently pointed to by CR3. This may occur for both instruction fetches and data references. Such entries remain cached in the TLBs and may be used in subsequent translations. Loading a translation speculatively will set the Accessed bit, if not already set. A translation will *not* be loaded speculatively if the Dirty bit needs to be set.

Caching of Upper Level Translation Table Entries. Similarly, to improve the performance of table walks on TLB misses, AMD64 processors may save upper level translation table entries in special table walk caching structures which are kept coherent with the tables in memory via the same

mechanisms as the TLBs—by means of the INVLPG instruction, moves to CR3, and modification of paging control bits in CR0 and CR4. Like address translations in the TLB, these upper level entries may also be cached speculatively and by false-path execution. These entries are never cached if their P (present) bits are set to 0.

Under certain circumstances, an upper-level table entry that cannot ultimately lead to a valid translation (because there are no valid entries in the lower level table to which it points) may also be cached. This can happen while executing down a false path, when an in-progress table walk gets cancelled by the branch mispredict before the low level table entry that would cause a fault is encountered. Said another way, the fact that a page table has no valid entries does not guarantee that upper level table entries won't be accessed and cached in the processor, as long as those upper level entries are marked as present. For this reason, it is not safe to modify an upper level entry, even if no valid lower-level entries exist, without first clearing its present bit, followed by an INVLPG instruction.

Use of Cached Entries When Reporting a Page Fault Exception. On current AMD64 processors, when any type of page fault exception is encountered by the MMU, any cached upper-level entries that lead to the faulting entry are flushed (along with the TLB entry, if already cached) and the table walk is repeated to confirm the page fault using the table entries in memory. This is done because a table entry is allowed to be upgraded (by marking it as present, or by removing its write, execute or supervisor restrictions) without explicitly maintaining TLB coherency. Such an upgrade will be found when the table is re-walked, which resolves the fault. If the fault is confirmed on the re-walk however, a page fault exception is reported, and upper level entries that may have been cached on the re-walk are flushed.

Handling of D-Bit Updates. When the processor needs to set the D bit in the PTE for a TLB entry that is already marked as writable at all cached TLB levels, the table walk that is performed to access the PTE in memory may use cached upper level table entries. This differs from the fault situation previously described, in which cached entries aren’t used to confirm the fault during the table walk.

Invalidation of Cached Upper-level Entries by INVLPG. The effect of INVLPG on TLB caching of upper-level page table entries is controlled by EFER[TCE] on processors that support the translation cache extension feature. If EFER[TCE] is 0, or if the processor does not support the translation cache extension feature, an INVLPG will flush all upper-level page table entries in the TLB as well as the target PTE. If EFER[TCE] is 1, INVLPG will flush only those upper-level entries that lead to the target PTE, along with the target PTE itself. INVLPGA may flush all upper-level entries regardless of the state of TCE. For further details, see Section 3.1.7 “Extended Feature Enable Register (EFER)” on page 56.

Handling of PDPT Entries in PAE Mode. When 32-bit PAE mode is enabled on AMD64 processors (CR4.PAE is set to 1) a third level of the address translation table hierarchy, the page directory pointer table (PDPT), is enabled. This table contains four entries. On older AMD64 processors, in native mode, these four entries are unconditionally loaded into the table walk cache whenever CR3 is written with the PDPT base address, and remain locked in. At this point they are also checked for reserved bit violations, and if such violations are present a general-protection exception (#GP) occurs.

Under SVM, however, when the processor is in guest mode with PAE enabled, the guest PDPT entries are not cached or validated at this point, but instead are loaded and checked on demand in the normal course of address translation, just like page directory and page table entries. Any reserved bit violations are detected at the point of use, and result in a page-fault (#PF) exception rather than a general-protection (#GP) exception. The cached PDPT entries are subject to displacement from the table walk cache and reloading from the PDPT, hence software must assume that the PDPT entries may be read by the processor at any point while those tables are active. Modern AMD processors implement this same behavior in native mode as well, rather than pre-loading the PDPT entries.

5.6 Page-Protection Checks

The AMD64 architecture provides the following forms of page-level memory protection:

- Supervisor pages. This form of protection prevents non-privileged (user) code from accessing privileged (supervisor) code and data.
- Read-only pages. This form of protection prevents writes into read-only address spaces.
- Instruction fetch restrictions. Two forms of page-level memory protection prevent the processor from fetching instructions from pages that are either known to contain non-executable data or that are accessible by user-mode code.
- Memory protection keys. This form of protection allows an application to manage page-based data access protections from user mode.
- Shadow stack pages. The processor restricts the types of memory accesses that are allowed to read or write a shadow stack page and prohibits the shadow stack mechanism from accessing non-shadow stack pages. See “Shadow Stack Protection” on page 161.

Access protection checks are performed when a virtual address is translated into a physical address. For those checks, the processor examines the page-level memory-protection bits in the translation tables to determine if the access is allowed. The page table bits involved in these checks are:

- User/Supervisor (U/S)—See “User/Supervisor (U/S) Bit” on page 151.
- Read/Write (R/W)—See “Read/Write (R/W) Bit” on page 151.
- No-Execute (NX)—See “No Execute (NX) Bit” on page 152.
- Memory Protection Key (MPK)—See “Memory Protection Keys (MPK) Bit” on page 161.

Access protection actions taken by the processor are controlled by the following bits:

- Write-Protect enable (CR0.WP)—See “Write Protect (WP) Bit” on page 44.
- No-Execute Enable (EFER.NXE)—See “No-Execute Enable (NXE) Bit” on page 58.
- Supervisor-mode Execution Prevention enable (CR4.SMEP)—See “Supervisor Mode Execution Prevention (SMEP)” on page 51.
- Supervisor-mode Access Prevention enable (CR4.SMAP)—See “Supervisor-Mode Access Prevention(CR4.SMAP) Bit” on page 160.
- Alignment Check Bit (RFLAGS.AC) – See ““Alignment Check (AC) Bit” on page 55.

- Protection Key Enable (CR4.PKE)—See “Protected-Mode Enable (PE) Bit” on page 43.
- Control-flow Enforcement Technology (CR4.CET) - See “CR4 Register” on page 47.

These protection checks are available at all levels of the page-translation hierarchy.

5.6.1 User/Supervisor (U/S) Bit

The U/S bit in the page-translation tables determines the privilege level required to access the page. If U/S=0 in any of the page table entries traversed during a table walk, the page is considered a supervisor page. If U/S=1 in all the page table entries traversed during a table walk, the page is considered a user page. Conceptually, user (non-privileged) pages correspond to a current privilege-level (CPL) of 3, or least-privileged. Supervisor (privileged) pages correspond to a CPL of 0, 1, or 2, all of which are jointly regarded as most-privileged.

When the processor is running at a CPL of 0, 1, or 2, it can access both user and supervisor pages unless restricted by SMEP or SMAP (See sections “Supervisor-Mode Execution Prevention (CR4.SMEP) Bit” on page 160 and “Supervisor-Mode Access Prevention(CR4.SMAP) Bit” on page 160 for further details). However, when the processor is running at a CPL of 3, it can only access user pages. If an attempt is made to access a supervisor page while the processor is running at CPL = 3, a page-fault exception (#PF) occurs.

See “Privilege-Level Concept” on page 105 for more information on processor privilege levels.

5.6.2 Read/Write (R/W) Bit

The R/W bit in the page-translation tables specifies the access type allowed for the page. If R/W=0 in any of the page table entries traversed during a table walk, the page is read-only. If R/W=1 in all the page table entries traversed during a table walk, the page is read/write. A page-fault exception (#PF) occurs if an attempt is made by user software to write to a read-only page. If supervisor software attempts to write a read-only page, the outcome depends on the value of the CR0.WP bit (described below).

5.6.3 No Execute (NX) Bit

The NX bit provides the ability to mark a page as non-executable. If the NX bit is set at any level of the page-table hierarchy in the table entries traversed during a table walk, the page mapped by those entries is a no-execute page. When no-execute protection is enabled, any attempt to fetch an instruction from a no-execute page results in a page-fault exception (#PF).

The no-execute protection check applies to all privilege levels. It does not distinguish between supervisor and user-level accesses.

The no-execute protection feature is supported only in PAE-paging mode. In 32-bit PAE mode, the NX bit is not supported at the Page Directory Pointer table level. In this mode, the value of the NX bit at the PDP level defaults to 0.

No-execute protection is enabled by setting the NXE bit in the EFER register to 1. Before setting this bit, system software must verify the processor supports the no-execute feature by checking the CPUID NX feature flag (CPUID Fn8000_0001_EDX[NX]).

5.6.4 Write Protect (CR0.WP) Bit

The ability to write to read-only pages is governed by the processor mode and whether write protection is enabled. If write protection is *not* enabled, a processor running at CPL 0, 1, or 2 can write to any physical page, even if it is marked as read-only. Enabling write protection by setting the WP bit in CR0 prevents supervisor code from writing into read-only pages, including read-only user-level pages.

A page-fault exception (#PF) occurs if software attempts to write (at any privilege level) into a read-only page while write protection is enabled.

5.6.5 Supervisor-Mode Execution Prevention (CR4.SMEP) Bit

When supported and enabled, a page-fault exception (#PF) is generated if the processor attempts to fetch an instruction from a user page while running at CPL 0, 1, or 2.

Supervisor-mode execution prevention is enabled by setting the SMEP bit (bit 20) in the CR4 register to 1. Before setting this bit, system software must verify the processor supports the SMEP feature by checking the SMEP feature flag (CPUID Fn0000_0007_EBX[SMEP]_x0 = 1).

For more information using the CPUID instruction see Section 3.3 “Processor Feature Identification” on page 71.

5.6.6 Supervisor-Mode Access Prevention(CR4.SMAP) Bit

When SMAP is supported and enabled, a page-fault exception (#PF) is generated if the processor attempts to read or write data from a user page and one of the following is true:

- The access is an implicit supervisor-mode access OR
- The access is made while running at CPL 0,1, or 2 and RFLAGS.AC=0.

Some accesses are considered implicit supervisor-mode accesses. Implicit supervisor-mode accesses are subject to the SMAP check regardless of the value of RFLAGS.AC. An implicit supervisor-mode access is one that is considered a supervisor access regardless of the value of CPL.

Supervisor-mode access prevention is enabled by setting the SMAP bit (bit 21) in the CR4 register to 1. Before setting this bit, system software must verify the processor supports the SMAP feature by checking the SMAP feature flag CPUID Fn0000_0007_EBX[SMAP]_x0 (bit 20) = 1.

Shadow stack accesses are not subject to the SMAP check.

For more information on using the CPUID instruction see “Processor Feature Identification” on page 71.

5.6.7 Memory Protection Keys (MPK) Bit

The Memory Protection Key (MPK) feature provides a way for applications to impose page-based data access protections (read/write, read-only or no access), without requiring modification of page tables and subsequent TLB invalidations when the application changes protection domains.

When MPK is enabled (CR4.PKE=1), a protection key is located in bits 62:59 of final page table entry mapping each virtual address. This 4-bit protection key is used as an index (i) into the user-accessible PKRU register which contains 16 access-disable/write-disable (WDi/ADi) pairs.

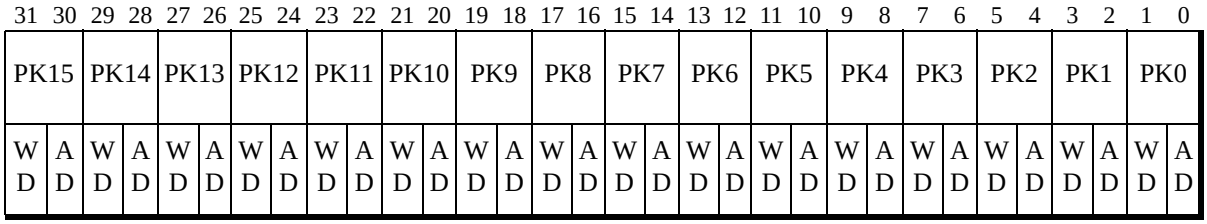


Figure 5-29. PKRU Register

The WDi/ADi pairs operate as follows:

If ADi=0, data access is permitted

- If ADi=1, no data access is permitted (regardless of CPL)
- If WDi == 0, write access is allowed
- If WDi == 1: User-mode write access is not allowed. Supervisor access is controlled by CR0.WP:
  - If CR0.WP=1, supervisor-mode writes are not allowed
  - If CR0.WP=0, supervisor-mode writes are allowed

Software can use the RDPKRU and WRPKRU instructions to read and write the PKRU register. These instructions are not privileged and can be used in user mode or in supervisor mode.

The MPK mechanism is ignored in the following cases:

- if CR4.PKE=0
- if long mode is disabled (EFER.LMA=0)
- for instruction fetches
- for pages marked in the paging structures as supervisor addresses (U/S=0)

5.7 Shadow Stack Protection

When the shadow stack feature is enabled (CR4.CET=1), certain combinations of page-table protection bits are used to distinguish pages containing shadow stacks from ordinary pages. As described in the following sections, the processor restricts the types of memory accesses that can be

made to shadow stack pages and prohibits the shadow stack mechanism from accessing non-shadow stack pages. (See “Shadow Stacks” on page 647 for details on the shadow stack feature).

5.7.1 Shadow Stack Accesses

The processor treats certain memory accesses as shadow stack accesses. Shadow stack accesses are generated only by the shadow stack instructions or by the shadow stack mechanism. As with ordinary data accesses, a shadow stack access can be either a supervisor access or a user access, depending on the CPL when the access is made. Shadow stack accesses made when the processor is at CPL 0, 1, or 2 are supervisor-shadow stack accesses, and accesses made at CPL 3 are user-shadow stack accesses. (An exception is the WRUSS instruction, whose accesses are always treated as user-shadow stack accesses).

5.7.2 Shadow Stack Pages

Shadow stack accesses are allowed only to linear addresses that are mapped to shadow stack pages. A shadow stack is described by the following combination of page-table protection bits:

- R/W(Read/Write)=0 and D(Dirty)=1 in the final page-table entry that maps the linear address.
- R/W(Read/Write)=1 in all other page-mapping structures leading to the final page-table entry.

The U/S (User/Supervisor) bit in the page-translation tables determines the privilege level required to access a given shadow stack page. If U/S=0, the page is considered a supervisor-shadow stack page and if U/S=1 the page is considered a user-shadow stack page.

5.7.3 Shadow Stack Protection Checks

The processor restricts the types of memory accesses that are allowed to read or write a shadow stack page. The page-level protection bits and the type of memory access are examined to determine if the access is allowed. The following section assumes the memory protection key field allows access to the given page, if memory protection keys are enabled, and that CR0.WP=1 (which is prerequisite for enabling the shadow stack feature).

The following memory accesses are allowed to shadow stack pages:

- User-shadow stack accesses can read or write user-shadow stack pages,
- Supervisor-shadow stack accesses can read or write supervisor-shadow stack pages.

*(**Note**: shadow stack write accesses are allowed to complete, even though the R/W bit is 0).*

- Non-shadow stack reads can read any shadow stack page (subject to U/S page protections).

The following memory accesses are not allowed:

- User-shadow stack access to supervisor-shadow stack pages.
- Supervisor-shadow stack access to user-shadow stack pages.
- Any shadow stack access to a non-shadow stack page.
- Non-shadow stack writes to a shadow stack page.

If the memory access is not allowed, a page-fault exception (#PF) is generated with the paging-protection violation bits (user/supervisor, read/write, or both) set in the error code as appropriate. The SS bit is set in the #PF error code if the page-fault was caused by a shadow stack access. (See “Page-Fault Error Code” on page 254).

5.8 Protection Across Paging Hierarchy

The privilege level and access type specified at each level of the page-translation hierarchy have a combined effect on the protection of the translated physical page. Enabling and disabling write protection via CR0.WP further qualifies the protection effect on the physical page.

Table 5-2 shows the overall effect that privilege level and access type have on physical-page protection when write protection is disabled (CR0.WP=0). In this case, when *any* translation-table entry is specified as supervisor level, the physical page is a supervisor page and can only be accessed by software running at CPL 0, 1, or 2. Such a page allows read/write access even if all levels of the page-translation hierarchy specify read-only access.

Table 5-2. Physical-Page Protection, CR0.WP=0

Page-Map Level-4 Entry		Page-Directory-Pointer Entry		Page-Directory Entry		Page-Table Entry		Effective Result on Physical Page	
U/S	R/W	U/S	R/W	U/S	R/W	U/S	R/W	U/S	R/W
S	—	—	—	—	—	—	—	S	R/W
—	—	S	—	—	—	—	—		
—	—	—	—	S	—	—	—		
—	—	—	—	—	—	S	—		
U	R	U	—	U	—	U	—	U	R[1]
U	—	U	R	U	—	U	—		
U	—	U	—	U	R	U	—		
U	—	U	—	U	—	U	R		
U	R/W	U	R/W	U	R/W	U	R/W	U	R/W

Note:
S = Supervisor Level (CPL=0, 1, or 2), U = User Level (CPL = 3), R = Read-Only Access, R/W = Read/Write Access, — = Don’t Care.

Note:
1. *Supervisor-level programs can access these pages as R/W.*

If *all* table entries in the translation hierarchy are specified as user level the physical page is a user page, and both supervisor and user software can access it. In this case the physical page is read-only if any table entry in the translation hierarchy specifies read-only access. All table entries in the translation hierarchy must specify read/write access for the physical page to be read/write.

Table 5-3 shows the overall effect that privilege level and access type have on physical-page access when write protection is enabled (CR0.WP=1). When any translation-table entry is specified as

supervisor level, the physical page is a supervisor page and can only be accessed by supervisor software. In this case, the physical page is read-only if any table entry in the translation hierarchy specifies read-only access. All table entries in the translation hierarchy must specify read/write access for the supervisor page to be read/write.

Table 5-3. Effect of CR0.WP=1 on Supervisor Page Access

Page-Map Level-4 Entry	Page Directory-Pointer Entry	Page Directory Entry	Page Table Entry	Physical Page
R/W	R/W	R/W	R/W	R/W
R	—	—	—	R
—	R	—	—	
—	—	R	—	
—	—	—	R	
W	W	W	W	W

Note:
R = Read-Only Access Type, W = Read/Write Access Type, — = Don't Care.
Physical page is in supervisor mode, as determined by U/S settings in Table 5-2.

5.8.1 Access to User Pages when CR0.WP=1

As shown in Table 5-2 on page 163, read/write access to user-level pages behaves the same as when write protection is disabled (CR0.WP=0), with one critical difference. When write protection is enabled, supervisor programs cannot write into read-only user pages.

5.9 Effects of Segment Protection

Segment-protection and page-protection checks are performed serially by the processor, with segment-privilege checks performed first, followed by page-protection checks. Page-protection checks are not performed if a segment-protection violation is found. If a violation is found during either segment-protection or page-protection checking, an exception occurs and no memory access is performed. Segment-protection violations cause either a general-protection exception (#GP) or a stack exception (#SS) to occur. Page-protection violations cause a page-fault exception (#PF) to occur.

5.10 Upper Address Ignore

The Upper Address Ignore feature provides a way for software to use bits [63:57] of an address as an arbitrary software-assigned and software-interpreted tag. When this feature is enabled, these address bits are excluded from the canonicality check that's done when the address is used for certain memory references.

5.10.1 Detecting and Enabling Upper Address Ignore

Support for the Upper Address Ignore feature is indicated by CPUID Fn8000_0021_EAX[UpperAddressIgnore](bit 7)=1. The Upper Address Ignore feature is enabled by setting EFER.UAIE (bit 20) in 64-bit mode (EFER.LMA=CS.L=1). EFER.UAIE is ignored outside of 64-bit mode.

5.10.2 Upper Address Ignore Operation

When Upper Address Ignore is active the processor no longer performs a canonical address check on bits [63:57] of the logical address for memory references that use either the DS or ES segment, it checks only bits [56:48] of the logical address.

Any memory reference made using the segment registers CS, SS, FS or GS still performs the normal canonical address check. This includes indirect jump, call, and return target addresses since they are CS based. Canonical checks for implicit references to IDT, GDT, LDT and TSS are not suppressed when Upper Address Ignore is active.

5.10.3 Address Tag Storage

The following registers, which hold virtual addresses for communication to software, are not guaranteed to hold the address tag in the upper 7 bits:

- CR2
- x87 DP
- IBS Data Cache Linear Address Register

5.10.4 Debug Breakpoint behavior with Upper Address Ignore

The Debug Breakpoint Address Registers (DR0-3) hold full 64-bit addresses and Upper Address Ignore changes how the address match is performed for data address breakpoints. When Upper Address Ignore is active the value in DR0-3[63:57] is ignored and the linear address of the memory access is only compared against DR0-3[56:0].

6 System Instructions

System instructions provide control over the resources used to manage the processor operating environment. This includes memory management, memory protection, task management, interrupt and exception handling, system-management mode, software debug and performance analysis, and model-specific features. Most instructions used to access these resources are privileged and can only be executed while the processor is running at CPL=0, although some instructions can be executed at any privilege level.

Table 6-1 summarizes the instructions used for system management. These include all privileged instructions, instructions whose privilege requirement is under the control of system software, non-privileged instructions that are used primarily by system software, and instructions used to transfer control to system software. Most of the instructions listed in Table 6-1 are summarized in this chapter, although a few are introduced elsewhere in this manual, as indicated in the *Reference* column of Table 6-1.

For details on individual system instructions, see “System Instruction Reference” in Volume 3.

Table 6-1. System Management Instructions

Mnemonic	Name	Privilege			Reference
		CPL=0	O/S[1]	Any	
ARPL	Adjust Requestor Privilege Level			X	“Adjusting Access Rights” on page 178
CLAC	Clear Alignment Check Flag	X			“CLAC and STAC Instructions” on page 175
CLGI	Clear Global Interrupt Flag	X			“Global Interrupt Flag, STGI and CLGI Instructions” on page 514
CLI	Clear Interrupt Flag		X		“CLI and STI Instructions” on page 175
CLRSSBSY	Clear Shadow Stack Busy	X			“CLRSSBSY” on page 179
CLTS	Clear Task-Switched Flag in CR0	X			“CLTS Instruction” on page 175
HLT	Halt	X			“Processor Halt” on page 178
INCSSP	Increment SSP			X	“INCSSP” on page 179
INT3	Interrupt to Debug Vector			X	“Breakpoint Instruction (INT3)” on page 398
INVD	Invalidate Caches	X			“Cache Management” on page 178
INVLPG	Invalidate TLB Entry	X			“INVLPG Instruction” on page 179
INVLPGA	Invalidate TLB Entry in a Specified ASID	X			“Invalidate Page, Alternate ASID” on page 514

Note:
1. *The operating system controls the privilege required to use the instruction.*

Table 6-1. System Management Instructions (continued)

Mnemonic	Name	Privilege			Reference
		CPL=0	O/S[1]	Any	
INVLPGB	Invalidate TLB Entries with Broadcast	X			"INVLPGB Instruction" on page 179
INVPCID	Invalidate TLB Entries in Specified Processor Context	X			"INVPCID Instruction" on page 179
IRET*x*	Interrupt Return (all forms)			X	"Returning From Interrupt Procedures" on page 268
LAR	Load Access-Rights Byte			X	"Checking Access Rights" on page 177
LGDT	Load Global-Descriptor-Table Register	X			"LGDT and LIDT Instructions" on page 176
LIDT	Load Interrupt-Descriptor-Table Register	X			
LLDT	Load Local-Descriptor-Table Register	X			"LLDT and LTR Instructions" on page 177
LMSW	Load Machine-Status Word	X			"LMSW and SMSW Instructions" on page 174
LSL	Load Segment Limit			X	"Checking Segment Limits" on page 177
LTR	Load Task Register	X			"LLDT and LTR Instructions" on page 177
MONITOR	Setup Monitor Address		X		--
MOV CR*n*	Move to/from Control Registers	X			"MOV CR*n* Instructions" on page 174
MOV DR*n*	Move to/from Debug Registers	X			"Accessing Debug Registers" on page 175
MWAIT	Monitor Wait		X		--
RDFSBASE	Read FS Base Address			X	"RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE Instructions" on page 176
RDGSBASE	Read GS Base Address			X	
RDMSR	Read Model-Specific Register	X			"RDMSR and WRMSR Instructions" on page 175
RDPMC	Read Performance-Monitor Counter		X		"RDPMC Instruction" on page 175
RDSSP	Read SSP			X	"RDSSP" on page 179
RDTSC	Read Time-Stamp Counter		X		"RDTSC Instruction" on page 175
RDTSCP	Read Time-Stamp Counter and Processor ID		X		"RDTSCP Instruction" on page 176
RSM	Return from System-Management Mode			X	"Leaving SMM" on page 333

Note:

1. *The operating system controls the privilege required to use the instruction.*

Table 6-1. System Management Instructions (continued)

Mnemonic	Name	Privilege			Reference
		CPL=0	O/S[1]	Any	
RSTORSSP	Restore SSP			X	"RSTORSSP" on page 179
SAVEPREVSSP	Save Previous SSP			X	"SAVEPREVSSP" on page 179
SETSSBSY	Set Shadow Stack Busy	X			"SETSSBSY" on page 179
SGDT	Store Global-Descriptor-Table Register		X		"SGDT and SIDT Instructions" on page 177
SIDT	Store Interrupt-Descriptor-Table Register		X		
SKINIT	Secure Init and Jump with Attestation	X			"Security" on page 540
SLDT	Store Local-Descriptor-Table Register		X		"SLDT and STR Instructions" on page 177
SMSW	Store Machine-Status Word		X		"LMSW and SMSW Instructions" on page 174
STAC	Set Alignment Check Flag		X		"CLAC and STAC Instructions" on page 175
STI	Set Interrupt Flag		X		"CLI and STI Instructions" on page 175
STGI	Set Global Interrupt Flag	X			"Global Interrupt Flag, STGI and CLGI Instructions" on page 514
STR	Store Task Register		X		"SLDT and STR Instructions" on page 177
SWAPGS	Swap GS and KernelGSbase Registers	X			"SWAPGS Instruction" on page 173
SYSCALL	Fast System Call			X	"SYSCALL and SYSRET" on page 171
SYSENTER	System Call			X	"SYSENTER and SYSEXIT (Legacy Mode Only)" on page 173
SYSEXIT	System Return	X			
SYSRET	Fast System Return	X			"SYSCALL and SYSRET" on page 171
VERR	Verify Segment for Reads			X	"Checking Read/Write Rights" on page 177
VERW	Verify Segment for Writes			X	
VMLOAD	Load State from VMCB	X			"VMSAVE and VMLOAD Instructions" on page 491
VMMCALL	Call VMM	X			"VMMCALL Instruction" on page 515
VMRUN	Run Virtual Machine	X			"VMRUN Instruction" on page 486

Note:

1. *The operating system controls the privilege required to use the instruction.*

Table 6-1. System Management Instructions (continued)

Mnemonic	Name	Privilege			Reference
		CPL=0	O/S[1]	Any	
VMSAVE	Save State to VMCB	X			"VMSAVE and VMLOAD Instructions" on page 491
WBINVD	Writeback and Invalidate Caches	X			"Cache Management" on page 178
WBNOINVD	Writeback No Invalidate	X			
WRFSBASE	Write FS Base Address			X	"RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE Instructions" on page 176
WRGSBASE	Write GS Base Address			X	
WRMSR	Write Model-Specific Register	X			"RDMSR and WRMSR Instructions" on page 175
WRSS	Write to Shadow Stack			X	"WRSS" on page 180
WRUSS	Write to User Shadow Stack	X			"WRUSS" on page 180

Note:
1. *The operating system controls the privilege required to use the instruction.*

The following instructions are summarized in this chapter but are not categorized as system instructions, because of their importance to application programming:

- The CPUID instruction returns information critical to system software in initializing the operating environment. It is fully described in Section 3.3, "Processor Feature Identification," on page 71.
- The PUSHF and POPF instructions set and clear certain rFLAGS bits depending on the processor operating mode and privilege level. These dependencies are described in "POPF and PUSHF Instructions" on page 175.

The MOV, PUSH, and POP instructions can be used to load and store segment registers, as described in "MOV, POP, and PUSH Instructions" on page 176.

User Mode Instruction Prevention (UMIP)

This security mode restricts certain instructions so that they do not reveal information about structures that are controlled by the processor when it is at CPL=0. The presence of the UMIP feature is indicated by CPUID Function 0000_0007, ECX[2]=1 . This mode is enabled by setting CR4 bit 11 to a 1. Attempts to set CR4 bit 11 when the UMIP feature is not supported result in a #GP fault. Once CR4[11] is set to 1, the SGDT, SIDT, SLDT, SMSW and STR instructions become available only at CPL=0. Any attempt to execute them with CPL>0 results in a #GP fault with error code 0.

6.1 Fast System Call and Return

Operating systems can use both paging and segmentation to implement protected memory models. Segment descriptors provide the necessary memory protection and privilege checking for segment accesses. By setting segment-descriptor fields appropriately, operating systems can enforce access restrictions as needed.

A disadvantage of segment-based protection and privilege checking is the overhead associated with loading a new segment selector (and its corresponding descriptor) into a segment register. Even when using the flat-memory model, this overhead still occurs when switching between privilege levels because code segments (CS) and stack segments (SS) are reloaded with different segment descriptors.

To initiate a call to the operating system, an application transfers control to the operating system through a gate descriptor (call, interrupt, trap, or task gate). In the past, control was transferred using either a far CALL instruction or a software interrupt. Transferring control through one of these gates is slowed by the segmentation-related overhead, as is the later return using a far RET or IRET instruction. The following checks are performed when control is transferred in this manner:

- Selectors, gate descriptors, and segment descriptors are in the proper form.
- Descriptors lie within the bounds of the descriptor tables.
- Gate descriptors reference the appropriate segment descriptors.
- The caller, gate, and target privileges all allow the control transfer to take place.
- The stack created by the call has sufficient properties to allow the transfer to take place.

In addition to these call-gate checks, other checks are made involving the task-state segment when a task switch occurs.

6.1.1 SYSCALL and SYSRET

SYSCALL and SYSRET Instructions. SYSCALL and SYSRET are low-latency system call and return instructions. These instructions assume the operating system implements a flat-memory model, which greatly simplifies calls to and returns from the operating system. This simplification comes from eliminating unneeded checks, and by loading pre-determined values into the CS and SS segment registers (both visible and hidden portions). As a result, SYSCALL and SYSRET can take fewer than one-fourth the number of internal clock cycles to complete than the legacy CALL and RET instructions. SYSCALL and SYSRET are particularly well-suited for use in 64-bit mode, which requires implementation of a paged, flat-memory model.

SYSCALL and SYSRET require that the code-segment base, limit, and attributes (except for DPL) are consistent for all application and system processes. Only the DPL is allowed to vary. The processor assumes (but does not check) that the SYSCALL target CS segment descriptor entry has DPL=0 and the SYSRET target CS segment descriptor entry has DPL=3.

For details on the SYSCALL and SYSRET instructions, see “System Instruction Reference” in Volume 3.

Because SYSCALL and SYSRET do not use the program stack to store return addresses, the shadow stack mechanism is not used to validate their return addresses. However, when shadow stacks are enabled, SYSCALL and SYSRET save and restore the current SSP as follows:

- If the shadow stack feature is enabled at the current CPL (typically CPL=3), SYSCALL saves the current SSP to the PL3_SSP MSR
- If shadow stacks are enabled at the target CPL (CPL=0), SYSCALL clears the SSP to 0.

- If shadow stacks are enabled at CPL=3, SYSRET restores SSP from PL3_SSP.

SYSCALL and SYSRET MSRs. The STAR, LSTAR, and CSTAR registers are model-specific registers (MSRs) used to specify the target address of a SYSCALL instruction as well as the CS and SS selectors of the called and returned procedures. The SFMASK register is used in long mode to specify how rFLAGS is handled by these instructions. Figure 6-1 shows the STAR, LSTAR, CSTAR, and SFMASK register formats.

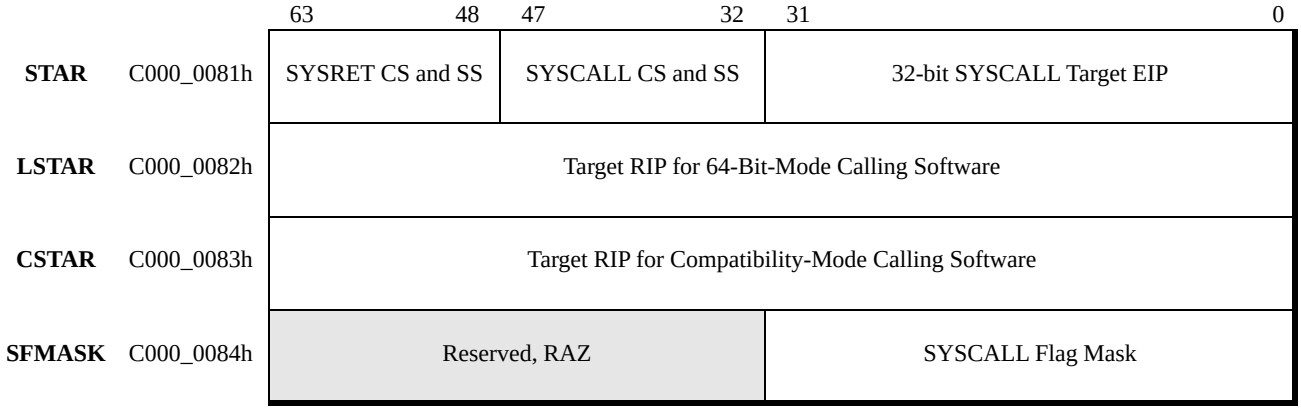


Figure 6-1. STAR, LSTAR, CSTAR, and MASK MSRs

- *STAR*—The STAR register has the following fields (unless otherwise noted, all bits are read/write):
  - *SYSRET CS and SS Selectors*—Bits 63:48. This field is used to specify both the CS and SS selectors loaded into CS and SS during SYSRET. If SYSRET is returning to 32-bit mode (either legacy or compatibility), this field is copied directly into the CS selector field. If SYSRET is returning to 64-bit mode, the CS selector is set to this field + 16. SS.Sel is set to this field + 8, regardless of the target mode. Because SYSRET always returns to CPL 3, the RPL bits 49:48 should be initialized to 11b.
  - *SYSCALL CS and SS Selectors*—Bits 47:32. This field is used to specify both the CS and SS selectors loaded into CS and SS during SYSCALL. This field is copied directly into CS.Sel. SS.Sel is set to this field + 8. Because SYSCALL always switches to CPL 0, the RPL bits 33:32 should be initialized to 00b.
  - *32-bit SYSCALL Target EIP*—Bits 31:0. This is the target EIP of the called procedure.

  The legacy STAR register is not expanded in long mode to provide a 64-bit target RIP address. Instead, long mode provides two new STAR registers—long STAR (LSTAR) and compatibility STAR (CSTAR)—that hold a 64-bit target RIP.
- *LSTAR and CSTAR*—The LSTAR register holds the target RIP of the called procedure in long mode when the calling software is in 64-bit mode. The CSTAR register holds the target RIP of the called procedure in long mode when the calling software is in compatibility mode. The WRMSR instruction is used to load the target RIP into the LSTAR and CSTAR registers. If the RIP written

to either of the MSRs is not in canonical form, a #GP fault is generated on the WRMSR instruction.

- *SFMASK*—The SFMASK register is used to specify which RFLAGS bits are cleared during a SYSCALL. In long mode, SFMASK is used to specify which RFLAGS bits are cleared when SYSCALL is executed. If a bit in SFMASK is *set to 1*, the corresponding bit in RFLAGS is *cleared to 0*. If a bit in SFMASK is cleared to 0, the corresponding RFLAGS bit is not modified.

6.1.2 SYSENTER and SYSEXIT (Legacy Mode Only)

SYSENTER and SYSEXIT Instructions. Like SYSCALL and SYSRET, SYSENTER and SYSEXIT are low-latency system call and return instructions designed for use by system and application software implementing a flat-memory model. However, *these instructions are illegal in long mode and result in an undefined opcode exception (#UD) if software attempts to use them*. Software should use the SYSCALL and SYSRET instructions when running in long mode.

SYSENTER and SYSEXIT MSRs. Three model-specific registers (MSRs) are used to specify the target address and stack pointers for the SYSENTER instruction as well as the CS and SS selectors of the called and returned procedures. The register fields are:

- *SYSENTER Target CS*—Holds the CS selector of the called procedure.
- *SYSENTER Target ESP*—Holds the called-procedure stack pointer. The SS selector is updated automatically to point to the next descriptor entry after the SYSENTER Target CS, and ESP is the offset into that stack segment.
- *SYSENTER Target EIP*—Holds the offset into the CS of the called procedure.

Figure 6-2 shows the register formats and their corresponding MSR IDs.

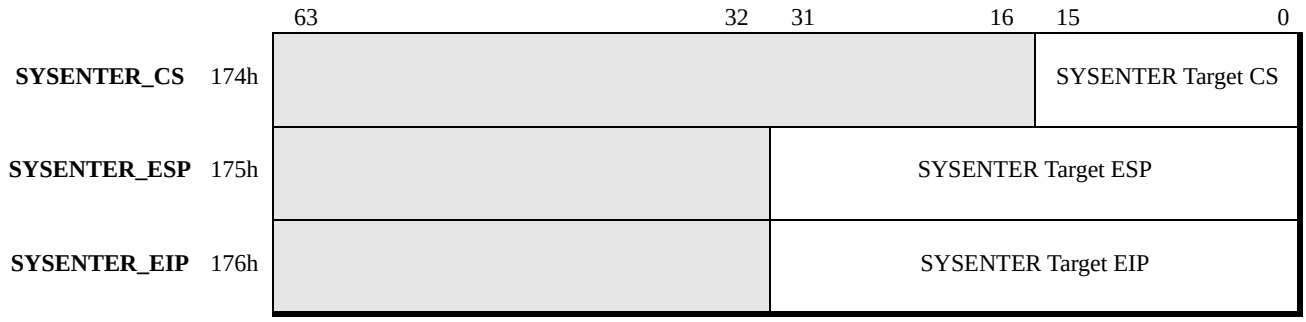


Figure 6-2. SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP MSRs

6.1.3 SWAPGS Instruction

The SWAPGS instruction provides a fast method for system software to load a pointer to system data structures. SWAPGS can be used upon entering system-software routines as a result of a SYSCALL instruction or as a result of an interrupt or exception. Before returning to application software, SWAPGS can restore an application data-structure pointer that was replaced by the system data-structure pointer.

SWAPGS exchanges the base-address value located in the KernelGSbase model-specific register (MSR address C000_0102h) with the base-address value located in the hidden portion of the GS selector register (GS.base). This exchange allows the system-kernel software to quickly access kernel data structures by using the GS segment-override prefix during memory references.

The need for SwapGS arises from the requirement that, upon entry to the OS kernel, the kernel needs to obtain a 64-bit pointer to its essential data structures. When using SYSCALL to implement system calls, no kernel stack exists at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures, from which the kernel stack pointer could be read. Thus, the kernel cannot save GPRs or reference memory. SwapGS does not require any GPR or memory operands, so no registers need to be saved before using it. Similarly, when the OS kernel is entered via an interrupt or exception (where the kernel stack is already set up), SwapGS can be used to quickly get a pointer to the kernel data structures.

See “FS and GS Registers in 64-Bit Mode” on page 80 for more information on using the GS.base register in 64-bit mode.

6.2 System Status and Control

System-status and system-control instructions are used to determine the features supported by a processor, gather information about the current execution state, and control the processor operating modes.

6.2.1 Processor Feature Identification (CPUID)

CPUID Instruction. The CPUID instruction provides complete information about the processor implementation and its capabilities. Software operating at any privilege level can execute the CPUID instruction to collect this information. System software normally uses the CPUID instruction to determine which optional features are available so the system can be configured appropriately. See Section 3.3, “Processor Feature Identification,” on page 71.

6.2.2 Accessing Control Registers

MOV CR*n* Instructions. The MOV CRn instructions can be used to copy data between the control registers and the general-purpose registers. These instructions are privileged and cause a general-protection exception (#GP) if non-privileged software attempts to execute them.

LMSW and SMSW Instructions. The machine status word is located in CR0 register bits 15:0. The *load machine status word* (LMSW) instruction writes only the least-significant four status-word bits (CR0[3:0]). All remaining status-word bits (CR0[15:4]) are left unmodified by the instruction. The instruction is privileged and causes a #GP to occur if non-privileged software attempts to execute it.

The *store machine status word* (SMSW) instruction stores all 16 status-word bits (CR0[15:0]) into the target GPR or memory location. The instruction is not privileged and can be executed by all software.

CLTS Instruction. The *clear task-switched bit* instruction (CLTS) clears CR0.TS to 0. The CR0.TS bit is set to 1 by the processor every time a task switch takes place. The bit is useful to system software in determining when the x87 and multimedia register state should be saved or restored. See “Task Switched (TS) Bit” on page 44 for more information on using CR0.TS to manage x87-instruction state. The CLTS instruction is privileged and causes a #GP to occur if non-privileged software attempts to execute it.

6.2.3 Accessing the RFLAGS Register

The RFLAGS register contains both application and system bits. This section describes the instructions used to read and write system bits. Descriptions of instruction effects on application flags can be found in “Flags Register” in Volume 1 and “Instruction Effects on rFLAGS” in Volume 3.

POPF and PUSHF Instructions. The *pop and push rFLAGS* instructions are used for moving data between the rFLAGS register and the stack. They are not strictly system instructions, but their behavior is mode-dependent.

CLI and STI Instructions. The *clear interrupt* (CLI) and *set interrupt* (STI) instructions modify only the RFLAGS.IF bit or RFLAGS.VIF bit. Clearing RFLAGS.IF to 0 causes the processor to ignore maskable interrupts. Setting RFLAGS.IF to 1 causes the processor to allow maskable interrupts.

See “Virtual Interrupts” on page 279 for more information on the operation of these instructions when virtual-8086 mode extensions are enabled (CR4.VME=1).

CLAC and STAC Instructions. The *clear alignment check flag (CLAC) and set alignment check flag (STAC)* instructions modify only the RFLAGS.AC bit.

6.2.4 Accessing Debug Registers

The MOV DR*n* instructions are used to copy data between the debug registers and the general-purpose registers. These instructions are privileged and cause a general-protection exception (#GP) if non-privileged software attempts to execute them. See “Debug Registers” on page 386 for a detailed description of the debug registers.

6.2.5 Accessing Model-Specific Registers

RDMSR and WRMSR Instructions. The *read/write model-specific register* instructions (RDMSR and WRMSR) can be used by privileged software to access the 64-bit MSRs. See “Model-Specific Registers (MSRs)” on page 59 for details about the MSRs.

RDPMC Instruction. The *read performance-monitoring counter* instruction, RDPMC, is used to read the model-specific performance-monitoring counter registers.

RDTSC Instruction. The *read time-stamp counter* instruction, RDTSC, is used to read the model-specific time-stamp counter (TSC) register.

RDTSCP Instruction. The *read time-stamp counter and processor ID* instruction, RDTSCP, is used to read the model-specific time-stamp counter (TSC) register. as well as the low 32 bits of the TSC_AUX register (MSR C000_0103h).

6.3 Segment Register and Descriptor Register Access

The AMD64 architecture supports the legacy instructions that load and store segment registers and descriptor registers. In some cases the instruction capabilities are expanded to support long mode.

6.3.1 Accessing Segment Registers

MOV, POP, and PUSH Instructions. The MOV and POP instructions can be used to load a selector into a segment register from a general-purpose register or memory (MOV) or from the stack (POP). Any segment register, except the CS register, can be loaded with the MOV and POP instructions. The CS register must be loaded with a far-transfer instruction.

All segment register selectors can be stored in a general-purpose register or memory using the MOV instruction or pushed onto the stack using the PUSH instruction.

When a selector is loaded into a segment register, the processor automatically loads the corresponding descriptor-table entry into the hidden portion of the selector register. The hidden portion contains the base address, limit, and segment attributes.

Segment-load and segment-store instructions work normally in 64-bit mode. The appropriate entry is read from the system descriptor table (GDT or LDT) and is loaded into the hidden portion of the segment descriptor register. However, the contents of data-segment and stack-segment descriptor registers are ignored, except in the case of the FS and GS segment-register base fields—see “FS and GS Registers in 64-Bit Mode” on page 80 for more information.

The ability to use segment-load instructions allows a 64-bit operating system to set up segment registers for a compatibility-mode application before switching to compatibility mode.

6.3.2 Accessing Segment Register Hidden State

WRMSR and RDMSR Instructions. The base address field of the hidden state of the FS and GS registers are mapped to MSRs and may be read and written by privileged software when running in 64-bit mode.

RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE Instructions. When supported and enabled, these instructions allow software running at any privilege level to read and write the base address field of the hidden state of the FS and GS segment registers. These instructions are only defined in 64-bit mode.

6.3.3 Accessing Descriptor-Table Registers

LGDT and LIDT Instructions. The *load GDTR* (LGDT) and *load IDTR* (LIDT) instructions load a *pseudo-descriptor* from memory into the GDTR or IDTR registers, respectively.

LLDT and LTR Instructions. The *load LDTR* (LLDT) and *load TR* (LTR) instructions load a system-segment descriptor from the GDT into the LDTR and TR segment-descriptor registers (hidden portion), respectively.

SGDT and SIDT Instructions. The *store GDTR* (SGDT) and *store IDTR* (SIDT) instructions reverse the operation of the LGDT and LIDT instructions. SGDT and SIDT store a pseudo-descriptor from the GDTR or IDTR register into memory.

SLDT and STR Instructions. In all modes, the *store LDTR* (SLDT) and *store TR* (STR) instructions store the LDT or task selector from the visible portion of the LDTR or TR register into a general-purpose register or memory, respectively. The hidden portion of the LDTR or TR register is not stored.

6.4 Protection Checking

Several instructions are provided to allow software to determine the outcome of a protection check before performing a memory access that could result in a protection violation. By performing the checks before a memory access, software can avoid violations that result in a general-protection exception (#GP).

6.4.1 Checking Access Rights

LAR Instruction. The *load access-rights* (LAR) instruction can be used to determine if access to a segment is allowed, based on privilege checks and type checks. The LAR instruction uses a segment-selector in the source operand to reference a descriptor in the GDT or LDT. LAR performs a set of access-rights checks and, if successful, loads the segment-descriptor access rights into the destination register. Software can further examine the access-rights bits to determine if access into the segment is allowed.

6.4.2 Checking Segment Limits

LSL Instruction. The *load segment-limit* (LSL) instruction uses a segment-selector in the source operand to reference a descriptor in the GDT or LDT. LSL performs a set of preliminary access-rights checks and, if successful, loads the segment-descriptor limit field into the destination register. Software can use the limit value in comparisons with pointer offsets to prevent segment limit violations.

6.4.3 Checking Read/Write Rights

VERR and VERW Instructions. The *verify read-rights* (VERR) and *verify write-rights* (VERW) can be used to determine if a target code or data segment (not a system segment) can be read or written from the current privilege level (CPL). The source operand for these instructions is a pointer to the segment selector to be tested. If the tested segment (code or data) is readable from the current CPL, the VERR instruction sets RFLAGS.ZF to 1; otherwise, it is cleared to zero. Likewise, if the tested data segment is writable, the VERW instruction sets the RFLAGS.ZF to 1. A code segment cannot be tested for writability.

6.4.4 Adjusting Access Rights

ARPL Instruction. The *adjust RPL-field* (ARPL) instruction can be used by system software to prevent access into privileged-data segments by lower-privileged software. This can happen if an application passes a selector to system software and the selector RPL is less than (has greater privilege than) the calling-application CPL. To prevent this surrogate access, system software executes ARPL with the following operands:

- The destination operand is the data-segment selector passed to system software by the application.
- The source operand is the application code-segment selector (available on the system-software stack as a result of the CALL into system software by the application).

ARPL is not supported in 64-bit mode.

6.5 Processor Halt

The *processor halt* instruction (HLT) halts instruction execution, leaving the processor in the halt state. No registers or machine state are modified as a result of executing the HLT instruction. The processor remains in the halt state until one of the following occurs:

- A non-maskable interrupt (NMI).
- An enabled, maskable interrupt (INTR).
- Processor reset (RESET).
- Processor initialization (INIT).
- System-management interrupt (SMI).

6.6 Cache and TLB Management

Cache-management instructions are used by system software to maintain coherency within the memory hierarchy. Memory coherency and caches are discussed in Chapter 7, “Memory System.” Similarly, TLB-management instructions are used to maintain coherency between page translations cached in the TLB and the translation tables maintained by system software in memory. See “Translation-Lookaside Buffer (TLB)” on page 154 for more information.

6.6.1 Cache Management

WBINVD and WBNOINVD Instructions. The *writeback and invalidate* (WBINVD) and *writeback no invalidate* (WBNOINVD) instructions are used to write all modified cache lines to memory so that memory contains the most recent copy of data. After the writes are complete, the WBINVD instruction invalidates all cache lines, whereas the WBNOINVD instruction may leave the lines in the cache hierarchy in a non-modified state. These instructions operate on all caches in the memory hierarchy, including caches that are external to the processor. See the instructions' description in Volume 3 for further operational details.

INVD Instruction. The *invalidate* (INVD) instruction is used to invalidate all cache lines in all caches in the memory hierarchy. Unlike the WBINVD instruction, no modified cache lines are written to memory. The INVD instruction should only be used in situations where memory coherency is not required.

6.6.2 TLB Invalidation

INVLPG Instruction. The *invalidate TLB entry* (INVLPG) instruction can be used to invalidate specific entries within the TLB. The source operand is a virtual-memory address that specifies the TLB entry to be invalidated. Invalidating a TLB entry does not remove the associated page-table entry from the data cache. See “Translation-Lookaside Buffer (TLB)” on page 154 for more information.

INVLPGA Instruction. The *invalidate TLB entry in a Specified ASID* instruction (INVLPGA) can be used to invalidate TLB entries associated with the specified ASID. See “Invalidate Page, Alternate ASID” on page 514.

INVLPGB Instruction. The *invalidate TLB with Broadcast* instruction (INVLPGB) can be used to invalidate a specified range of TLB entries on the local processor and broadcast the invalidation to remote processors. See “INVLPGB” in Volume 3.

INVPCID Instruction. The *invalidate TLB entries in Specified PCID* instruction (INVPCID) can be used to invalidate TLB entries of the specified Processor Context ID. See “INVPCID” in Volume 3.

6.7 Shadow Stack Management

The following instructions are available to software for use in managing shadow stacks if the shadow stack feature is present as indicated by CPUID Fn0000_0007_x0_ECX[CET_SS] (bit 7) =1. Except for RDSSP, attempting to execute these instructions when shadow stacks are disabled results in a #UD exception. For more information refer to the detailed instruction descriptions in APM volume 3.

CLRSSBSY. Validates a shadow stack token and clears the tokens busy bit. This is a privileged instruction.

INCSSP. Increment SSP by ‘n’ stack frames. Used to pop unneeded items from a shadow stack.

RDSSP. Read the SSP into a GPR. Treated as a NOP if shadow stacks are disabled.

RSTORSSP. Used to switch shadow stacks. Expects a ‘shadow stack restore token’ at the top of the new shadow stack. Upon validating this token, sets the token’s busy bit and sets SSP to the top of the new shadow stack.

SAVEPREVSSP. Copies a ‘previous SSP token’ from the current shadow stack back to the previous stack for later use by an RSTORSSP instruction.

SETSSBSY. Validates the shadow stack token pointed to by the PL0_SSP MSR. If valid, sets the busy bit to 1 and sets SSP = PL0_SSP. This is a privileged instruction.

WRSS. Writes the source operand to a shadow stack. This instruction must be enabled in the U_CET and S_CET MSRs, otherwise a #UD is generated.

WRUSS. Writes the source operand to a user shadow stack. This is a privileged instruction.

7 Memory System

This chapter describes:

- Cache coherency mechanisms
- Cache control mechanisms
- Memory typing
- Memory mapped I/O
- Memory ordering rules
- Serializing instructions

Figure 7-1 on page 182 shows a conceptual picture of a processor and memory system, and how data and instructions flow between the various components. This diagram is not intended to represent a specific microarchitectural implementation but instead is used to illustrate the major memory-system components covered by this chapter.

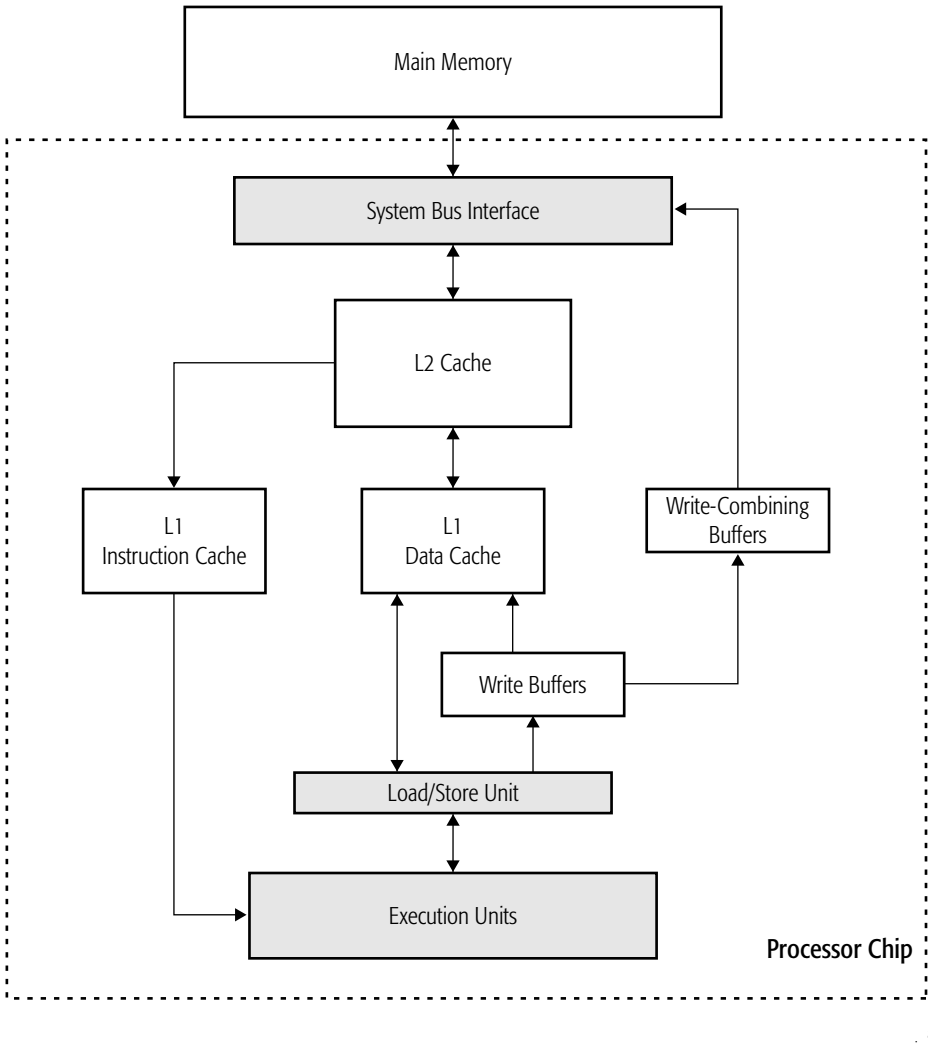


Figure 7-1. Processor and Memory System

The memory-system components described in this chapter are shown as *unshaded* boxes in Figure 7-1. Those items are summarized in the following paragraphs.

Main memory is external to the processor chip and is the memory-hierarchy level farthest from the processor execution units.

Caches are the memory-hierarchy levels closest to the processor execution units. They are much smaller and much faster than main memory, and can be either internal or external to the processor chip. Caches contain copies of the most frequently used instructions and data. By allowing fast access to frequently used data, software can run much faster than if it had to access that data from main memory. Figure 7-1 shows three caches, all internal to the processor:

- *L1 Data Cache*—The L1 (level-1) data cache holds the data most recently read or written by the software running on the processor.
- *L1 Instruction Cache*—The L1 instruction cache is similar to the L1 data cache except that it holds only the instructions executed most frequently. In some processor implementations, the L1 instruction cache can be combined with the L1 data cache to form a unified L1 cache.
- *L2 Cache*—The L2 (level-2) cache is usually several times larger than the L1 caches, but it is also slower. It is common for L2 caches to be implemented as a unified cache containing both instructions and data. Recently used instructions and data that do not fit within the L1 caches can reside in the L2 cache. The L2 cache can be exclusive, meaning it does not cache information contained in the L1 cache. Conversely, inclusive L2 caches contain a copy of the L1-cached information.

Memory-read operations from cacheable memory first check the cache to see if the requested information is available. A *read hit* occurs if the information is available in the cache, and a *read miss* occurs if the information is not available. Likewise, a *write hit* occurs if the memory write can be stored in the cache, and a *write miss* occurs if it cannot be stored in the cache.

Caches are divided into fixed-size blocks called *cache lines*. The cache allocates lines to correspond to regions in memory of the same size as the cache line, aligned on an address boundary equal to the cache-line size. For example, in a cache with 32-byte lines, the cache lines are aligned on 32-byte boundaries and byte addresses 0007h and 001Eh are both located in the same cache line. The size of a cache line is implementation dependent. Most implementations have either 32-byte or 64-byte cache lines. The implemented cache line size is reported by CPUID Fn8000_0005 and Fn8000_0006 for the various caches, as described in Appendix E of Volume 3.

The process of loading data into a cache is a *cache-line fill*. Even if only a single byte is requested, all bytes in a cache line are loaded from memory. Typically, a cache-line fill must remove (evict) an existing cache line to make room for the new line loaded from memory. This process is called *cache-line replacement*. If the existing cache line was modified before the replacement, the processor performs a cache-line *writeback* to main memory when it performs the cache-line fill.

Cache-line writebacks help maintain *coherency* between the caches and main memory. Internally, the processor can also maintain cache coherency by *internally probing* (checking) the other caches and write buffers for a more recent version of the requested data. External devices can also check processor caches for more recent versions of data by *externally probing* the processor. Throughout this document, the term *probe* is used to refer to external probes, while internal probes are always qualified with the word *internal*.

Write buffers temporarily hold data writes when main memory or the caches are busy with other memory accesses. The existence of write buffers is implementation dependent.

Implementations of the architecture can use *write-combining buffers* if the order and size of non-cacheable writes to main memory is not important to the operation of software. These buffers can combine multiple, individual writes to main memory and transfer the data in fewer bus transactions.

7.1 Single-Processor Memory Access Ordering

The flexibility with which memory accesses can be ordered is closely related to the flexibility in which a processor implementation can *execute* and *retire* instructions. Instruction execution *creates* results and status and determines whether or not the instruction causes an exception. Instruction retirement *commits* the results of instruction execution, in program order, to software-visible resources such as memory, caches, write-combining buffers, and registers, or it causes an exception to occur if instruction execution created one.

Implementations of the AMD64 architecture retire instructions in program order, but implementations can execute instructions in any order, subject only to data dependencies. Implementations can also *speculatively execute* instructions—executing instructions before knowing they are needed. Internally, implementations manage data reads and writes so that instructions complete in order. However, because implementations can execute instructions out of order and speculatively, the sequence of memory accesses performed by the hardware can appear to be out of program order. The following sections describe the rules governing memory accesses to which processor implementations adhere. These rules may be further restricted, depending on the memory type being accessed. Further, these rules govern single processor operation; see “Multiprocessor Memory Access Ordering” on page 186 for multiprocessor ordering rules.

7.1.1 Read Ordering

Generally, reads do not affect program order because they do not affect the state of software-visible resources other than register contents. However, some system devices might be sensitive to reads. In such a situation software can map a read-sensitive device to a memory type that enforces strong read-ordering, or use read/write barrier instructions to force strong read-ordering.

For cacheable memory types, the following rules govern read ordering:

- Out-of-order reads are allowed to the extent that they can be performed transparently to software, such that the appearance of in-order execution is maintained. Out-of-order reads can occur as a result of out-of-order instruction execution or speculative execution. The processor can read memory and perform cache refills out-of-order to allow out-of-order execution to proceed.
- Speculative reads are allowed. A speculative read occurs when the processor begins executing a memory-read instruction before it knows the instruction will actually complete. For example, the processor can predict a branch will occur and begin executing instructions following the predicted branch before it knows whether the prediction is valid. When one of the speculative instructions reads data from memory, the read itself is speculative. Cache refills may also be performed speculatively.
- Reads can be reordered ahead of writes. Reads are generally given a higher priority by the processor than writes because instruction execution stalls if the read data required by an instruction is not immediately available. Allowing reads ahead of writes usually maximizes software performance.
- A read *cannot* be reordered ahead of a prior write if the read is from the same location as the prior write. In this case, the read instruction stalls until the write instruction completes execution. The

read instruction requires the result of the write instruction for proper software operation. For cacheable memory types, the write data can be forwarded to the read instruction before it is actually written to memory.

- Instruction fetching constitutes a parallel, asynchronous stream of reads that is independent from and unordered with respect to the read accesses performed by loads in that instruction stream.

7.1.2 Write Ordering

Writes affect program order because they affect the state of software-visible resources. The following rules govern write ordering:

- Generally, out-of-order writes are *not* allowed. Write instructions executed out of order cannot commit (write) their result to memory until all previous instructions have completed in program order. The processor can, however, hold the result of an out-of-order write instruction in a private buffer (not visible to software) until that result can be committed to memory.
- It is possible for writes to *write-combining* memory types to appear to complete out of order, relative to writes into other memory types. See “Memory Types” on page 192 and “Write Combining” on page 198 for additional information.
- Speculative writes are *not* allowed. As with out-of-order writes, speculative write instructions cannot commit their result to memory until all previous instructions have completed in program order. Processors can hold the result in a private buffer (not visible to software) until the result can be committed.
- Write buffering is allowed. When a write instruction completes and commits its result, that result can be buffered until it is actually written to system memory in program order. Although the write buffer itself is not directly accessible by software, the results in the buffer are accessible by subsequent memory accesses to the locations that are buffered, including reads for which only a subset of bytes being accessed are in the buffer. For example, a doubleword read that overlaps a single modified byte in the write buffer can return the buffered value for that byte before that write has been committed to memory.

  In general, any read from cacheable memory returns the net result of all prior globally and locally visible writes to those bytes, as performed in program order. A given implementation may provide bytes from the write buffer to satisfy this, or may stall the read until any overlapping buffered writes have been committed to memory. For cacheable memory types, the write buffer can be read out-of-order and speculatively, just like memory.
- Write combining is allowed. In some situations software can relax the write-ordering rules through the use of a Write Combining memory type or non-temporal store instructions, and allow several writes to be combined into fewer writes to memory. When write-combining is used, it is possible for writes to other memory types to proceed ahead of (out-of-order) memory-combining writes, unless the writes are to the same address. Write-combining should be used only when the order of writes does not affect program order (for example, writes to a graphics frame buffer).

7.1.3 Read/Write Barriers

When the order of memory accesses must be strictly enforced, software can use read/write barrier instructions to force reads and writes to proceed in program order. Read/write barrier instructions force all prior reads or writes to complete before subsequent reads or writes are executed. The LFENCE, SFENCE, and MFENCE instructions are provided as dedicated read, write, and read/write barrier instructions (respectively). Serializing instructions, I/O instructions, and locked instructions (including the implicitly locked XCHG instruction) can also be used as read/write barriers. Barrier instructions are useful for controlling ordering between differing memory types as well as within one memory type; see Section 7.3.1, “Special Coherency Considerations,” on page 191 for details.

Table 7-1 on page 194 summarizes the memory-access ordering possible for each memory type supported by the AMD64 architecture.

7.2 Multiprocessor Memory Access Ordering

The term memory ordering refers to the sequence in which memory accesses are performed by the memory system, as observed by all processors or programs.

To improve performance of applications, AMD64 processors can speculatively execute instructions out of program order and temporarily hold out-of-order results. However, certain rules are followed with regard to normal cacheable accesses on naturally aligned boundaries to WB memory.

In the examples below, all memory values are initialized to zero.

From the point of view of a program, in ascending order of priority:

- All loads, stores and I/O operations from a single processor appear to occur in program order to the code running on that processor and all instructions appear to execute in program order.
- Successive stores from a single processor are committed to system memory and visible to other processors in program order. A store by a processor cannot be committed to memory before a read appearing earlier in the program has captured its targeted data from memory. In other words, stores from a processor cannot be reordered to occur prior to a load preceding it in program order.

  In this context:

  - Loads do not pass previous loads (loads are not reordered). Stores do not pass previous stores (stores are not reordered)

Processor 0	**Processor 1**
Store A ← 1	Load B
Store B ← 1	Load A

    Load A cannot read 0 when Load B reads 1. (This rule may be violated in the case of loads as part of a string operation, in which one iteration of the string reads 0 for Load A while another iteration reads 1 for Load B.)
  - Stores do not pass loads

Processor 0	Processor 1
Load A	Load B
Store B ← 1	Store A ← 1

Load A and Load B cannot both read 1.

- Stores from a processor appear to be committed to the memory system in program order; however, stores can be delayed arbitrarily by store buffering while the processor continues operation. Therefore, stores from a processor may not appear to be sequentially consistent.

Processor 0	Processor 1
Store A ← 1	Store B ← 1
…	…
Store A ← 2	Store B ← 2
…	…
Load B	Load A

Both Load A and Load B may read 1. Also, due to possible write combining one or both processors may not actually store a 1 at the designated location.

- Non-overlapping Loads may pass stores.

Processor 0	Processor 1
Store A ← 1	Store B ← 1
Load B	Load A

All combinations of values (00, 01, 10, and 11) may be observed by Processors 0 and 1.

- Where sequential consistency is needed (for example in Dekker's algorithm for mutual exclusion), an MFENCE instruction should be used between the store and the subsequent load, or a locked access, such as XCHG, should be used for the store.

Processor 0	Processor 1
Store A ← 1	Store B ← 1
MFENCE	MFENCE
Load B	Load A

Load A and Load B cannot both read 0.

- Loads that partially overlap prior stores may return the modified part of the load operand from the store buffer, combining globally visible bytes with bytes that are only locally visible. To ensure that such loads return only a globally visible value, an MFENCE or locked access must be used between the store and the dependent load, or the store or load must be performed with a locked operation such as XCHG.

- Stores to different locations in memory observed from two (or more) other processors will appear in the same order to all observers. Behavior such as that shown in this code example,

Processor 0	Processor 1	Processor X	Processor Y
Store A ← 1	Store B ← 1		
		Load A (1)	Load B (1)
		Load B (0)	Load A (0)

in which processor X sees store A from processor 0 before store B from processor 1, while processor Y sees store B from processor 1 before store A from processor 0, is not allowed.

- Dependent stores between different processors appear to occur in program order, as shown in the code example below.

Processor 0	Processor 1	Processor 2
Store A ← 1		
	Load A (1)	
	Store B ← 1	
		Load B (1)
		Load A (1)

If processor 1 reads a value from A (written by processor 0) before carrying out a store to B, and if processor 2 reads the updated value from B, a subsequent read of A must also be the updated value.

- The local visibility (within a processor) for a memory operation may differ from the global visibility (from another processor). Using a data bypass, a local load can read the result of a local store in a store buffer, before the store becomes globally visible. Program order is still maintained when using such bypasses.

Processor 0	Processor 1
Store A ← 1	Store B ← 1
Load r1 A	Load r3 B
Load r2 B	Load r4 A

Load A in processor 0 can read 1 using the data bypass, while Load A in processor 1 can read 0. Similarly, Load B in processor 1 can read 1 while Load B in processor 0 can read 0. Therefore, the result r1 = 1, r2 = 0, r3 = 1 and r4 = 0 may occur. There are no constraints on the relative order of when the Store A of processor 0 is visible to processor 1 relative to when the Store B of processor 1 is visible to processor 0.

If a very strong memory ordering model is required that does not allow local store-load bypasses, an MFENCE instruction or a synchronizing instruction such as XCHG or a locked Read-modify-write should be used between the store and the subsequent load. This enforces a memory ordering stronger than total store ordering.

Processor 0	Processor 1
Store A ← 1	Store B ← 1
MFENCE	MFENCE

Processor 0	Processor 1
Load r1 A	Load r3 B
Load r2 B	Load r4 A

In this example, the MFENCE instruction ensures that any buffered stores are globally visible before the loads are allowed to execute, so the result r1 = 1, r2 = 0, r3 = 1 and r4 = 0 will not occur.

7.3 Memory Coherency and Protocol

Implementations that support caching support a cache-coherency protocol for maintaining coherency between main memory and the caches. The cache-coherency protocol is also used to maintain coherency between all processors in a multiprocessor system. The cache-coherency protocol supported by the AMD64 architecture is the *MOESI* (modified, owned, exclusive, shared, invalid) protocol. The states of the MOESI protocol are:

- *Invalid*—A cache line in the invalid state does not hold a valid copy of the data. Valid copies of the data can be either in main memory or another processor cache.
- *Exclusive*—A cache line in the exclusive state holds the most recent, correct copy of the data. The copy in main memory is also the most recent, correct copy of the data. No other processor holds a copy of the data.
- *Shared*—A cache line in the shared state holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the shared state, as well. If no other processor holds it in the *owned* state, then the copy in main memory is also the most recent.
- *Modified*—A cache line in the modified state holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy.
- *Owned*—A cache line in the owned state holds the most recent, correct copy of the data. The owned state is similar to the shared state in that other processors can hold a copy of the most recent, correct data. Unlike the shared state, however, the copy in main memory can be stale (incorrect). Only one processor can hold the data in the owned state—all other processors must hold the data in the shared state.

Figure 7-2 on page 190 shows the general MOESI state transitions possible with various types of memory accesses. This is a logical software view, not a hardware view, of how cache-line state transitions. Instruction-execution activity and external-bus transactions can both be used to modify the cache MOESI state in multiprocessing or multi-mastering systems.

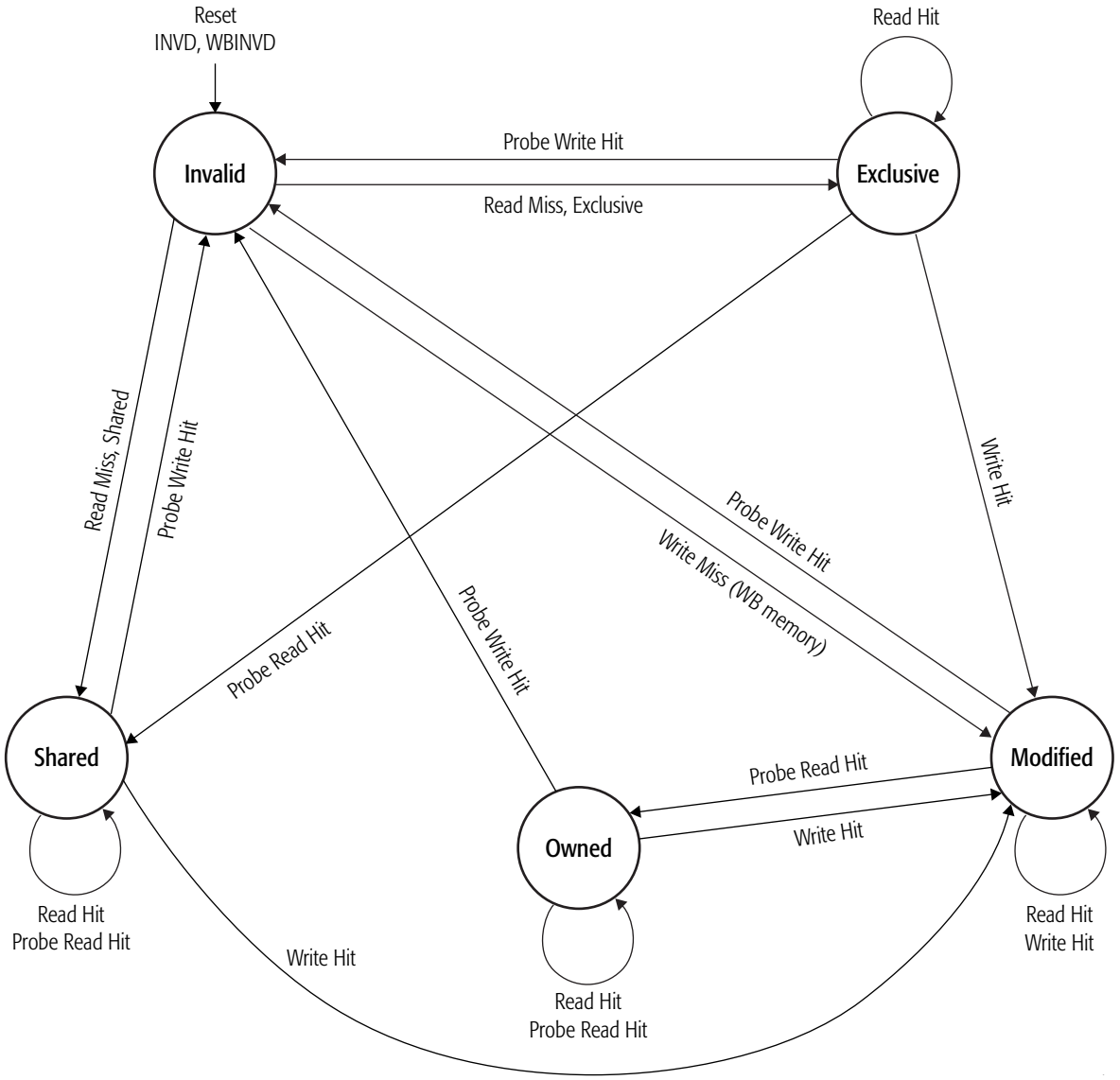


Figure 7-2. MOESI State Transitions

To maintain memory coherency, external bus masters (typically other processors with their own internal caches) need to acquire the most recent copy of data before caching it internally. That copy can be in main memory or in the internal caches of other bus-mastering devices. When an external master has a cache read-miss or write-miss, it *probes* the other mastering devices to determine whether the most recent copy of data is held in any of their caches. If one of the other mastering devices holds the most recent copy, it provides it to the requesting device. Otherwise, the most recent copy is provided by main memory.

There are two general types of bus-master probes:

- Read probes indicate the external master is requesting the data for read purposes.
- Write probes indicate the external master is requesting the data for the purpose of modifying it.

Referring back to Figure 7-2 on page 190, the state transitions involving probes are initiated by other processors and external bus masters into the processor. Some read probes are initiated by devices that intend to cache the data. Others, such as those initiated by I/O devices, do not intend to cache the data. Some processor implementations do not change the data MOESI state if the read probe is initiated by a device that does not intend to cache the data.

State transitions involving read misses and write misses can cause the processor to generate probes into external bus masters and to read main memory.

Read hits do not cause a MOESI-state change. Write hits generally cause a MOESI-state change into the modified state. If the cache line is already in the modified state, a write hit does not change its state.

The specific operation of external-bus signals and transactions and how they influence a cache MOESI state are implementation dependent. For example, an implementation could convert a write miss to a WB memory type into two separate MOESI-state changes. The first would be a read-miss placing the cache line in the exclusive state. This would be followed by a write hit into the exclusive cache line, changing the cache-line state to modified.

7.3.1 Special Coherency Considerations

In some cases, data can be modified in a manner that is impossible for the memory-coherency protocol to handle due to the effects of instruction prefetching. In such situations software must use serializing instructions and/or cache-invalidation instructions to ensure subsequent data accesses are coherent.

An example of this type of a situation is a page-table update followed by accesses to the physical pages referenced by the updated page tables. The following sequence of events shows what can happen when software changes the translation of virtual-page *A* from physical-page *M* to physical-page *N*:

1. Software invalidates the TLB entry. The tables that translate virtual-page *A* to physical-page *M* are now held only in main memory. They are not cached by the TLB.
2. Software changes the page-table entry for virtual-page A in main memory to point to physical-page *N* rather than physical-page *M*.
3. Software accesses data in virtual-page A.

During Step 3, software expects the processor to access the data from physical-page *N*. However, it is possible for the processor to prefetch the data from physical-page *M* before the page table for virtual-page *A* is updated in Step 2. This is because the physical-memory references for the *page tables* are different than the physical-memory references for the *data*. Because the physical-memory references are different, the processor does not recognize them as requiring coherency checking and believes it is safe to prefetch the data from virtual-page A, which is translated into a read from physical page M. Similar behavior can occur when instructions are prefetched from beyond the page table update instruction.

To prevent this problem, software must use an INVLPG or MOV CR3 instruction immediately after the page-table update to ensure that subsequent instruction fetches and data accesses use the correct virtual-page-to-physical-page translation. It is not necessary to perform a TLB invalidation operation preceding the table update.

7.3.2 Access Atomicity

Cacheable, naturally-aligned single loads or stores of up to a quadword are atomic on any processor model, as are misaligned loads or stores of less than a quadword that are contained entirely within a naturally-aligned quadword. Misaligned load or store accesses typically incur a small latency penalty. Model-specific relaxations of this quadword atomicity boundary, with respect to this latency penalty, may be found in a given processor's Software Optimization Guide.

Misaligned accesses can be subject to interleaved accesses from other processors or cache-coherent devices which can result in unintended behavior. Atomicity for misaligned accesses can be achieved where necessary by using the XCHG instruction or any suitable LOCK-prefixed instruction. Note that misaligned locked accesses may incur a significant performance penalty on various processor models.

7.4 Memory Types

Memory type is an attribute that can be associated with a specific region of virtual or physical memory. Memory type designates certain caching and ordering behaviors for loads and stores to addresses in that region. Most memory types are explicitly assigned, although some are inferred by the hardware from current processor state and instruction context.

The AMD64 architecture defines the following memory types:

- *Uncacheable (UC)*—Reads from, and writes to, UC memory are not cacheable. Reads from UC memory cannot be speculative. Write-combining to UC memory is not allowed. Reads from or writes to UC memory cause the write buffers to be written to memory and be invalidated prior to the access to UC memory.

  The UC memory type is useful for memory-mapped I/O devices where strict ordering of reads and writes is important. Note that this strong ordering is with respect to UC accesses only; reads to memory types which support speculative operation may bypass non-conflicting UC accesses.
- *Cache Disable (CD)*—The CD memory type is a form of uncacheable memory type that is inferred when the L1 caches are disabled but not invalidated, or for certain conflicting memory type assignments from the Page Attribute Table (PAT) and Memory Type Range Register (MTRR) mechanisms. The former case occurs when caches are disabled by setting CR0.CD to 1 without invalidating the caches with either the INVD or WBINVD instruction for any reference to a region designated as cacheable. The latter case occurs when a specific type has been assigned to a virtual page via PAT, and a conflicting type has been assigned to the mapped physical page via an MTRR (see “Combined Effect of MTRRs and PAT” on page 221 and “Combining Memory Types, MTRRs” on page 536 for details).

  For the L1 data cache and the L2 cache, reads from, and writes to, CD memory that hit the cache, or any other caches in the system, cause the cache line(s) to be invalidated before accessing main

memory. If a cache line is in the modified state, the line is written to main memory prior to being invalidated. The access is allowed to proceed after any invalidations are complete.

For the L1 instruction cache, instruction fetches from CD memory that hit the cache read the cached instructions rather than access main memory. Instruction fetches that miss the cache access main memory and do not cause cache-line replacement. Writes to CD memory that hit in the instruction cache cause the line to be invalidated.

- *Write-Combining (WC)*—Reads from, and writes to, WC memory are not cacheable. Reads from WC memory can be speculative.

  Writes to this memory type can be combined internally by the processor and written to memory as a single write operation to reduce memory accesses. For example, four word writes to consecutive addresses can be combined by the processor into a single quadword write, resulting in one memory access instead of four.

  The WC memory type is useful for graphics-display memory buffers where the order of writes is not important.

- *Write-Combining Plus (WC+)*—WC+ is an uncacheable memory type, and combines writes in write-combining buffers like WC. Unlike WC (but like the CD memory type), accesses to WC+ memory probe the caches on all processors (including the caches of the processor issuing the request) to maintain coherency. This ensures that cacheable writes are observed by WC+ accesses.

- *Write-Protect (WP)*—Reads from WP memory are cacheable and allocate cache lines on a read miss. Reads from WP memory can be speculative.

  Writes to WP memory that hit in the cache do not update the cache. Instead, all writes update memory (write to memory), and writes that hit in the cache invalidate the cache line. Write buffering of WP memory is allowed.

  The WP memory type is useful for shadowed-ROM memory where updates must be immediately visible to all devices that read the shadow locations.

- *Writethrough (WT)*—Reads from WT memory are cacheable and allocate cache lines on a read miss. Reads from WT memory can be speculative.

  All writes to WT memory update main memory, and writes that hit in the cache update the cache line (cache lines remain in the same state after a write that hits a cache line). Writes that miss the cache do not allocate a cache line. Write buffering of WT memory is allowed.

- *Writeback (WB)*—Reads from WB memory are cacheable and allocate cache lines on a read miss. Cache lines can be allocated in the shared, exclusive, or modified states. Reads from WB memory can be speculative.

  All writes that hit in the cache update the cache line and place the cache line in the modified state. Writes that miss the cache allocate a new cache line and place the cache line in the modified state. Writes to main memory only take place during writeback operations. Write buffering of WB memory is allowed.

  The WB memory type provides the highest-possible performance and is useful for most software and data stored in system memory (DRAM).

Table 7-1 shows the memory access ordering possible for each memory type supported by the AMD64 architecture. Table 7-3 on page 196 shows the ordering behavior of various operations on various memory types in greater detail. Table 7-2 on page 194 shows the caching policy for the same memory types.

Table 7-1. Memory Access by Memory Type

Memory Access Allowed		Memory Type				
		UC/CD	WC	WP	WT	WB
Read	Out-of-Order	no	yes	yes	yes	yes
	Speculative	no	yes	yes	yes	yes
	Reorder Before Write	no	yes	yes	yes	yes
Write	Out-of-Order	no	yes	no	no	no
	Speculative	no	no	no	no	no
	Buffering	no	yes	yes	yes	yes
	Combining[1]	no	yes	no	yes	yes

Note:
1. *Write-combining buffers are separate from write (store) buffers.*

Table 7-2. Caching Policy by Memory Type

Caching Policy	Memory Type					
	UC	CD	WC	WP	WT	WB
Read Cacheable	no	no	no	yes	yes	yes
Write Cacheable	no	no	no	no	yes	yes
Read Allocate	no	no	no	yes	yes	yes
Write Allocate	no	no	no	no	no	yes
Write Hits Update Memory	yes[2]	yes[1]	yes[2]	yes[3]	yes	no

Note:
1. *For the L1 data cache and the L2 cache, if an access hits the cache, the cache line is invalidated. If the cache line is in the modified state, the line is written to main memory and then invalidated. For the L1 instruction cache, read (instruction fetch) hits access the cache rather than main memory.*
2. *The data is not cached, so a cache write hit cannot occur. However, memory is updated.*
3. *Write hits update memory and invalidate the cache line.*

7.4.1 Instruction Fetching from Uncacheable Memory

Instruction fetches from an uncacheable memory type (including those for the CD type which don't hit in the instruction cache) may read a naturally-aligned block of memory no larger than an instruction cache line that contains multiple instructions, and may or may not repeat reads of a given block in the course of extracting instructions from it. In general, the exact sequence of read accesses is not deterministic, regardless of instruction stream contents, aside from the following constraints:

- instruction fetching of branch targets from uncacheable memory will only be done non-speculatively

- sequential instruction fetching will not transition speculatively from a cacheable memory type to an uncacheable memory type
- sequential instruction fetching will not speculatively cross more than one 4KB page boundary

It is recommended that MMIO devices that have read side-effects be separated from memory that's subject to uncacheable instruction fetches by at least one 4KB page.

7.4.2 Memory Barrier Interaction with Memory Types

Memory types other than WB may allow weaker ordering in certain respects. When the ordering of memory accesses to differing memory types must be strictly enforced, software can use the LFENCE, MFENCE or SFENCE barrier instructions to force loads and stores to proceed in program order. Table 7-3 on page 196 summarizes the cases where a memory barrier must be inserted between two memory operations.

The table is read as follows: the ROW is the first memory operation in program order, followed by the COLUMN, which is the second memory operation in program order. Each cell represents the ordered combination of the two memory operations and the letters *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*, and *l* within the cell represent the applicable memory ordering rule for that combination. These symbols are described in the footnotes below the table. In the table and footnotes, the abbreviation *nt* stands for non-temporal (load or store), *io* stands for input / output, *lf* for LFENCE, *sf* for SFENCE, and *mf* for MFENCE.

Table 7-3. Memory Access Ordering Rules

First Memory Operation	Second Memory Operation								
	Load (wp, wt, wb)	Load (uc)	Load (wc, wc+)	Store (wp, wt, wb)	Store (uc)	Store (wc, wc+, non-temporal)	Load/Store (io)	Lock (atomic)	Serialize instructions/ Interrupts/Exceptions
Load (wp, wt, wb)	a	f	b (lf)	c	c	c	d	d	d
Load (uc)	a	f	b (lf)	c	c	c	d	d	d
Load (wc, wc+)	a	f	b (lf)	c	c	c	d	d	d
Store (wp, wt, wb)	e (mf)	f	e (mf)	g	g	h (sf)	d	d	d
Store (uc)	i	f	i	g	g	h (sf)	d	d	d
Store (wc, wc+, non-temporal)	e (mf)	f	e (mf)	j (sf)	g, m	h (sf)	d	d	d
Load/Store (io)	k	k	k	k	k	l	d, k	d, k	d, k
Lock (atomic)	k	k	k	k	k	k	d, k	d, k	d, k
Serialize instruction/ Interrupts/Exceptions	l	l	l	l	l	l	d, l	d, l	d, l

a — A load (wp, wt, wb) may not pass a previous load (wp, wt, wb, wc, wc+, uc).

b — A load (wc, wc+) may pass a previous load (wp, wt, wb, wc, wc+). To ensure memory order, an LFENCE instruction must be inserted between the two loads.

c — A store (wp, wt, wb, uc, wc, wc+, nt) may not pass a previous load (wp, wt, wb, uc, wc, wc+, nt).

d — All previous loads and stores complete to memory or I/O space before a memory access for an I/O, locked or serializing instruction is issued.

e — A load (wp, wt, wb, wc, wc+) may pass a previous non-conflicting store (wp, wt, wb, wc, wc+, nt). To ensure memory order, an MFENCE instruction must be inserted between the store and the load.

f — A load or store (uc) does not pass a previous load or store (wp, wt, wb, uc, wc, wc+, nt).

g — A store (wp, wt, wb, uc) does not pass a previous store (wp, wt, wb, uc).

h — A store (wc, wc+, nt) may pass a previous store (wp, wt, wb) or non-conflicting store (wc, wc+, nt). To ensure memory order, an SFENCE instruction must be inserted between these two stores. A store (wc, wc+, nt) does not pass a previous conflicting store (wc, wc+, nt, uc).

i — A load (wp, wt, wb, wc, wc+) may pass a previous non-conflicting store (uc). To ensure memory order, an MFENCE instruction must be inserted between the store and the load.

j — A store (wp, wt, wb) may pass a previous store (wc, wc+, nt). To ensure memory order, an SFENCE instruction must be inserted between these two stores.

k — All loads and stores associated with the I/O and locked instructions complete to memory (no buffered stores) before a load or store from a subsequent instruction is issued.

l — All loads and stores complete to memory for the serializing instruction before the subsequent instruction fetch is issued.

m — A store (uc) does not pass a previous store (wc, wc+).

7.5 Buffering and Combining Memory Writes

7.5.1 Write Buffering

Writes to memory (main memory and caches) can be stored internally by the processor in *write buffers* (also known as store buffers) before actually writing the data into a memory location. System performance can be improved by buffering writes, as shown in the following examples:

- When higher-priority memory transactions, such as reads, compete for memory access with writes, writes can be delayed in favor of reads, which minimizes or eliminates an instruction-execution stall due to a memory-operand read.
- When the memory is busy, buffering writes while the memory is busy removes the writes from the instruction-execution pipeline, which frees instruction-execution resources.

The processor manages the write buffer so that it is transparent to software. Memory accesses check the write buffer, and the processor completes writes into memory from the buffer in program order. Also, the processor completely empties the write buffer by writing the contents to memory as a result of performing any of the following operations:

- *SFENCE Instruction*—Executing a store-fence (SFENCE) instruction forces all memory writes before the SFENCE (in program order) to be written into memory (or, for WB type, the cache) before memory writes that follow the SFENCE instruction. The memory-fence (MFENCE) instruction has a similar effect, but it forces the ordering of loads in addition to stores.
- *Serializing Instructions*—Executing a serializing instruction forces the processor to retire the serializing instruction (complete both instruction execution and result writeback) before the next instruction is fetched from memory.
- *I/O instructions*—Before completing an I/O instruction, all previous reads and writes must be written to memory, and the I/O instruction must complete before completing subsequent reads or writes. Writes to I/O-address space (OUT instruction) are never buffered.
- *Locked Instructions*—A locked instruction (an instruction executed using the LOCK prefix) or an XCHG instruction (which is implicitly locked) must complete *after* all previous reads and writes and *before* subsequent reads and writes. Locked writes are never buffered, although locked reads and writes are cacheable.
- *Interrupts and Exceptions*—Interrupts and exceptions, including virtualization intercepts (#VMEXIT), are serializing events that force the processor to write all results from the write buffer to memory before fetching the first instruction from the interrupt or exception service routine.
- *UC Memory Reads*—UC memory reads are not reordered ahead of writes.

Write buffers can behave similarly to *write-combining buffers* because multiple writes may be collected internally before transferring the data to caches or main memory. See the following section for a description of write combining.

7.5.2 Write Combining

Write-combining memory uses a different buffering scheme than write buffering described above. Writes to write-combining (WC) memory can be combined internally by the processor in a buffer for more efficient transfer to main memory at a later time. For example, 16 doubleword writes to consecutive memory addresses can be combined in the WC buffers and transferred to main memory as a single burst operation rather than as individual memory writes.

The following instructions perform writes to WC memory:

- (V)MASKMOVDQU
- MASKMOVQ
- (V)MOVNTDQ
- MOVNTI
- (V)MOVNTPD
- (V)MOVNTPS
- MOVNTQ
- MOVNTSD
- MOVNTSS

WC memory is not cacheable. A WC buffer writes its contents only to main memory.

The size and number of WC buffers available is implementation dependent. The processor assigns an address range to an empty WC buffer when a WC-memory write occurs. The size and alignment of this address range is equal to the buffer size. All subsequent writes to WC memory that fall within this address range can be stored by the processor in the WC-buffer entry until an event occurs that causes the processor to write the WC buffer to main memory. After the WC buffer is written to main memory, the processor can assign a new address range on a subsequent WC-memory write.

Writes to consecutive addresses in WC memory are not required for the processor to combine them. The processor combines any WC memory write that falls within the active-address range for a buffer. Multiple writes to the same address overwrite each other (in program order) until the WC buffer is written to main memory.

It is possible for writes to proceed out of program order when WC memory is used. For example, a write to cacheable memory that follows a write to WC memory can be written into the cache before the WC buffer is written to main memory. For this reason, and the reasons listed in the previous paragraph, software that is sensitive to the order of memory writes should avoid using WC memory.

WC buffers are written to main memory under the same conditions as the write buffers, namely when:

- Executing a store-fence (SFENCE) instruction.

- Executing a serializing instruction.
- Executing an I/O instruction.
  - Executing an MMIO access (load or store to UC memory type)
- Executing a locked instruction (an instruction executed using the LOCK prefix).
- Executing an XCHG instruction
- An interrupt or exception occurs.

WC buffers are also written to main memory when:

- A subsequent non-write-combining operation has a write address that matches the WC-buffer active-address range.
- A write to WC memory falls outside the WC-buffer active-address range. The existing buffer contents are written to main memory, and a new address range is established for the latest WC write.

7.6 Memory Caches

The AMD64 architecture supports the use of internal and external caches. The size, organization, coherency mechanism, and replacement algorithm for each cache is implementation dependent. Generally, the existence of the caches is transparent to both application and system software. In some cases, however, software can use cache-structure information to optimize memory accesses or manage memory coherency. Such software can use the extended-feature functions of the CPUID instruction to gather information on the caching subsystem supported by the processor. For more information, see Section 3.3, “Processor Feature Identification,” on page 71.

7.6.1 Cache Organization and Operation

Although the detailed organization of a processor cache depends on the implementation, the general constructs are similar. L1 caches—data and instruction, or unified—and L2 caches usually are implemented as n-way set-associative caches. Figure 7-3 on page 200 shows a typical *logical* organization of an n-way set-associative cache. The physical implementation of the cache can be quite different.

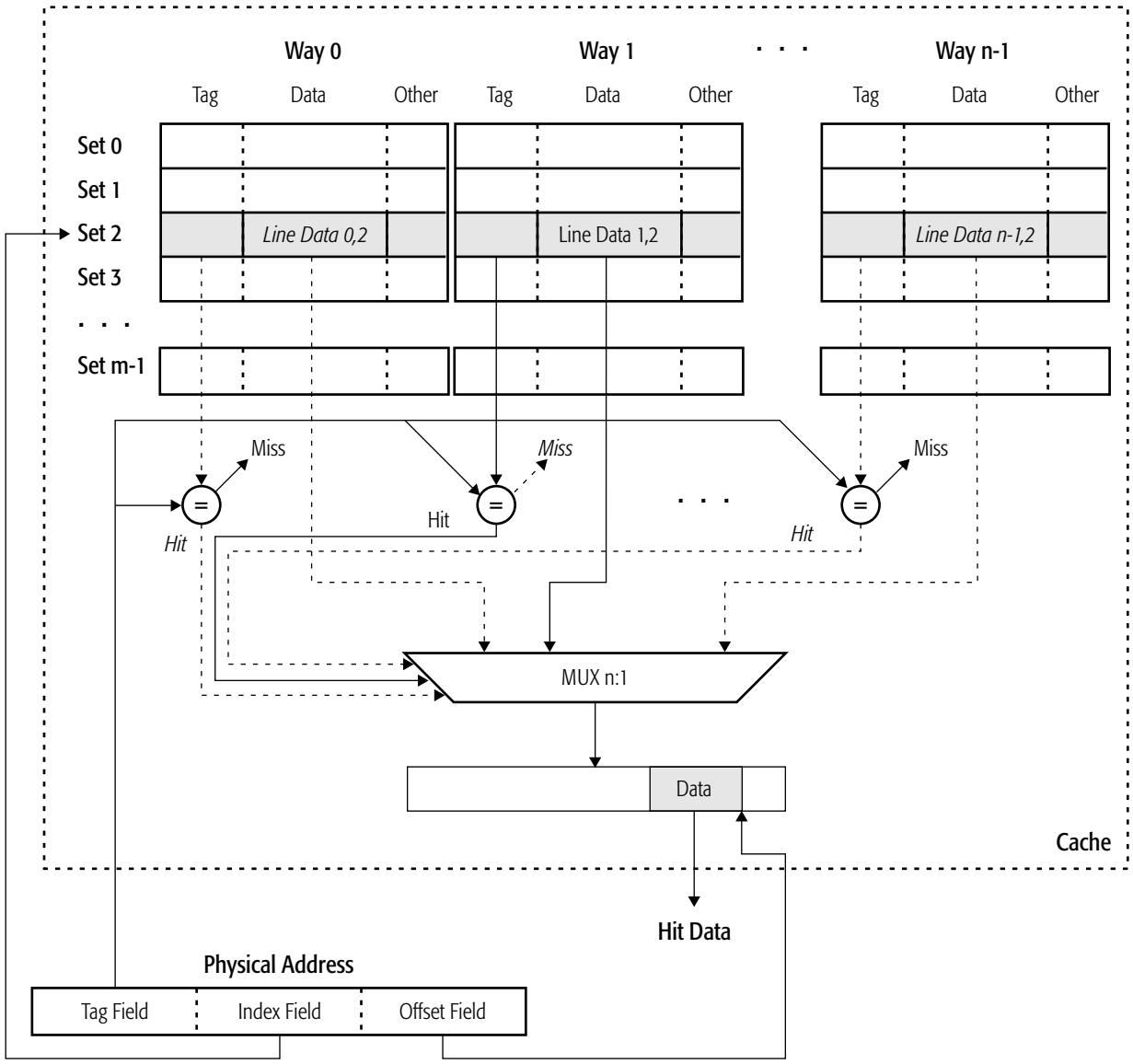


Figure 7-3. Cache Organization Example

As shown in Figure 7-3, the cache is organized as an array of cache lines. Each cache line consists of three parts: a cache-data line (a fixed-size copy of a memory block), a tag, and other information. Rows of cache lines in the cache array are *sets*, and columns of cache lines are *ways*. In an n-way set-associative cache, each set is a collection of n lines. For example, in a four-way set-associative cache, each set is a collection of four cache lines, one from each way.

The cache is accessed using the physical address of the data or instruction being referenced. To access data within a cache line, the physical address is used to select the set, way, and byte from the cache. This is accomplished by dividing the physical address into the following three fields:

- *Index*—The *index field* selects the cache set (row) to be examined for a hit. All cache lines within the set (one from each way) are selected by the index field.
- *Tag*—The *tag field* is used to select a specific cache line from the cache set. The physical-address tag field is compared with each cache-line tag in the set. If a match is found, a cache hit is signalled, and the appropriate cache line is selected from the set. If a match is not found, a cache miss is signalled.
- *Offset*—The *offset field* points to the first byte in the cache line corresponding to the memory reference. The referenced data or instruction value is read from (or written to, in the case of memory writes) the selected cache line starting at the location selected by the offset field.

In Figure 7-3 on page 200, the physical-address index field is shown selecting Set 2 from the cache. The tag entry for each cache line in the set is compared with the physical-address tag field. The tag entry for Way 1 matches the physical-address tag field, so the cache-line data for Set 2, Way 1 is selected using the n:1 multiplexor. Finally, the physical-address offset field is used to point to the first byte of the referenced data (or instruction) in the selected cache line.

Cache lines can contain other information in addition to the data and tags, as shown in Figure 7-3 on page 200. MOESI state and the state bits associated with the cache-replacement algorithm are typical pieces of information kept with the cache line. Instruction caches can also contain pre-decode or branch-prediction information. The type of information stored with the cache line is implementation dependent.

Self-Modifying Code. Software that stores into its own pending instruction stream with the intent of then executing the modified instructions is classified as self-modifying code. To support self-modifying code, AMD64 processors will flush any lines from the instruction cache that such stores hit, and will additionally check whether an instruction being modified is already in decode or execution behind the store instruction. If so, it will flush the pipeline and restart instruction fetch to acquire and re-decode the updated instruction bytes. No special action is needed by software for such updates to be immediately recognized. As with cache coherency, the check for instructions that are in flight is performed using physical addresses to avoid aliasing issues that could arise with virtual (linear) addresses.

When the modified bytes are in cacheable memory, the data cache may retain a copy of the modified cache line in a shared state, and the instruction cache refill may be satisfied from any suitable place in the memory hierarchy in a model-dependent manner that maintains cache coherency.

Cross-Modifying Code. Software that stores into the active instruction stream of another executing thread with the intent that the other thread subsequently execute the modified instruction stream is classified as cross-modifying code. There are two approaches to consider: asynchronous modification and synchronous modification.

Asynchronous modification. This is done with a write to the target instruction stream with no particular coordination being done between the writing and receiving threads. The nature of the code being executed by the target thread is such that it is insensitive to the exact timing of the update, for example executing in a known loop until an update to a branch instruction's offset takes it down a new path (or an update to an immediate operand, or opcode, or other instruction field). Such modifications must be done via a single store to the target thread's instruction stream that is contained entirely within a naturally-aligned quadword, and is subject to the constraints given here. A key aspect is that, although the store is performed atomically, the affected quadword may be read more than once in the process of extracting instruction bytes from it. This can result in the following scenarios resulting from a single store:

1. An update to two successive instructions, A and B, to A' and B' may result in execution of an A-B' sequence rather than A'-B'. However it will not result in an A'-B sequence since stores become visible to instruction fetchers in program order, and instruction fetchers read memory sequentially between taken branches.
2. A modification to one instruction A that changes it to two instructions A'-B will only result in execution of A'-B.
3. A modification to two instructions A-B that combines them into one instruction A' may result in a sequence of A-X, where X starts at the point in A' where B previously started.

Note that since stores to the instruction stream are observed by the instruction fetcher in program order, one can do multiple modifications to an area of the target thread's code that is beyond reach of the thread's current control flow, followed by a final asynchronous update that alters the control flow to expose the modified code to fetching and execution.

If the desired action cannot be achieved within these constraints, a synchronous modification approach must be used for reliable operation.

Synchronous modification. This entails a producer-consumer approach to the modification, where the target thread waits on a signal from the modifying thread, such as changing the state of a shared variable, before executing the modified code. The modifying thread writes to the target instruction bytes in any desired manner, then writes the synchronizing variable to release the target thread. Upon release, the target thread must then execute a serializing instruction such as CPUID or MFENCE (a locked operation is not sufficient) before proceeding to the modified code to avoid executing a stale view of the instructions which may have been speculatively fetched. Note that such speculative fetching is a function of branch predictor operation which is completely beyond the control of software.

See Volume 1, Chapter 3, “Semaphores,” for a discussion of instructions that are useful for interprocessor synchronization.

7.6.2 Cache Control Mechanisms

The AMD64 architecture provides a number of mechanisms for controlling the cacheability of memory. These are described in the following sections.

Cache Disable. Bit 30 of the CR0 register is the cache-disable bit, CR0.CD. Caching is enabled when CR0.CD is cleared to 0, and caching is disabled when CR0.CD is set to 1. When caching is disabled, reads and writes access main memory.

Software can disable the cache while the cache still holds valid data (or instructions). If a read or write hits the L1 data cache or the L2 cache when CR0.CD=1, the processor does the following:

1. Writes the cache line back if it is in the modified or owned state.
2. Invalidates the cache line.
3. Performs a non-cacheable main-memory access to read or write the data.

If an instruction fetch hits the L1 instruction cache when CR0.CD=1, some processor models may read the cached instructions rather than access main memory. When CR0.CD=1, the exact behavior of L2 and L3 caches is model-dependent, and may vary for different types of memory accesses.

The processor also responds to cache probes when CR0.CD=1. Probes that hit the cache cause the processor to perform Step 1. Step 2 (cache-line invalidation) is performed only if the probe is performed on behalf of a memory write or an exclusive read.

Writethrough Disable. Bit 29 of the CR0 register is the *not writethrough* disable bit, CR0.NW. In early x86 processors, CR0.NW is used to control cache writethrough behavior, and the combination of CR0.NW and CR0.CD determines the cache operating mode.

In early x86 processors, clearing CR0.NW to 0 enables writeback caching for main memory, effectively disabling writethrough caching for main memory. When CR0.NW=0, software can disable writeback caching for specific memory pages or regions by using other cache control mechanisms. When software sets CR0.NW to 1, writeback caching is disabled for main memory, while writethrough caching is enabled.

In implementations of the AMD64 architecture, CR0.NW is not used to qualify the cache operating mode established by CR0.CD. Table 7-4 shows the effects of CR0.NW and CR0.CD on the AMD64 architecture cache-operating modes.

Table 7-4. AMD64 Architecture Cache-Operating Modes

CR0.CD	CR0.NW	Cache Operating Mode
0	0	Cache enabled with a writeback-caching policy.
0	1	Invalid setting—causes a general-protection exception (#GP).
1	0	Cache disabled. See “Cache Disable” on page 203.
1	1	

Page-Level Cache Disable. Bit 4 of all paging data-structure entries controls page-level cache disable (PCD). When a data-structure-entry PCD bit is cleared to 0, the page table or physical page pointed to by that entry is cacheable, as determined by the CR0.CD bit. When the PCD bit is set to 1, the page table or physical page is not cacheable. The PCD bit in the paging data-structure base-register

(bit 4 in CR3) controls the cacheability of the highest-level page table in the page-translation hierarchy.

Page-Level Writethrough Enable. Bit 3 of all paging data-structure entries is the page-level writethrough enable control (PWT). When a data-structure-entry PWT bit is cleared to 0, the page table or physical page pointed to by that entry has a writeback caching policy. When the PWT bit is set to 1, the page table or physical page has a writethrough caching policy. The PWT bit in the paging data-structure base-register (bit 3 in CR3) controls the caching policy of the highest-level page table in the page-translation hierarchy.

The corresponding PCD bit must be cleared to 0 (page caching enabled) for the PWT bit to have an effect.

Memory Typing. Two mechanisms are provided for software to control access to and cacheability of specific memory regions:

- The memory-type range registers (MTRRs) control cacheability based on physical addresses. See “MTRRs” on page 209 for more information on the use of MTRRs.
- The page-attribute table (PAT) mechanism controls cacheability based on virtual addresses. PAT extends the capabilities provided by the PCD and PWT page-level cache controls. See “Page-Attribute Table Mechanism” on page 218 for more information on the use of the PAT mechanism.

System software can combine the use of both the MTRRs and PAT mechanisms to maximize control over memory cacheability.

If the MTRRs are disabled in implementations that support the MTRR mechanism, the default memory type is set to uncacheable (UC). Memory accesses are not cached even if the caches are enabled by clearing CR0.CD to 0. Cacheable memory types must be established using the MTRRs in order for memory accesses to be cached.

Cache Control Precedence. The cache-control mechanisms are used to define the memory type and cacheability of main memory and regions of main memory. Taken together, the most restrictive memory type takes precedence in defining the caching policy of memory. The order of precedence is:

1. Uncacheable (UC)
2. Write-combining (WC)
3. Write-protected (WP)
4. Writethrough (WT)
5. Writeback (WB)

For example, assume a large memory region is designated a writethrough type using the MTRRs. Individual pages within that region can have caching disabled by setting the appropriate page-table PCD bits. However, no pages within that region can have a writeback caching policy, regardless of the page-table PWT values.

7.6.3 Cache and Memory Management Instructions

Data Prefetch. The prefetch instructions are used by software as a hint to the processor that the referenced data is likely to be used in the near future. The processor can preload the cache line containing the data in anticipation of its use. PREFETCH provides a hint that the data is to be read. PREFETCHW provides a hint that the data is to be written. The processor can mark the line as modified if it is preloaded using PREFETCHW.

Memory Ordering. Instructions are provided for software to enforce memory ordering (serialization) in weakly-ordered memory types. These instructions are:

- *SFENCE (store fence)*—forces all memory writes (stores) preceding the SFENCE (in program order) to be written into memory before memory writes following the SFENCE.
- *LFENCE (load fence)*—forces all memory reads (loads) preceding the LFENCE (in program order) to be read from memory before memory reads following the LFENCE. In some systems, LFENCE may be configured to be dispatch serializing. In systems where CPUID Fn8000_0021_EAX[LFenceAlwaysSerializing] (bit 2) = 1, LFENCE is always dispatch serializing.
- *MFENCE (memory fence)*—forces all memory accesses (reads and writes) preceding the MFENCE (in program order) to be written into or read from memory before memory accesses following the MFENCE.

Cache Line Writeback and Flush. The CLFLUSH instruction (writeback, if modified, and invalidate) takes the byte memory-address operand (a linear address), and checks to see if the address is cached. If the address is cached, the entire cache line containing the address is invalidated. If any portion of the cache line is dirty (in the modified or owned state), the entire line is written to main memory before it is invalidated. CLFLUSH affects *all caches* in the memory hierarchy—internal and external to the processor, and across all cores. The CLWB instruction operates in the same manner except it does not invalidate the cache line. The checking and invalidation process continues until the address has been updated in memory and, for CLFLUSH, invalidated in all caches.

In most cases, the underlying memory type assigned to the address has no effect on the behavior of this instruction. However, when the underlying memory type for the address is UC or WC (as defined by the MTRRs), the processor does not proceed with checking all caches to see if the address is cached. In both cases, the address is uncacheable, and invalidation is unnecessary. Write-combining buffers are written back to memory if the corresponding physical address falls within the buffer active-address range.

Cache Writeback and Invalidate. Unlike the CLFLUSH and CLWB instructions, the WBINVD and WBNOINVD instructions operate on the entire cache, rather than a single cache line. The WBINVD and WBNOINVD instructions first write back all cache lines that are dirty (in the modified or owned state) to main memory. After writeback is complete, the WBINVD instruction additionally invalidates all cache lines. The checking and invalidation process continues until all internal caches in the executing core's path to system memory are invalidated. In some implementations this may include caches in other branches of the system's cache hierarchy; see the description of these instructions in

volume 3 for more detail. For either instruction, a special bus cycle is transmitted to higher-level external caches directing them to perform a writeback-and-invalidate operation.

Cache Invalidate. The INVD instruction is used to invalidate all cache lines. Unlike the WBINVD instruction, dirty cache lines are not written to main memory. The process continues until all internal caches have been invalidated. A special bus cycle is transmitted to higher-level external caches directing them to perform an invalidation.

The INVD instruction should only be used in situations where memory coherency is not required.

7.6.4 Serializing Instructions

Serializing instructions force the processor to retire the serializing instruction and all previous instructions before the next instruction is fetched. A serializing instruction is retired when the following operations are complete:

- The instruction has executed.
- All registers modified by the instruction are updated.
- All memory updates performed by the instruction are complete.
- All data held in the write buffers have been written to memory.

Serializing instructions can be used as a barrier between memory accesses to force strong ordering of memory operations. Care should be exercised in using serializing instructions because they modify processor state and may affect program flow. The instructions also force execution serialization, which can significantly degrade performance. When strongly-ordered memory accesses are required, but execution serialization is not, it is recommended that software use the memory-ordering instructions described on page 205.

The following are serializing instructions:

- *Non-Privileged Instructions*
  - CPUID
  - IRET
  - RSM
  - MFENCE
- *Privileged Instructions*
  - MOV CRn
  - MOV DRn
  - LGDT, LIDT, LLDT, LTR
  - WRMSR (see note 1)
  - WBINVD, WBNOINVD, INVD
  - INVLPG

Note 1: *Writes to the following MSRs are not serializing:* SPEC_CTRL, PRED_CMD, all x2APIC MSRs.

A dispatch serializing instruction is a lighter form of ordering than a serializing instruction. A dispatch serializing instruction forces the processor to retire the serializing instruction and all previous instructions before the next instruction is executed. In some systems, LFENCE may be configured to be dispatch serializing. In systems where CPUID Fn8000_0021_EAX[LFenceAlwaysSerializing](bit 2) = 1, LFENCE is always dispatch serializing.

7.6.5 Cache and Processor Topology

Cache and processor topology information is useful in the optimal management of system and application resources. Exposing processor and cache topology information to the programmer allows software to make more efficient use of hardware multithreading resources delivering optimal performance. Shared resources in a specific cache and processor topology may require special consideration in the optimization of multiprocessing software performance.

The processor topology allows software to determine which cores or logical processors are siblings in a compute unit, node, and processor package. For example, a scheduler can then choose to either compact or scatter threads (or processes) to cores in compute units, nodes, or across the cores in the entire physical package in order to optimize for a power and performance profile.

Topology extensions define processor topology at both the node, compute unit and cache level. Topology extensions include cache properties with sharing and the processor topology identified. The result is a simplified extension to the CPUID instruction that describes the processors cache topology and leverages existing industry cache properties folded into AMD’s topology extension description.

Topology extensions definition supports existing and future processors with varying degrees of cache level sharing. Topology extensions also support the description of a simple compute unit with one core or packages where the number of cores in a node and/or compute unit are not an even power of two.

CPUID Function 8000_001D: Cache Topology Definition. CPUID Function 8000_001D describes the hierarchical relationships of cache levels relative to the cores which share these resources. Function 8000_001D is defined to be called iteratively with the value 8000001Dh in EAX and an additional parameter in ECX. To gather information for all cache levels, software must call CPUID with 8000001Dh in EAX and ECX set to increasing values beginning with 0 until a value of 0 is returned from EAX[4:0], which indicates no more cache descriptions.

If software dynamically manages cache configuration, it will need to update any stored cache properties for the processor.

CPUID Function 8000_001E: Processor Topology Definition. CPUID Function 8000_001E describes processor topology with component identifiers. To read the processor topology, definition software calls the CPUID instruction with the value 8000001Eh in EAX. After execution the APIC ID is represented in EAX. EBX contains the compute unit description in the processor, while ECX contains system unique node identification. Software may read this information once for each core.

The following CPUID functions provide information about processor topology:

- CPUID Fn8000_0001_ECX
- CPUID Fn8000_0008_ECX
- CPUID Fn8000_001D_EAX, EBX, ECX, EDX
- CPUID Fn8000_001E_EAX, EBX, ECX

For more information using the CPUID instruction, see Section 3.3, “Processor Feature Identification,” on page 71.

7.7 Memory-Type Range Registers

The AMD64 architecture supports three mechanisms for software access-control and cacheability-control over memory regions. These mechanisms can be used in place of similar capabilities provided by external chipsets used with early x86 processors.

This section describes a control mechanism that uses a set of programmable model-specific registers (MSRs) called the *memory-type-range registers* (MTRRs). The MTRR mechanism provides system software with the ability to manage hardware-device memory mapping. System software can characterize physical-memory regions by type (e.g., ROM, flash, memory-mapped I/O) and assign hardware devices to the appropriate physical-memory type.

Another control mechanism is implemented as an extension to the page-translation capability and is called the *page attribute table* (PAT). It is described in “Page-Attribute Table Mechanism” on page 218. Like the MTRRs, PAT provides system software with the ability to manage hardware-device memory mapping. With PAT, however, system software can characterize physical pages and assign virtually-mapped devices to those physical pages using the page-translation mechanism. PAT may be used in conjunction with the MTTR mechanism to maximize flexibility in memory control.

Finally, control mechanisms are provided for managing memory-mapped I/O. These mechanisms employ extensions to the MTRRs and a separate feature called the *top-of-memory registers*. The MTRR extensions include additional MTRR type-field encodings for fixed-range MTRRs and variable-range I/O range registers (IORRs). These mechanisms are described in “Memory-Mapped I/O” on page 222.

7.7.1 MTRR Type Fields

The MTRR mechanism provides a means for associating a physical-address range with a memory type (see “Memory Types” on page 192). The MTRRs contain a type field used to specify the memory type in effect for a given physical-address range.

There are two variants of the memory type-field encodings: standard and extended. Both the standard and extended encodings use type-field bits 2:0 to specify the memory type. For the standard encodings, bits 7:3 are reserved and must be zero. For the extended encodings, bits 7:5 are reserved, but bits 4:3 are defined as the RdMem and WrMem bits. “Extended Fixed-Range MTRR Type-Field Encodings” on page 223 describes the function of these extended bits and how software enables them.

Only the fixed-range MTRRs support the extended type-field encodings. Variable-range MTRRs use the standard encodings.

Table 7-5 on page 209 shows the memory types supported by the MTRR mechanism and their encoding in the MTRR type fields referenced throughout this section. Unless the extended type-field encodings are explicitly enabled, the processor uses the type values shown in Table 7-5.

Table 7-5. MTRR Type Field Encodings

Type Value	Type Name	Type Description
00h	UC—Uncacheable	All accesses are uncacheable. Write combining is not allowed. Speculative accesses are not allowed
01h	WC—Write-Combining	All accesses are uncacheable. Write combining is allowed. Speculative reads are allowed
04h	WT—Writethrough	Reads allocate cache lines on a cache miss. Cache lines are not allocated on a write miss. Write hits update the cache and main memory.
05h	WP—Write-Protect	Reads allocate cache lines on a cache miss. All writes update main memory. Cache lines are not allocated on a write miss. Write hits invalidate the cache line and update main memory.
06h	WB—Writeback	Reads allocate cache lines on a cache miss, and can allocate to either the shared, exclusive, or modified state. Writes allocate to the modified state on a cache miss.

If the MTRRs are disabled in implementations that support the MTRR mechanism, the default memory type is set to uncacheable (UC). *Memory accesses are not cached even if the caches are enabled by clearing CR0.CD to 0.* Cacheable memory types must be established using the MTRRs to enable memory accesses to be cached.

7.7.2 MTRRs

Both fixed-size and variable-size address ranges are supported by the MTRR mechanism. The fixed-size ranges are restricted to the lower 1 Mbyte of physical-address space, while the variable-size ranges can be located anywhere in the physical-address space.

Figure 7-4 on page 210 shows an example mapping of physical memory using the fixed-size and variable-size MTRRs. The areas shaded gray are not mapped by the MTRRs. Unmapped areas are set to the software-selected default memory type.

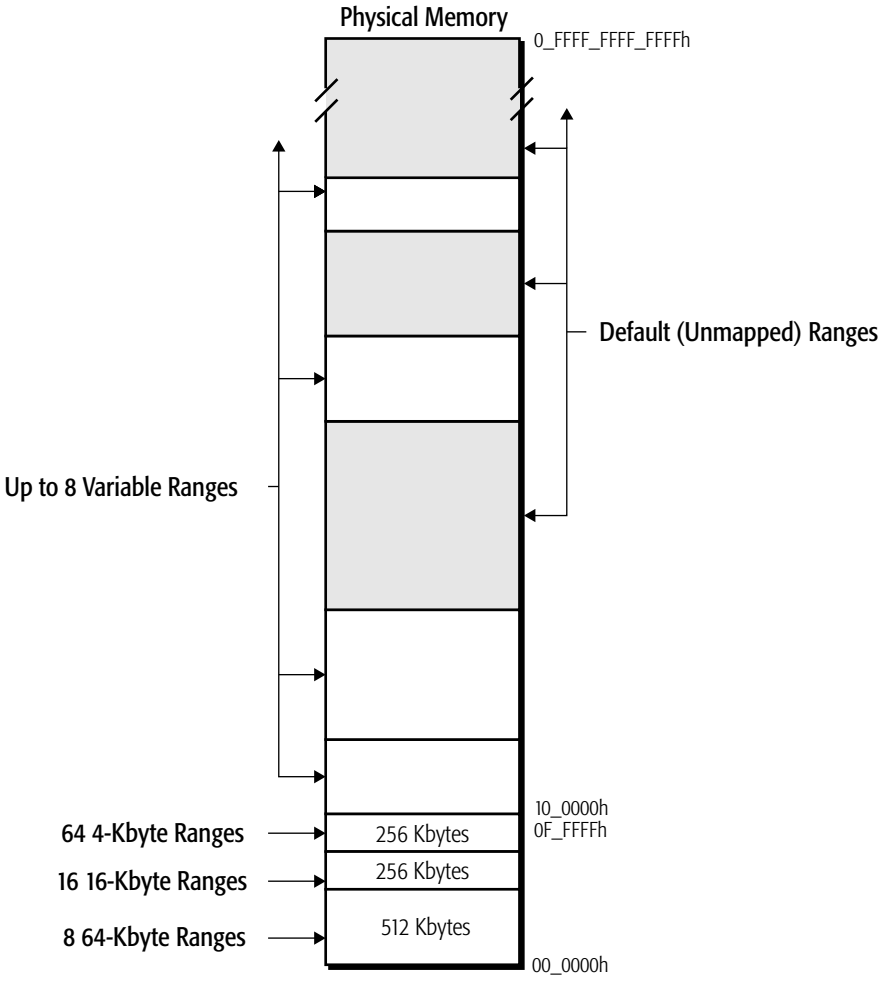


Figure 7-4. MTRR Mapping of Physical Memory

MTRRs are 64-bit model-specific registers (MSRs). They are read using the RDMSR instruction and written using the WRMSR instruction. See “Memory-Typing MSRs” on page 669 for a listing of the MTRR MSR numbers. The following sections describe the types of MTRRs and their function.

Fixed-Range MTRRs. The fixed-range MTRRs are used to characterize the first 1 Mbyte of physical memory. Each fixed-range MTRR contains eight type fields for characterizing a total of eight memory ranges. Fixed-range MTRRs support extended type-field encodings as described in “Extended Fixed-Range MTRR Type-Field Encodings” on page 223. The extended type field allows a fixed-range MTRR to be used as a fixed-range IORR. Figure 7-5 on page 211 shows the format of a fixed-range MTRR.

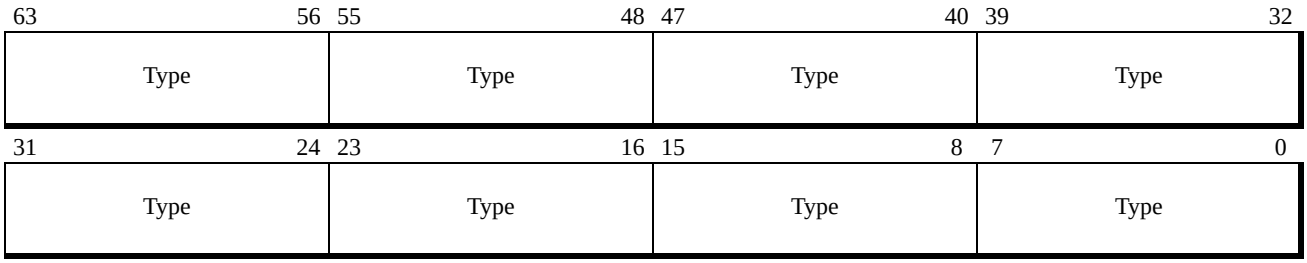


Figure 7-5. Fixed-Range MTRR

For the purposes of memory characterization, the first 1 Mbyte of physical memory is segmented into a total of 88 non-overlapping memory ranges, as follows:

- The 512 Kbytes of memory spanning addresses 00_0000h to 07_FFFFh are segmented into eight 64-Kbyte ranges. A single MTRR is used to characterize this address space.
- The 256 Kbytes of memory spanning addresses 08_0000h to 0B_FFFFh are segmented into 16 16-Kbyte ranges. Two MTRRs are used to characterize this address space.
- The 256 Kbytes of memory spanning addresses 0C_0000h to 0F_FFFFh are segmented into 64 4-Kbyte ranges. Eight MTRRs are used to characterize this address space.

Table 7-6 shows the address ranges corresponding to the type fields within each fixed-range MTRR. The gray-shaded heading boxes represent the bit ranges for each type field in a fixed-range MTTR. See Table 7-5 on page 209 for the type-field encodings.

Table 7-6. Fixed-Range MTRR Address Ranges

Physical Address Range (in hexadecimal)								Register Name
63–56	55–48	47–40	39–32	31–24	23–16	15–8	7–0	
70000–7FFFF	60000–6FFFF	50000–5FFFF	40000–4FFFF	30000–3FFFF	20000–2FFFF	10000–1FFFF	00000–0FFFF	**MTRRfix64K_00000**
9C000–9FFFF	98000–9BFFF	94000–97FFF	90000–93FFF	8C000–8FFFF	88000–8BFFF	84000–87FFF	80000–83FFF	**MTRRfix16K_80000**
BC000–BFFFF	B8000–BBFFF	B4000–B7FFF	B0000–B3FFF	AC000–AFFFF	A8000–ABFFF	A4000–A7FFF	A0000–A3FFF	**MTRRfix16K_A0000**
C7000–C7FFF	C6000–C6FFF	C5000–C5FFF	C4000–C4FFF	C3000–C3FFF	C2000–C2FFF	C1000–C1FFF	C0000–C0FFF	**MTRRfix4K_C0000**
CF000–CFFFF	CE000–CEFFF	CD000–CDFFF	CC000–CCFFF	CB000–CBFFF	CA000–CAFFF	C9000–C9FFF	C8000–C8FFF	**MTRRfix4K_C8000**
D7000–D7FFF	D6000–D6FFF	D5000–D5FFF	D4000–D4FFF	D3000–D3FFF	D2000–D2FFF	D1000–D1FFF	D0000–D0FFF	**MTRRfix4K_D0000**
DF000–DFFFF	DE000–DEFFF	DD000–DDFFF	DC000–DCFFF	DB000–DBFFF	DA000–DAFFF	D9000–D9FFF	D8000–D8FFF	**MTRRfix4K_D8000**
E7000–E7FFF	E6000–E6FFF	E5000–E5FFF	E4000–E4FFF	E3000–E3FFF	E2000–E2FFF	E1000–E1FFF	E0000–E0FFF	**MTRRfix4K_E0000**

Table 7-6. Fixed-Range MTRR Address Ranges (continued)

Physical Address Range (in hexadecimal)								Register Name
63–56	55–48	47–40	39–32	31–24	23–16	15–8	7–0	
EF000–EFFFF	EE000–EEFFF	ED000–EDFFF	EC000–ECFFF	EB000–EBFFF	EA000–EAFFF	E9000–E9FFF	E8000–E8FFF	**MTRRfix4K_E8000**
F7000–F7FFF	F6000–F6FFF	F5000–F5FFF	F4000–F4FFF	F3000–F3FFF	F2000–F2FFF	F1000–F1FFF	F0000–F0FFF	**MTRRfix4K_F0000**
FF000–FFFFF	FE000–FEFFF	FD000–FDFFF	FC000–FCFFF	FB000–FBFFF	FA000–FAFFF	F9000–F9FFF	F8000–F8FFF	**MTRRfix4K_F8000**

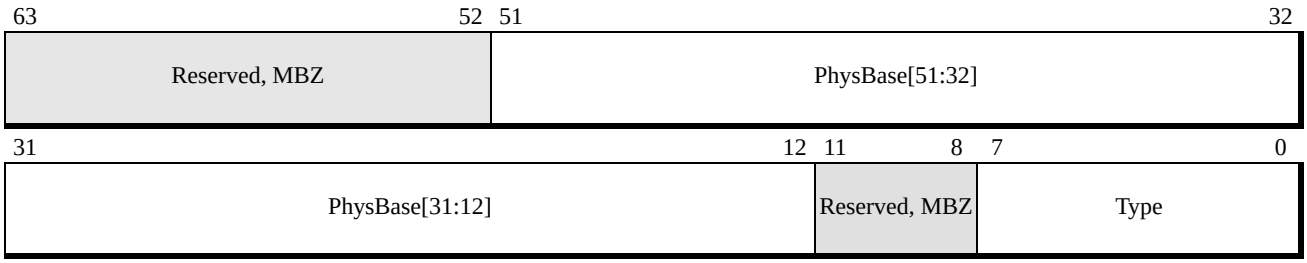
Variable-Range MTRRs. The variable-range MTRRs can be used to characterize any address range within the physical-memory space, including all of physical memory. Up to eight address ranges of varying sizes can be characterized using the MTRR. Two variable-range MTRRs are used to characterize each address range: MTRRphysBase*n* and MTRRphysMask*n* (*n* is the address-range number from 0 to 7). For example, address-range 3 is characterized using the MTRRphysBase3 and MTRRphysMask3 register pair.

Figure 7-6 shows the format of the MTRRphysBase*n* register and Figure 7-7 on page 213 shows the format of the MTRRphysMask*n* register. The fields within the register pair are read/write.

MTRRphysBase*n* Registers. The fields in these variable-range MTRRs, shown in Figure 7-6, are:

- *Type*—Bits 7:0. The memory type used to characterize the memory range. See Table 7-5 on page 209 for the type-field encodings. Variable-range MTRRs do not support the extended type-field encodings.
- *Range Physical Base-Address (PhysBase)*—Bits 51:12. The memory-range base-address in physical-address space. PhysBase is aligned on a 4-Kbyte (or greater) address in the 52-bit physical-address space supported by the AMD64 architecture. PhysBase represents the most-significant 40-address bits of the physical address. Physical-address bits 11:0 are assumed to be 0.

Note that a given processor may implement less than the architecturally-defined physical address size of 52 bits.

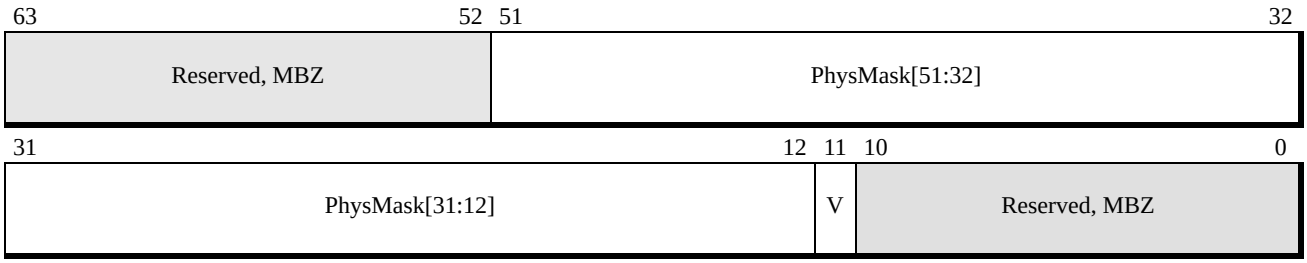


Bits	Mnemonic	Description	Access type
63:52	—	Reserved	MBZ
51:12	PhysBase	Range Physical Base Address	R/W
11:8	—	Reserved	MBZ
7:0	Type	Default Memory Type	R/W

Figure 7-6. MTRRphysBase*n* Register

MTRRphysMask*n* Registers. The fields in these variable-range MTRRs, shown in Figure 7-7, are:

- *Valid (V)*—Bit 11. Indicates that the MTRR pair is valid (enabled) when set to 1. When the valid bit is cleared to 0 the register pair is not used.
- *Range Physical Mask (PhysMask)*—Bits 51:12. The mask value used to specify the memory range. Like PhysBase, PhysMask is aligned on a 4-Kbyte physical-address boundary. Bits 11:0 of PhysMask are assumed to be 0.



Bits	Mnemonic	Description	Access type
63:52	—	Reserved	MBZ
51:12	PhysMask	Range Physical Mask	R/W
11	V	MTRR Pair Enable (Valid)	R/W
10:0	—	Reserved	MBZ

Figure 7-7. MTRRphysMask*n* Register

PhysMask and PhysBase are used together to determine whether a target physical-address falls within the specified address range. PhysMask is logically ANDed with PhysBase and separately ANDed with the upper 40 bits of the target physical-address. If the results of the two operations are identical, the target physical-address falls within the specified memory range. The pseudo-code for the operation is:

```
MaskBase = PhysMask AND PhysBase
MaskTarget = PhysMask AND Target_Address[51:12]
IF MaskBase == MaskTarget
   target address is in range
ELSE
   target address is not in range
```

Variable Range Size and Alignment. The size and alignment of variable memory-ranges (MTRRs) and I/O ranges (IORRs) are restricted as follows:

- The boundary on which a variable range is aligned must be equal to the range size. For example, a memory range of 16 Mbytes must be aligned on a 16-Mbyte boundary.
- The range size must be a power of 2 (2^n, 52 > n > 11), with a minimum allowable size of 4 Kbytes. For example, 4 Mbytes and 8 Mbytes are allowable memory range sizes, but 6 Mbytes is not allowable.

PhysMask and PhysBase Values. Software can calculate the PhysMask value using the following procedure:

1. Subtract the memory-range physical base-address from the upper physical-address of the memory range.
2. Subtract the value calculated in Step 1 from the physical memory size.
3. Truncate the lower 12 bits of the result in Step 2 to create the PhysMask value to be loaded into the MTRRphysMask*n* register. Truncation is performed by right-shifting the value 12 bits.

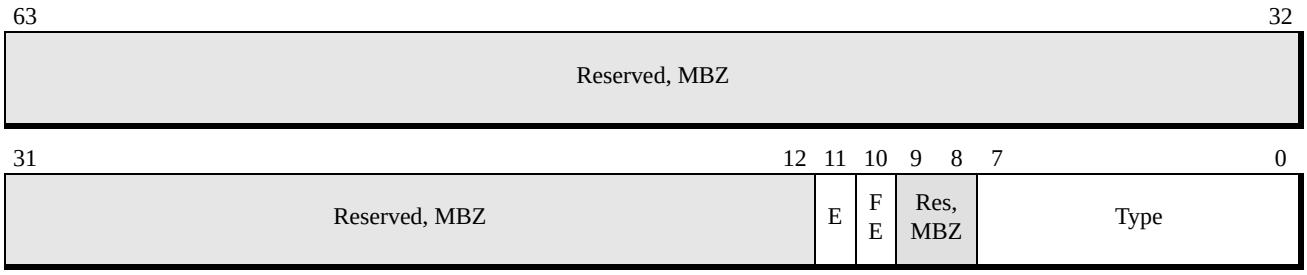
For example, assume a 32-Mbyte memory range is specified within the 52-bit physical address space, starting at address 200_0000h. The upper address of the range is 3FF_FFFFh. Following the process outlined above yields:

1. 3FF_FFFFh–200_0000h = 1FF_FFFFh
2. F_FFFF_FFFF_FFFF–1FF_FFFFh = F_FFFF_FE00_0000h
3. Right shift (F_FFFF_FE00_0000h) by 12 = FF_FFFF_E000h

In this example, the 40-bit value loaded into the PhysMask field is FF_FFFF_E000h.

Software must also truncate the lower 12 bits of the physical base-address before loading it into the PhysBase field. In the example above, the 40-bit PhysBase field is 00_0000_2000h.

Default-Range MTRRs. Physical addresses that are not within ranges established by fixed-range and variable-range MTRRs are set to a default memory-type using the MTRRdefType register. The format of this register is shown in Figure 7-8.



Bits	Mnemonic	Description	Access type
63:12	—	Reserved	MBZ
11	E	MTRR Enable	R/W
10	FE	Fixed Range Enable	R/W
9:8	—	Reserved	MBZ
7:0	Type	Default Memory Type	R/W

Figure 7-8. MTRRdefType Register Format

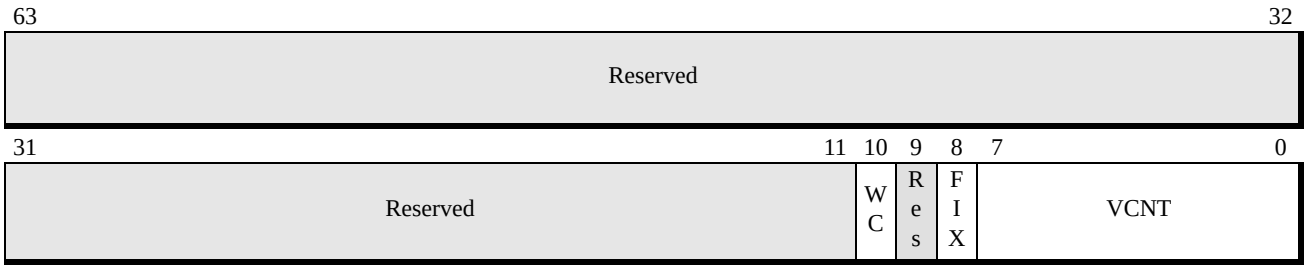
The fields within the MTRRdefType register are read/write. These fields are:

- *Type*—Bits 7:0. The default memory-type used to characterize physical-memory space. See Table 7-5 on page 209 for the type-field encodings. The extended type-field encodings are not supported by this register.
- *Fixed-Range Enable (FE)*—Bit 10. All fixed-range MTRRs are enabled when FE is set to 1. Clearing FE to 0 disables all fixed-range MTRRs. Setting and clearing FE has no effect on the variable-range MTRRs. The FE bit has no effect unless the E bit is set to 1 (see below).
- *MTRR Enable (E)*—Bit 11. This is the MTRR memory typing enable bit. The memory typing capabilities of all fixed-range and variable-range MTRRs are enabled when E is set to 1. Clearing E to 0 disables the memory typing capabilities of all fixed-range and variable-range MTRRs and sets the default memory-type to uncacheable (UC) regardless of the value of the Type field. This bit does not affect the operation of the RdMem and WrMem fields.

7.7.3 Using MTRRs

Identifying MTRR Features. Software determines whether a processor supports the MTRR mechanism by executing the CPUID instruction with either function 0000_0001h or function 8000_0001h. If MTRRs are supported, bit 12 in the EDX register is set to 1 by CPUID. See “Processor Feature Identification” on page 71 for more information on the CPUID instruction.

The MTRR capability register (MTRRcap) is a read-only register containing information describing the level of MTRR support provided by the processor. Figure 7-9 shows the format of this register. If MTRRs are supported, software can read MTRRcap using the RDMSR instruction. Attempting to write to the MTRRcap register causes a general-protection exception (#GP).



Bits	Mnemonic	Description	Access type
63:11	—	Reserved	R
10	WC	Write Combining	R
9	—	Reserved	R
8	FIX	Fixed-Range Registers	R
7:0	VCNT	Variable-Range Register Count	R

Figure 7-9. MTRR Capability Register Format

The MTRRcap register field are:

- *Variable-Range Register Count (VCNT)*—Bits 7:0. The VCNT field contains the number of variable-range register pairs supported by the processor. For example, a processor supporting eight register pairs returns a 08h in this field.
- *Fixed-Range Registers (FIX)*—Bit 8. The FIX bit indicates whether or not the fixed-range registers are supported. If the processor returns a 1 in this bit, *all* fixed-range registers are supported. If the processor returns a 0 in this bit, *no* fixed-range registers are supported.
- *Write-Combining (WC)*—Bit 10. The WC bit indicates whether or not the write-combining memory type is supported. If the processor returns a 1 in this bit, WC memory is supported, otherwise it is not supported.

7.7.4 MTRRs and Page Cache Controls

When paging and the MTRRs are both enabled, the address ranges defined by the MTRR registers can span multiple pages, each of which can characterize memory with different types (using the PCD and PWT page bits). When caching is enabled (CR0.CD=0 and CR0.NW=0), the *effective memory type* is determined as follows:

1. If the page is defined as cacheable and writeback (PCD=0 and PWT=0), then the MTRR defines the effective memory-type.
2. If the page is defined as not cacheable (PCD=1), then UC is the effective memory-type.
3. If the page is defined as cacheable and writethrough (PCD=0 and PWT=1), then the MTRR defines the effective memory-type *unless* the MTRR specifies WB memory, in which case WT is the effective memory-type.

Table 7-7 lists the MTRR and page-level cache-control combinations and their combined effect on the final memory-type, if the PAT register holds the default settings.

Table 7-7. Combined MTRR and Page-Level Memory Type with Unmodified PAT MSR

MTRR Memory Type	Page PCD Bit	Page PWT Bit	Effective Memory-Type
UC	—	—	UC
WC	0	—	WC
	1	0	WC[1]
	1	1	UC
WP	0	—	WP
	1	—	UC
WT	0	—	WT
	1	—	UC
WB	0	0	WB
	0	1	WT
	1	—	UC

Note:
1. The effective memory-type resulting from the combination of PCD=1, PWT=0, and an MTRR WC memory type is implementation dependent.

Large Page Sizes. When paging is enabled, software can use large page sizes (2 Mbytes and 4 Mbytes) in addition to the more typical 4-Kbyte page size. When large page sizes are used, it is possible for multiple MTRRs to span the memory range within a single large page. Each MTRR can characterize the regions within the page with different memory types. If this occurs, the effective memory-type used by the processor within the large page is undefined.

Software can avoid the undefined behavior in one of the following ways:

- Avoid using multiple MTRRs to characterize a single large page.
- Use multiple 4-Kbyte pages rather than a single large page.
- If multiple MTRRs must be used within a single large page, software can set the MTRR type fields to the same value.
- If the multiple MTRRs must have different type-field values, software can set the large page PCD and PWT bits to the most restrictive memory type defined by the multiple MTRRs.

Overlapping MTRR Registers. If the address ranges of two or more MTRRs overlap, the following rules are applied to determine the memory type used to characterize the overlapping address range:

1. Fixed-range MTRRs, which characterize only the first 1 Mbyte of physical memory, have precedence over variable-range MTRRs.
2. If two or more variable-range MTRRs overlap, the following rules apply:

a. If the memory types are identical, then that memory type is used.

b. If at least one of the memory types is UC, the UC memory type is used.

c. If at least one of the memory types is WT, and the only other memory type is WB, then the WT memory type is used.

d. If the combination of memory types is not listed Steps A through C immediately above, then the memory type used is undefined.

7.7.5 MTRRs in Multi-Processing Environments

In multi-processing environments, the MTRRs located in all processors must characterize memory in the same way. Generally, this means that identical values are written to the MTRRs used by the processors. This also means that values CR0.CD and the PAT must be consistent across processors. Failure to do so may result in coherency violations or loss of atomicity. Processor implementations *do not* check the MTRR settings in other processors to ensure consistency. It is the responsibility of system software to initialize and maintain MTRR consistency across all processors.

7.8 Page-Attribute Table Mechanism

The page-attribute table (PAT) mechanism extends the page-table entry format and enhances the capabilities provided by the PCD and PWT page-level cache controls. PAT (and PCD, PWT) allow memory-type characterization based on the virtual (linear) address. The PAT mechanism provides the same memory-typing capabilities as the MTRRs but with the added flexibility of the paging mechanism. Software can use both the PAT and MTRR mechanisms to maximize flexibility in memory-type control.

7.8.1 PAT Register

Like the MTRRs, the PAT register is a 64-bit model-specific register (MSR). The format of the PAT registers is shown in Figure 7-10. See “Memory-Typing MSRs” on page 669 for more information on the PAT MSR number and reset value.

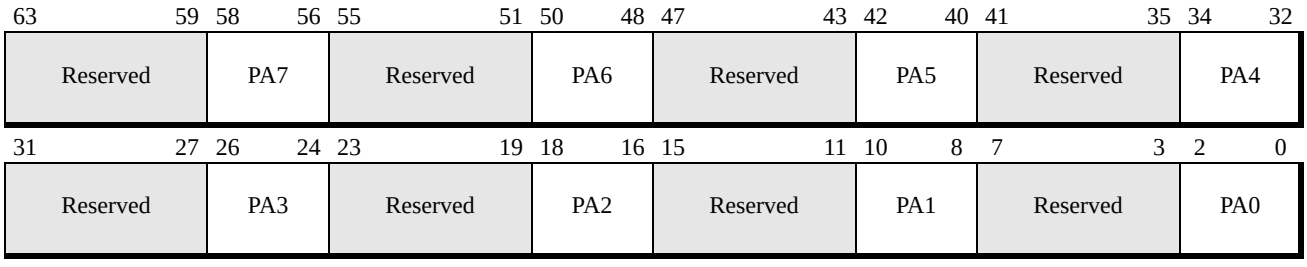


Figure 7-10. PAT Register

The PAT register contains eight page-attribute (PA) fields, numbered from PA0 to PA7. The PA fields hold the encoding of a memory type, as found in Table 7-8 on page 219. The PAT type-encodings

match the MTRR type-encodings, with the exception that PAT adds the 07h encoding. The 07h encoding corresponds to a *UC−* type. The UC− type (07h) is identical to the UC type (00h) except it can be overridden by an MTRR type of WC.

Software can write any supported memory-type encoding into any of the eight PA fields. An attempt to write anything but zeros into the reserved fields causes a general-protection exception (#GP). An attempt to write an unsupported type encoding into a PA field also causes a #GP exception.

The PAT register fields are initiated at processor reset to the default values shown in Table 7-9 on page 220.

Table 7-8. PAT Type Encodings

Type Value	Type Name	Type Description
00h	UC—Uncacheable	All accesses are uncacheable. Write combining is not allowed. Speculative accesses are not allowed.
01h	WC—Write-Combining	All accesses are uncacheable. Write combining is allowed. Speculative reads are allowed.
04h	WT—Writethrough	Reads allocate cache lines on a cache miss, but only to the shared state. Cache lines are not allocated on a write miss. Write hits update the cache and main memory.
05h	WP—Write-Protect	Reads allocate cache lines on a cache miss, but only to the shared state. All writes update main memory. Cache lines are not allocated on a write miss. Write hits invalidate the cache line and update main memory.
06h	WB—Writeback	Reads allocate cache lines on a cache miss, and can allocate to either the shared or exclusive state. Writes allocate to the modified state on a cache miss.
07h	UC– (UC minus)	All accesses are uncacheable. Write combining is not allowed. Speculative accesses are not allowed. Can be overridden by an MTRR with the WC type.

7.8.2 PAT Indexing

PA fields in the PAT register are selected using three bits from the page-table entries. These bits are:

- *PAT (page attribute table)*—The PAT bit is bit 7 in 4-Kbyte PTEs; it is bit 12 in 2-Mbyte and 4-Mbyte PDEs. Page-table entries that don't have a PAT bit (PML4 entries, for example) assume PAT = 0.
- *PCD (page cache disable)*—The PCD bit is bit 4 in all page-table entries. The PCD from the PTE or PDE is selected depending on the paging mode.
- *PWT (page writethrough)*—The PWT bit is bit 3 in all page-table entries. The PWT from the PTE or PDE is selected depending on the paging mode.

Table 7-9 on page 220 shows the various combinations of the PAT, PCD, and PWT bits used to select a PA field within the PAT register. Table 7-9 also shows the default memory-type values established in the PAT register by the processor after a reset. The default values correspond to the memory types

established by the PCD and PWT bits alone in processor implementations that do not support the PAT mechanism. In such implementations, the PAT field in page-table entries is reserved and cleared to 0. See “Page-Translation-Table Entry Fields” on page 150 for more information on the page-table entries.

Table 7-9. PAT-Register PA-Field Indexing

Page Table Entry Bits			PAT Register Field	Default Memory Type
PAT	PCD	PWT		
0	0	0	PA0	WB
0	0	1	PA1	WT
0	1	0	PA2	UC–[1]
0	1	1	PA3	UC
1	0	0	PA4	WB
1	0	1	PA5	WT
1	1	0	PA6	UC–[1]
1	1	1	PA7	UC

Note:
1. Can be overridden by WC memory type set by an MTRR.

7.8.3 Identifying PAT Support

Software determines whether a processor supports the PAT mechanism by executing the CPUID instruction with either function 0000_0001h or function 8000_0001h. If PAT is supported, bit 16 in the EDX register is set to 1 by CPUID. See Section 3.3, “Processor Feature Identification,” on page 71 for more information on the CPUID instruction.

If PAT is supported by a processor implementation, it is *always enabled*. The PAT mechanism cannot be disabled by software. Software can effectively avoid using PAT by:

- Not setting PAT bits in page-table entries to 1.
- Not modifying the reset values of the PA fields in the PAT register.

In this case, memory is characterized using the same types that are used by implementations that do not support PAT.

7.8.4 PAT Accesses

In implementations that support the PAT mechanism, all memory accesses that are translated through the paging mechanism use the PAT index bits to specify a PA field in the PAT register. The memory type stored in the specified PA field is applied to the memory access. The process is summarized as:

1. A virtual address is calculated as a result of a memory access.
2. The virtual address is translated to a physical address using the page-translation mechanism.
3. The PAT, PCD and PWT bits are read from the corresponding page-table entry during the virtual-address to physical-address translation.

4. The PAT, PCD and PWT bits are used to select a PA field from the PAT register.
5. The memory type is read from the appropriate PA field.
6. The memory type is applied to the physical-memory access using the translated physical address.

Page-Translation Table Access. The PAT bit exists only in the PTE (4K paging) or PDEs (2/4 Mbyte paging). In the remaining upper levels (PML4, PDP, and 4KB PDEs), only the PWT and PCD bits are used to index into the first 4 entries in the PAT register. The resulting memory type is used for the next lower paging level.

7.8.5 Combined Effect of MTRRs and PAT

The memory types established by the PAT mechanism can be combined with MTRR-established memory types to form an effective memory-type. The combined effect of MTRR and PAT memory types are shown in Figure 7-10. In the AMD64 architecture, reserved and undefined combinations of MTRR and PAT memory types result in undefined behavior. If the MTRRs are disabled in implementations that support the MTRR mechanism, the default memory type is set to uncacheable (UC).

Table 7-10. Combined Effect of MTRR and PAT Memory Types

PAT Memory Type	MTRR Memory Type	Effective Memory Type
UC	UC	UC
UC	WC, WP, WT, WB	CD
UC−	UC	UC
	WC	WC
	WP, WT, WB	CD
WC	—	WC
WP	UC	UC
	WC	CD
	WP	WP
	WT	CD
	WB	WP
WT	UC	UC
	WC, WP	CD
	WT, WB	WT
WB	UC	UC
	WC	WC
	WP	WP
	WT	WT
	WB	WB

7.8.6 PATs in Multi-Processing Environments

In multi-processing environments, values of CR0.CD and the PAT must be consistent across all processors and the MTRRs in all processors must characterize memory in the same way. In other words, matching address ranges and cachability types are written to the MTRRs for each processor.

Failure to do so may result in coherency violations or loss of atomicity. Processor implementations *do not* check the MTRR, CR0.CD and PAT values in other processors to ensure consistency. It is the responsibility of system software to initialize and maintain consistency across all processors.

7.8.7 Changing Memory Type

A physical page should not have differing cacheability types assigned to it through different virtual mappings; they should be either all of a cacheable type (WB, WT, WP) or all of a non-cacheable type (UC, WC). Otherwise, this may result in a loss of cache coherency, leading to stale data and unpredictable behavior. For this reason, certain precautions must be taken when changing the memory type of a page. In particular, when changing from a cachable memory type to an uncachable type the caches must be flushed, because speculative execution by the processor may have resulted in memory being cached even though it was not programatically referenced. The following table summarizes the serialization requirements for safely changing memory types.

Table 7-11. Serialization Requirements for Changing Memory Types

		New Type				
		WB	**WT**	**WP**	**UC**	**WC**
Old Type	WB	–	a	a	b	b
	WT	a	–	a	b	b
	WP	a	a	–	b	b
	UC	a	a	a	–	a
	WC	a	a	a	a	–

Note:

a. Remove the previous mapping (make it not present in the page tables); Flush the TLBs including the TLBs of other processors that may have used the mapping, even speculatively; Create a new mapping in the page tables using the new type.

b. In addition to the steps described in note a, software should flush the page from the caches of any processor that may have used the previous mapping. This must be done after the TLB flushing in note a has been completed.

7.9 Memory-Mapped I/O

Processor implementations can independently direct reads and writes to either system memory or memory-mapped I/O. The method used for directing those memory accesses is implementation dependent. In some implementations, separate system-memory and memory-mapped I/O buses can be provided at the processor interface. In other implementations, system memory and memory-mapped I/O share common data and address buses, and system logic uses sideband signals from the processor to route accesses appropriately. Refer to AMD data sheets and application notes for more information about particular hardware implementations of the AMD64 architecture.

The I/O range registers (IORRs), and the top-of-memory registers allow system software to specify where memory accesses are directed for a given address range. The MTRR extensions are described in the following section. “IORRs” on page 224 describes the IORRs and “Top of Memory” on page 226 describes the top-of-memory registers. *In implementations that support these features, the default action taken when the features are disabled is to direct memory accesses to memory-mapped I/O.*

7.9.1 Extended Fixed-Range MTRR Type-Field Encodings

The fixed-range MTRRs support extensions to the type-field encodings that allow system software to direct memory accesses to system memory or memory-mapped I/O. The extended MTRR type-field encodings use previously reserved bits 4:3 to specify whether reads and writes to a physical-address range are to system memory or to memory-mapped I/O. The format for this encoding is shown in Figure 7-11 on page 223. The new bits are:

- *WrMem*—Bit 3. When set to 1, the processor directs write requests for this physical address range to system memory. When cleared to 0, writes are directed to memory-mapped I/O.
- *RdMem*—Bit 4. When set to 1, the processor directs read requests for this physical address range to system memory. When cleared to 0, reads are directed to memory-mapped I/O.

The type subfield (bits 2:0) allows the encodings specified in Table 7-5 on page 209 to be used for memory characterization.

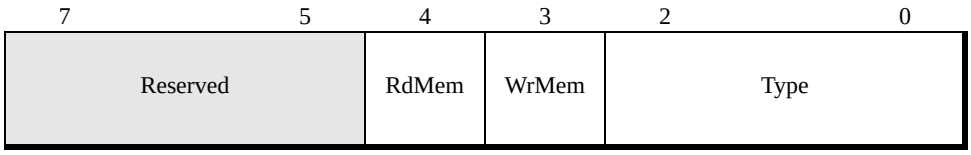


Figure 7-11. Extended MTRR Type-Field Format (Fixed-Range MTRRs)

These extensions are enabled using the following bits in the SYSCFG MSR:

- *MtrrFixDramEn*—Bit 18. When set to 1, RdMem and WrMem attributes are enabled. When cleared to 0, these attributes are disabled. *When disabled, accesses are directed to memory-mapped I/O space.*
- *MtrrFixDramModEn*—Bit 19. When set to 1, software can read and write the RdMem and WrMem bits. When cleared to 0, writes do not modify the RdMem and WrMem bits, and reads return 0.

To use the MTRR extensions, system software must first set MtrrFixDramModEn=1 to allow modification to the RdMem and WrMem bits. After the attribute bits are properly initialized in the fixed-range registers, the extensions can be enabled by setting MtrrFixDramEn=1.

RdMem and WrMem allow the processor to independently direct reads and writes to either system memory or memory-mapped I/O. The RdMem and WrMem controls are particularly useful when shadowing ROM devices located in memory-mapped I/O space. It is often useful to shadow such devices in RAM system memory to improve access performance, but writes into the RAM location can

corrupt the shadowed ROM information. The MTRR extensions solve this problem. System software can create the shadow location by setting WrMem = 1 and RdMem = 0 for the specified memory range and then copy the ROM location into itself. Reads are directed to the memory-mapped ROM, but writes go to the same physical addresses in system memory. After the copy is complete, system software can change the bit values to WrMem = 0 and RdMem = 1. Now reads are directed to the faster copy located in system memory, and writes are directed to memory-mapped ROM. The ROM responds as it would normally to a write, which is to ignore it.

Not all combinations of RdMem and WrMem are supported for each memory type encoded by bits 2:0. Table 7-12 on page 224 shows the allowable combinations. The behavior of reserved encoding combinations (shown as gray-shaded cells) is undefined and results in unpredictable behavior.

Table 7-12. Extended Fixed-Range MTRR Type Encodings

RdMem	WrMem	Type	Implication or Potential Use
0	0	0 (UC)	UC I/O
		1 (WC)	WC I/O
		4 (WT)	WT I/O
		5 (WP)	WP I/O
		6 (WB)	Reserved
0	1	0 (UC)	Used while creating a shadowed ROM
		1 (WC)	
		4 (WT)	Reserved
		5 (WP)	
		6 (WB)	
1	0	0 (UC)	Used to access a shadowed ROM
		1 (WC)	Reserved
		4 (WT)	
		5 (WP)	WP Memory (Can be used to access shadowed ROM)
		6 (WB)	Reserved
1	1	0 (UC)	UC Memory
		1 (WC)	WC Memory
		4 (WT)	WT Memory
		5 (WP)	Reserved
		6 (WB)	WB Memory

7.9.2 IORRs

The IORRs operate similarly to the variable-range MTRRs. The IORRs specify whether reads and writes in any physical-address range map to system memory or memory-mapped I/O. Up to two

address ranges of varying sizes can be controlled using the IORRs. A pair of IORRs are used to control each address range: IORRBase*n* and IORRMask*n* (*n* is the address-range number from 0 to 1).

Figure 7-12 on page 225 shows the format of the IORRBase*n* registers and Figure 7-13 on page 226 shows the format of the IORRMask*n* registers. The fields within the register pair are read/write.

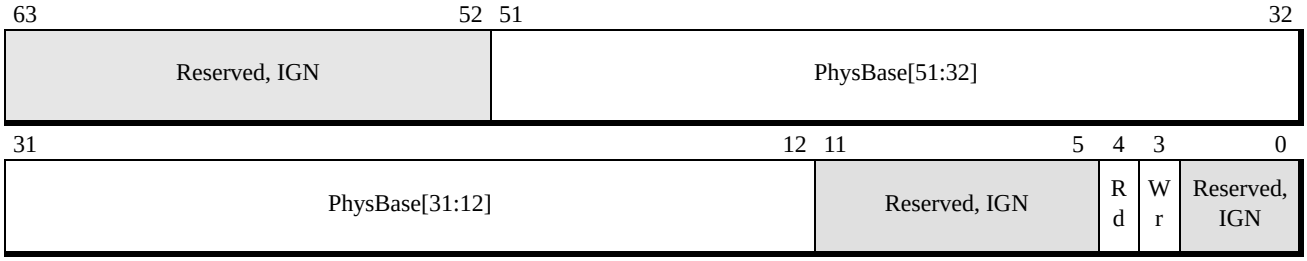
The intersection of the IORR range with the equivalent effective MTRR range follows the same type encoding table (Table 7-12) as the fixed-range MTRR, where the RdMem/WrMem and memory type are directly tied together.

IORRBase*n* Registers. The fields in these IORRs are:

- *WrMem*—Bit 3. When set to 1, the processor directs write requests for this physical address range to system memory. When cleared to 0, writes are directed to memory-mapped I/O.
- *RdMem*—Bit 4. When set to 1, the processor directs read requests for this physical address range to system memory. When cleared to 0, reads are directed to memory-mapped I/O.
- *Range Physical-Base-Address (PhysBase)*—Bits 51:12. The memory-range base-address in physical-address space. PhysBase is aligned on a 4-Kbyte (or greater) address in the 52-bit physical-address space supported by the AMD64 architecture. PhysBase represents the most-significant 40-address bits of the physical address. Physical-address bits 11:0 are assumed to be 0.

Note that a given processor may implement less than the architecturally-defined physical address size of 52 bits.

The format of these registers is shown in Figure 7-12.



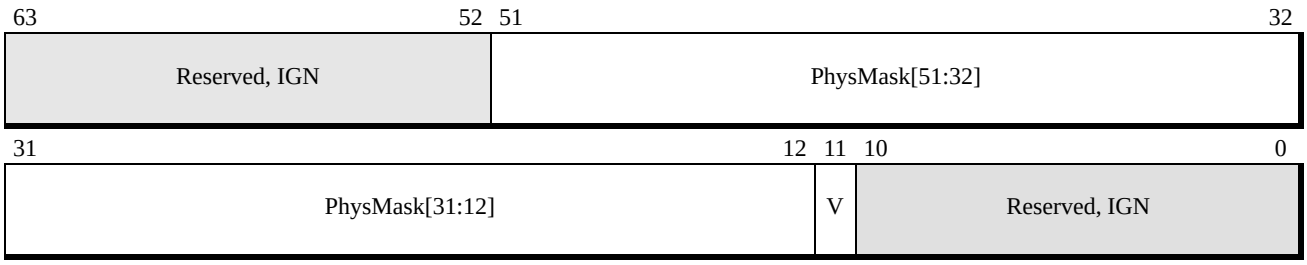
Bits	Mnemonic	Description	Access type
63:52	—	Reserved	Ignored
51:12	PhysBase	Range Physical Base Address	R/W
11:5	—	Reserved	Ignored
4	Rd	RdMem Enable	R/W
3	Wr	WrMem Enable	R/W
2:0	—	Reserved	Ignored

Figure 7-12. IORRBase*n* Register

IORRMask*n* Registers. The fields in these IORRs are:

- *Valid (V)*—Bit 11. Indicates that the IORR pair is valid (enabled) when set to 1. When the valid bit is cleared to 0 the register pair is not used for memory-mapped I/O control (disabled).
- *Range Physical-Mask (PhysMask)*—Bits 51:12. The mask value used to specify the memory range. Like PhysBase, PhysMask is aligned on a 4-Kbyte physical-address boundary. Bits 11:0 of PhysMask are assumed to be 0.

The format of these registers is shown in Figure 7-13 on page 226.



Bits	Mnemonic	Description	Access type
63:52	—	Reserved	Ignored
51:12	PhysMask	Range Physical Mask	R/W
11	V	I/O Register Pair Enable (Valid)	R/W
10:0	—	Reserved	Ignored

Figure 7-13. IORRMask*n* Register

The operation of the PhysMask and PhysBase fields is identical to that of the variable-range MTRRs. See page 213 for a description of this operation.

7.9.3 IORR Overlapping

The use of overlapping IORRs is not recommended. If overlapping IORRs are specified, the resulting behavior is implementation-dependent.

7.9.4 Top of Memory

The *top-of-memory* registers, TOP_MEM and TOP_MEM2, allow system software to specify physical addresses ranges as memory-mapped I/O locations. Processor implementations can direct accesses to memory-mapped I/O differently than system I/O, and the precise method depends on the implementation. System software specifies memory-mapped I/O regions by writing an address into each of the top-of-memory registers. The memory regions specified by the TOP_MEM registers are aligned on 8-Mbyte boundaries as follows:

- Memory accesses from physical address 0 to one less than the value in TOP_MEM are directed to system memory.
- Memory accesses from the physical address specified in TOP_MEM to FFFF_FFFFh are directed to memory-mapped I/O.

- Memory accesses from physical address 1_0000_0000h to one less than the value in TOP_MEM2 are directed to system memory.
- Memory accesses from the physical address specified in TOP_MEM2 to the maximum physical address supported by the system are directed to memory-mapped I/O.

Figure 7-14 on page 227 shows how the top-of-memory registers organize memory into separate system-memory and memory-mapped I/O regions.

The intersection of the top-of-memory range with the equivalent effective MTRR range follows the same type encoding table (Table 7-12 on page 224) as the fixed-range MTRR, where the RdMem/WrMem and memory type are directly tied together.

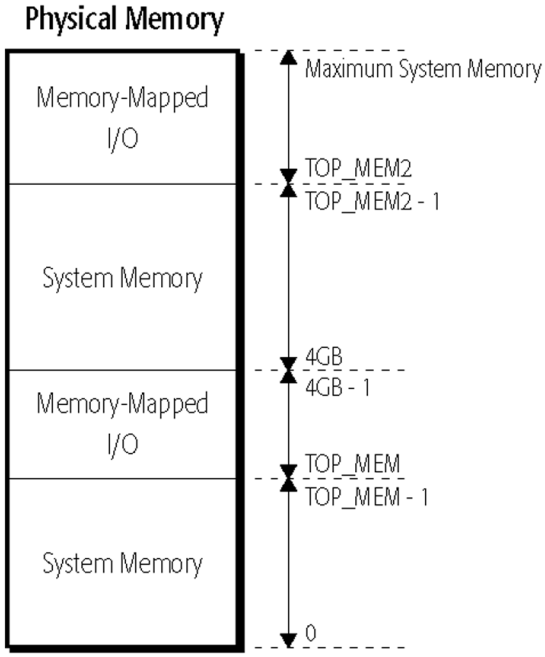


Figure 7-14. Memory Organization Using Top-of-Memory Registers

Figure 7-15 shows the format of the TOP_MEM and TOP_MEM2 registers. Bits 51:23 specify an 8-Mbyte aligned physical address. All remaining bits are reserved and ignored by the processor. System software should clear those bits to zero to maintain compatibility with possible future extensions to the registers. The TOP_MEM registers are model-specific registers. See “Memory-Typing MSRs” on page 669 for information on the MSR address and reset values for these registers.

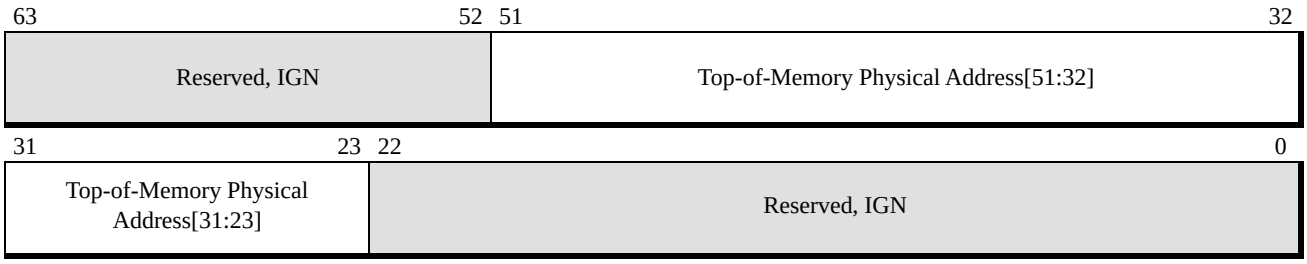


Figure 7-15. Top-of-Memory Registers (TOP_MEM, TOP_MEM2)

The TOP_MEM register is enabled by setting the MtrrVarDramEn bit in the SYSCFG MSR (bit 20) to 1 (one). The TOP_MEM2 register is enabled by setting the MtrrTom2En bit in the SYSCFG MSR (bit 21) to 1 (one). The registers are disabled when their respective enable bits are cleared to 0. When the top-of-memory registers are disabled, memory accesses default to memory-mapped I/O space.

Note that a given processor may implement fewer than the architecturally-defined number of physical address bits.

7.10 Secure Memory Encryption

Software running in non-virtualized (native) mode can utilize the Secure Memory Encryption (SME) feature to mark individual pages of memory as encrypted through the page tables. A page of memory marked encrypted will be automatically decrypted when read by software and automatically encrypted when written to DRAM. SME may therefore be used to protect the contents of DRAM from physical attacks on the system.

All memory encrypted using SME is encrypted with the same AES key which is created randomly each time a system is booted. The memory encryption key cannot be read or modified by software.

For details on using memory encryption in virtualized environments, please see Section 15.34, “Secure Encrypted Virtualization,” on page 571.

7.10.1 Determining Support for Secure Memory Encryption

Support for memory encryption features is reported in CPUID Fn8000_001F[EAX]. Bit 0 indicates support for Secure Memory Encryption. When this feature is present, CPUID Fn8000_001F[EBX] supplies additional information regarding the use of memory encryption such as which page table bit is used to mark pages as encrypted.

Additionally, in some implementations, the physical address size of the processor may be reduced when memory encryption features are enabled, for example from 48 to 43 bits. In this case the upper physical address bits are treated as reserved when the feature is enabled except where otherwise indicated. When memory encryption is supported in an implementation, CPUID Fn8000_001F[EBX] reports any physical address size reduction present. Bits reserved in this mode are treated the same as

other page table reserved bits, and will generate a page fault if found to be non-zero when used for address translation.

Complete CPUID details for encrypted memory features can be found in Volume 3, section E.4.17.

7.10.2 Enabling Memory Encryption Extensions

Prior to using SME, memory encryption features must be enabled by setting SYSCFG MSR bit 23 (MemEncryptionModEn) to 1. In implementations where the physical address size of the processor is reduced when memory encryption features are enabled, software must ensure it is executing from addresses where these upper physical address bits are 0 prior to setting SYSCFG[MemEncryptionModEn]. Memory encryption is then further controlled via the page tables.

Note that software should keep the value of SYSCFG[MemEncryptionModEn] consistent across all CPU cores in the system. Failure to do so may lead to unexpected results.

7.10.3 Supported Operating Modes

SME is supported in all CPU modes when CR4.PAE=1 and paging is enabled. This includes long mode as well as legacy PAE-enabled protected mode.

7.10.4 Page Table Support

Software utilizes the page tables to indicate if a memory page is encrypted or unencrypted. The location of the specific attribute bit (C-bit, or enCrypted bit) used is implementation-specific but may be determined by referencing CPUID Fn8000_001F[EBX] (see Volume 3, section E.4.17 for details) . In some implementations, the bit used may be a physical address bit (e.g., address bit 47), especially in cases where the physical address size is reduced by hardware when memory encryption features are enabled.

To mark a memory page for encryption when stored in DRAM, software sets the C-bit to 1 for the page. If the C-bit is 0, the page is not encrypted when stored in DRAM. The C bit can be applied to translation table entries for any size of page - 4KB, 2MB, or 1GB.

Note that it is possible for the page tables themselves to be located in encrypted memory. For instance, if the C-bit is set in a PML4 entry, the PDP table it points to (and thus all PDPEs in that table) will be loaded from encrypted memory.

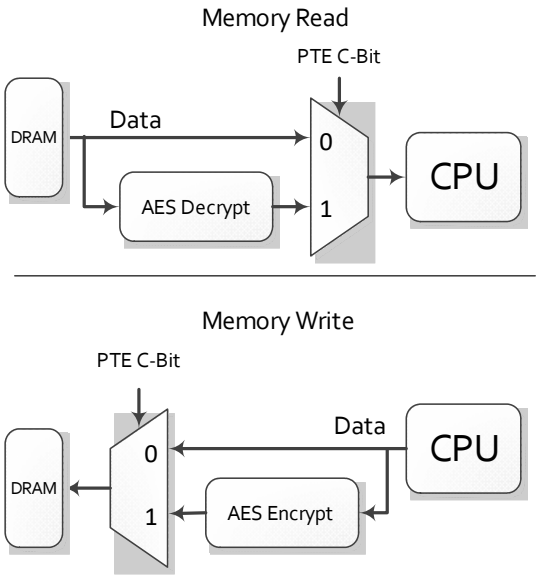


Figure 7-16. Encrypted Memory Accesses

7.10.5 I/O Accesses

In implementations where the physical address size is reduced when memory encryption features are enabled, memory range checks (e.g. MTRR/TOM/IORR/etc.) to determine memory types or DRAM/MMIO are performed using the reduced physical address size. In particular, the C-bit is not considered a physical address bit and is masked by hardware for purposes of these checks.

Additionally, any pages corresponding to MMIO addresses must be configured with the C-bit clear. Encrypted I/O pages are not allowed and accesses with the C-bit set will be ignored.

7.10.6 Restrictions

In some hardware implementations, coherency between the encrypted and unencrypted mappings of the same physical page are not enforced. In such a system, prior to changing the value of the C-bit for a page, software should flush the page from all CPU caches in the system. If a hardware implementation supports coherency across encryption domains as indicated by CPUID Fn8000_001F_EAX[10] then this flush is not required.

Simply changing the value of a C-bit on a page will not automatically encrypt the existing contents of a page, and any data in the page prior to the C-bit modification will become unintelligible. To set the C-bit on a page and cause its contents to become encrypted so the data remains accessible, see Section 7.10.8, “Encrypt-in-Place,” on page 231.

In legacy PAE mode, if the C-bit location is in the upper 32 bits of the page table entry, the first level page table (the PDP table) cannot be located in encrypted memory. This is because when the CPU is in 32-bit PAE mode, the CR3 value is only 32-bits in length.

7.10.7 SMM Interaction

SME is available when the processor is executing in SMM, once it has enabled paging. Any physical address bit restrictions that exist due to memory encryption features being enabled remain in place while in SMM.

7.10.8 Encrypt-in-Place

It is possible to perform an in-place encryption of data in physical memory. This technique is useful for setting the C-bit on a page while maintaining visibility to the page's contents such as during SME initialization. This is accomplished by creating two linear mappings of the same page where one mapping has the C-bit set to 0 and the other has the C-bit set to 1. To avoid possible data corruption, software should use the following algorithm for performing in-place encryption of memory:

1. Create two linear mappings X and Y that map to the same physical page. Mapping X has C-bit=0 and uses the WP (Write Protect) memory type. Mapping Y has C-bit=1 and uses the WB (Write-Back) memory type.
2. Perform a WBINVD on all cores in the system.
3. Copy N bytes from mapping X to a temporary buffer in conventionally-mapped memory (for which the C bit may or may not be set, as desired). N must be equal to the L1 cache line size as specified by CPUID Fn8000_0005[ECX].
4. Write N bytes from the temporary buffer to Y. Note that the initial cache refill of the line for this step will cause it to be decrypted, which corrupts the contents since it is not yet encrypted. This step restores the original contents. (If the line were evicted before this step was completed, the unwritten portion would get corrupted by the outgoing encryption, which is why the line can't be copied in-place, but rather must be copied from the temporary buffer.)
5. Repeat steps 3-4 until the entire page has been copied.

8 Exceptions and Interrupts

Exceptions and interrupts force control transfers from the currently-executing program to a system-software service routine that handles the interrupting event. These routines are referred to as *exception handler*s and *interrupt handlers*, or collectively as *event handlers*. Typically, interrupt events can be handled by the service routine transparently to the interrupted program. During the control transfer to the service routine, the processor stops executing the interrupted program and saves its return pointer. The system-software service routine that handles the exception or interrupt is responsible for saving the state of the interrupted program. This allows the processor to restart the interrupted program after system software has handled the event.

When an exception or interrupt occurs, the processor uses the interrupt vector number as an index into the interrupt-descriptor table (IDT). An IDT is used in all processor operating modes, including real mode (also called real-address mode), protected mode, and long mode.

Exceptions and interrupts come from three general sources:

- *Exceptions* occur as a result of software execution errors or other internal-processor errors. Exceptions also occur during non-error situations, such as program single stepping or address-breakpoint detection. Exceptions are considered *synchronous* events because they are a direct result of executing the interrupted instruction.
- *Software interrupts* occur as a result of executing interrupt instructions. Unlike exceptions and external interrupts, software interrupts allow intentional triggering of the interrupt-handling mechanism. Like exceptions, software interrupts are synchronous events.
- *External interrupts* are generated by system logic in response to an error or some other event outside the processor. They are reported over the processor bus using external signaling. External interrupts are *asynchronous* events that occur independently of the interrupted instruction.

Throughout this section, the term *masking* can refer to either disabling or delaying an interrupt. For example, masking external interrupts *delays* the interrupt, with the processor holding the interrupt as pending until it is unmasked. With floating-point exceptions (SSE and x87), masking *prevents* an interrupt from occurring and causes the processor to perform a default operation on the exception condition.

8.1 General Characteristics

Exceptions and interrupts have several different characteristics that depend on how events are reported and the implications for program restart.

8.1.1 Precision

Precision describes how the exception is related to the interrupted program:

- *Precise* exceptions are reported on a predictable instruction boundary. This boundary is generally the first instruction that has not completed when the event occurs. All previous instructions (in

program order) are allowed to complete before transferring control to the event handler. The pointer to the instruction boundary is saved automatically by the processor. When the event handler completes execution, it returns to the interrupted program and restarts execution at the interrupted-instruction boundary.

- *Imprecise* exceptions are not guaranteed to be reported on a predictable instruction boundary. The boundary can be any instruction that has not completed when the interrupt event occurs. Imprecise events can be considered asynchronous, because the source of the interrupt is not necessarily related to the interrupted instruction. Imprecise exception and interrupt handlers typically collect machine-state information related to the interrupting event for reporting through system-diagnostic software. The interrupted program is not restartable.

8.1.2 Instruction Restart

As mentioned above, precise exceptions are reported on an instruction boundary. The instruction boundary can be reported in one of two locations:

- Most exceptions report the boundary *before* the instruction causing the exception. In this case, all previous instructions (in program order) are allowed to complete, but the interrupted instruction is not. *No program state is updated as a result of partially executing an interrupted instruction.*
- Some exceptions report the boundary *after* the instruction causing the exception. In this case, all previous instructions—including the one executing when the exception occurred—are allowed to complete.

  Program state can be updated when the reported boundary is after the instruction causing the exception. This is particularly true when the event occurs as a result of a task switch. In this case, the general registers, segment-selector registers, page-base address register, and LDTR are all updated by the hardware task-switch mechanism. The event handler cannot rely on the state of those registers when it begins execution and must be careful in validating the state of the segment-selector registers before restarting the interrupted task. This is not an issue in long mode, however, because the hardware task-switch mechanism is disabled in long mode.

8.1.3 Types of Exceptions

There are three types of exceptions, depending on whether they are precise and how they affect program restart:

- *Faults* are precise exceptions reported on the boundary before the instruction causing the exception. Generally, faults are caused by an error condition involving the faulted instruction. Any machine-state changes caused by the faulting instruction are discarded so that the instruction can be restarted. The saved rIP points to the faulting instruction.
- *Traps* are precise exceptions reported on the boundary following the instruction causing the exception. The trapped instruction is completed by the processor and all state changes are saved. The saved rIP points to the instruction following the faulting instruction.
- *Aborts* are imprecise exceptions. Because they are imprecise, aborts typically do not allow reliable program restart.

8.1.4 Masking External Interrupts

General Masking Capabilities. Software can *mask* the occurrence of certain exceptions and interrupts. Masking can delay or even prevent triggering of the exception-handling or interrupt-handling mechanism when an interrupt-event occurs. External interrupts are classified as maskable or nonmaskable:

- *Maskable interrupts* trigger the interrupt-handling mechanism only when RFLAGS.IF=1. Otherwise they are held pending for as long as the RFLAGS.IF bit is cleared to 0.
- *Nonmaskable interrupts* (NMI) are unaffected by the value of the RFLAGS.IF bit. However, the occurrence of an NMI masks further NMIs until an IRET instruction is executed to completion or, in the event of a task switch, to the completion of the outgoing TSS update. An exception raised during execution of the IRET prior to these points will result in NMI continuing to be masked for the duration of the exception handler, until the exception handler completes an IRET.

Masking During Stack Switches. The processor delays recognition of maskable external interrupts and debug exceptions during certain instruction sequences that are often used by software to switch stacks. The typical programming sequence used to switch stacks is:

1. Load a stack selector into the SS register.
2. Load a stack offset into the ESP register.

If an interrupting event occurs after the selector is loaded but before the stack offset is loaded, the interrupted-program stack pointer is invalid during execution of the interrupt handler.

To prevent interrupts from causing stack-pointer problems, the processor does not allow external interrupts or debug exceptions to occur until the instruction immediately following the MOV SS or POP SS instruction completes execution.

The recommended method of performing this sequence is to use the LSS instruction. LSS loads both SS and ESP, and the instruction inhibits interrupts until both registers are updated successfully.

8.1.5 Masking Floating-Point and Media Instructions

Any x87 floating-point exceptions can be masked and reported later using bits in the x87 floating-point status register (FSW) and the x87 floating-point control register (FCW). The floating-point exception-pending exception is used for unmasked x87 floating-point exceptions (see Section “#MF—x87 Floating-Point Exception-Pending (Vector 16)” on page 248).

The SIMD floating-point exception is used for unmasked SSE floating-point exceptions (see Section “#XF—SIMD Floating-Point Exception (Vector 19)” on page 250). SSE floating-point exceptions are masked using the MXCSR register. The exception mechanism is not triggered when these exceptions are masked. Instead, the processor handles the exceptions in a default manner.

8.1.6 Disabling Exceptions

Disabling an exception prevents the exception condition from being recognized, unlike masking an exception which prevents triggering the exception mechanism after the exception is recognized. Some exceptions can be disabled by system software running at CPL=0, using bits in the CR0 register or CR4 register:

- Alignment-check exception (see Section “#AC—Alignment-Check Exception (Vector 17)” on page 249).
- Device-not-available exception (see Section “#NM—Device-Not-Available Exception (Vector 7)” on page 242).
- Machine-check exception (see Section “#MC—Machine-Check Exception (Vector 18)” on page 250).

The debug-exception mechanism provides control over when specific breakpoints are enabled and disabled. See Section “Setting Breakpoints” on page 393 for more information on how breakpoint controls are used for triggering the debug-exception mechanism.

8.2 Vectors

Specific exception and interrupt sources are assigned a fixed vector-identification number (also called an “interrupt vector” or simply “vector”). The interrupt vector is used by the interrupt-handling mechanism to locate the system-software service routine assigned to the exception or interrupt. Up to 256 unique interrupt vectors are available. The first 32 vectors are reserved for predefined exception and interrupt conditions. Software-interrupt sources can trigger an interrupt using any available interrupt vector.

Table 8-1 on page 237 lists the supported interrupt vector numbers, the corresponding exception or interrupt name, the mnemonic, the source of the interrupt event, and a summary of the possible causes.

Table 8-1. Interrupt Vector Source and Cause

Vector	Exception/Interrupt	Mnemonic	Cause
0	Divide-by-Zero-Error	#DE	DIV, IDIV, AAM instructions
1	Debug	#DB	Instruction accesses and data accesses
2	Non-Maskable-Interrupt	#NMI	External NMI signal
3	Breakpoint	#BP	INT3 instruction
4	Overflow	#OF	INTO instruction
5	Bound-Range	#BR	BOUND instruction
6	Invalid-Opcode	#UD	Invalid instructions
7	Device-Not-Available	#NM	x87 instructions
8	Double-Fault	#DF	Exception during the handling of another exception or interrupt
9	Coprocessor-Segment-Overrun	—	Unsupported (Reserved)
10	Invalid-TSS	#TS	Task-state segment access and task switch
11	Segment-Not-Present	#NP	Segment register loads
12	Stack	#SS	SS register loads and stack references
13	General-Protection	#GP	Memory accesses and protection checks
14	Page-Fault	#PF	Memory accesses when paging enabled
15	Reserved	—	
16	x87 Floating-Point Exception-Pending	#MF	x87 floating-point instructions
17	Alignment-Check	#AC	Misaligned memory accesses
18	Machine-Check	#MC	Model specific
19	SIMD Floating-Point	#XF	SSE floating-point instructions
20	Reserved	—	
21	Control-Protection Exception	#CP	RET/IRET or other control transfer
22–27	Reserved	—	
28	Hypervisor Injection Exception	#HV	Event injection
29	VMM Communication Exception	#VC	Virtualization event
30	Security Exception	#SX	Security-sensitive event in host
31	Reserved	—	
0–255	External Interrupts (Maskable)	#INTR	External interrupts
0–255	Software Interrupts	—	INT*n* instruction

Table 8-2 on page 238 shows how each interrupt vector is classified. Reserved interrupt vectors are indicated by the gray-shaded rows.

Table 8-2. Interrupt Vector Classification

Vector	Interrupt (Exception)	Type	Precise	Class[2]
0	Divide-by-Zero-Error	Fault	yes	Contributory
1	Debug	Fault or Trap		Benign
2	Non-Maskable-Interrupt	—	—	
3	Breakpoint	Trap	yes	
4	Overflow			
5	Bound-Range	Fault		
6	Invalid-Opcode			
7	Device-Not-Available			
8	Double-Fault	Abort	no	
9	Coprocessor-Segment-Overrun			
10	Invalid-TSS	Fault	yes	Contributory
11	Segment-Not-Present			
12	Stack			
13	General-Protection			
14	Page-Fault			Benign or Contributory
15	Reserved			
16	x87 Floating-Point Exception-Pending	Fault	no	Benign
17	Alignment-Check		yes	
18	Machine-Check	Abort	no	
19	SIMD Floating-Point	Fault	yes	
20	Reserved			
21	Control Protection	Fault	yes	Contributory
22–27	Reserved			
28	Hypervisor Injection Exception	–	–	Benign
29	VMM Communication Exception	Fault	yes	Contributory
30	Security Exception	–	yes	Contributory
31	Reserved			
0–255	External Interrupts (Maskable)	—[1]	—[1]	Benign
0–255	Software Interrupts			

Note:

1. *External interrupts are not classified by type or whether or not they are precise.*
2. *See Section "#DF—Double-Fault Exception (Vector 8)" on page 242 for a definition of benign and contributory classes.*

The following sections describe each interrupt in detail. The format of the error code reported by each interrupt is described in Section "Error Codes" on page 253.

8.2.1 #DE—Divide-by-Zero-Error Exception (Vector 0)

A #DE exception occurs when the denominator of a DIV instruction or an IDIV instruction is 0. A #DE also occurs if the result is too large to be represented in the destination.

#DE cannot be disabled.

Error Code Returned. None.

Program Restart. #DE is a fault-type exception. The saved instruction pointer points to the instruction that caused the #DE.

8.2.2 #DB—Debug Exception (Vector 1)

When the debug-exception mechanism is enabled, a #DB exception can occur under any of the following circumstances:

- Instruction execution.
- Instruction single stepping.
- Data read.
- Data write.
- I/O read.
- I/O write.
- Task switch.
- Debug-register access, or *general detect fault (debug register access when DR7.GD=1).*
- Executing the INT1 instruction (opcode 0F1h).

#DB conditions are enabled and disabled using the debug-control register, DR7 and RFLAGS.TF. Each #DB condition is described in more detail in Section “Setting Breakpoints” on page 393.

Error Code Returned. None. #DB information is returned in the debug-status register, DR6.

Program Restart. #DB can be either a fault-type or trap-type exception. In the following cases, the saved instruction pointer points to the instruction that caused the #DB:

- Instruction execution.
- Invalid debug-register access, or *general detect.*

In all other cases, the instruction that caused the #DB is completed, and the saved instruction pointer points to the instruction after the one that caused the #DB.

The RFLAGS.RF bit can be used to restart an instruction following an instruction breakpoint resulting in a #DB. In most cases, the processor clears RFLAGS.RF to 0 after every instruction is successfully executed. However, in the case of the IRET, JMP, CALL, and INT*n (*through a task gate) instructions, RFLAGS.RF is not cleared to 0 until the *next* instruction successfully executes.

When a non-debug exception occurs (or when a string instruction is interrupted), the processor normally sets RFLAGS.RF to 1 in the rFLAGS *image* that is pushed on the interrupt stack. A subsequent IRET back to the interrupted program pops the rFLAGS image off the stack and into the RFLAGS register, with RFLAGS.RF=1. The interrupted instruction executes without causing an instruction breakpoint, after which the processor clears RFLAGS.RF to 0.

However, when a #DB exception occurs, the processor clears RFLAGS.RF to 0 in the rFLAGS image that is pushed on the interrupt stack. The #DB handler has two options:

- Disable the instruction breakpoint completely.
- Set RFLAGS.RF to 1 in the interrupt-stack rFLAGS image. The instruction breakpoint condition is ignored immediately after the IRET, but reoccurs if the instruction address is accessed later, as can occur in a program loop.

8.2.3 NMI—Non-Maskable-Interrupt Exception (Vector 2)

An NMI exception occurs as a result of system logic signaling a non-maskable interrupt to the processor.

Error Code Returned. None.

Program Restart. NMI is an interrupt. The processor recognizes an NMI at an instruction boundary. The saved instruction pointer points to the instruction immediately following the boundary where the NMI was recognized.

Masking. NMI cannot be masked. However, when an NMI is recognized by the processor, recognition of subsequent NMIs are disabled until an IRET instruction is executed.

8.2.4 #BP—Breakpoint Exception (Vector 3)

A #BP exception occurs when an INT3 instruction is executed. The INT3 is normally used by debug software to set instruction breakpoints by replacing instruction-opcode bytes with the INT3 opcode.

#BP cannot be disabled.

Error Code Returned. None.

Program Restart. #BP is a trap-type exception. The saved instruction pointer points to the byte after the INT3 instruction. This location can be the start of the next instruction. However, if the INT3 is used to replace the first opcode bytes of an instruction, the restart location is likely to be in the middle of an instruction. In the latter case, the debug software must replace the INT3 byte with the correct instruction byte. The saved RIP instruction pointer must then be decremented by one before returning to the interrupted program. This allows the program to be restarted correctly on the interrupted-instruction boundary.

8.2.5 #OF—Overflow Exception (Vector 4)

An #OF exception occurs as a result of executing an INTO instruction while the overflow bit in RFLAGS is set to 1 (RFLAGS.OF=1).

#OF cannot be disabled.

Error Code Returned. None.

Program Restart. #OF is a trap-type exception. The saved instruction pointer points to the instruction following the INTO instruction that caused the #OF.

8.2.6 #BR—Bound-Range Exception (Vector 5)

A #BR exception can occur as a result of executing the BOUND instruction. The BOUND instruction compares an array index (first operand) with the lower bounds and upper bounds of an array (second operand). If the array index is not within the array boundary, the #BR occurs.

#BR cannot be disabled.

Error Code Returned. None.

Program Restart. #BR is a fault-type exception. The saved instruction pointer points to the BOUND instruction that caused the #BR.

8.2.7 #UD—Invalid-Opcode Exception (Vector 6)

A #UD exception occurs when an attempt is made to execute an invalid or undefined opcode. The validity of an opcode often depends on the processor operating mode. A #UD occurs under the following conditions:

- Execution of any reserved or undefined opcode in any mode.
- Execution of the UD0, UD1 or UD2 instructions.
- Use of the LOCK prefix on an instruction that cannot be locked.
- Use of the LOCK prefix on a lockable instruction with a non-memory target location.
- Execution of an instruction with an invalid-operand type.
- Execution of the SYSENTER or SYSEXIT instructions in long mode.
- Execution of any of the following instructions in 64-bit mode: AAA, AAD, AAM, AAS, BOUND, CALL (opcode 9A), DAA, DAS, DEC, INC, INTO, JMP (opcode EA), LDS, LES, POP (DS, ES, SS), POPA, PUSH (CS, DS, ES, SS), PUSHA, SALC.
- Execution of the ARPL, LAR, LLDT, LSL, LTR, SLDT, STR, VERR, or VERW instructions when protected mode is not enabled, or when virtual-8086 mode is enabled.
- Execution of any legacy SSE instruction when CR4.OSFXSR is cleared to 0. (For further information, see Section “FXSAVE/FXRSTOR Support (OSFXSR)” on page 50.

- Execution of any SSE instruction (uses YMM/XMM registers), or 64-bit media instruction (uses MMX™ registers) when CR0.EM = 1.
- Execution of any SSE floating-point instruction (uses YMM/XMM registers) that causes a numeric exception when CR4.OSXMMEXCPT = 0.
- Use of the DR4 or DR5 debug registers when CR4.DE = 1.
- Execution of RSM when not in SMM mode.

See the specific instruction description (in the other volumes) for additional information on invalid conditions.

#UD cannot be disabled.

Error Code Returned. None.

Program Restart. #UD is a fault-type exception. The saved instruction pointer points to the instruction that caused the #UD.

8.2.8 #NM—Device-Not-Available Exception (Vector 7)

A #NM exception occurs under any of the following conditions:

- An FWAIT/WAIT instruction is executed when CR0.MP=1 and CR0.TS=1.
- Any x87 instruction other than FWAIT is executed when CR0.EM=1.
- Any x87 instruction is executed when CR0.TS=1. The CR0.MP bit controls whether the FWAIT/WAIT instruction causes an #NM exception when TS=1.
- Any 128-bit or 64-bit media instruction when CR0.TS=1.

#NM can be enabled or disabled under the control of the CR0.MP, CR0.EM, and CR0.TS bits as described above. See Section 3.1.1 for more information on the CR0 bits used to control the #NM exception.

Error Code Returned. None.

Program Restart. #NM is a fault-type exception. The saved instruction pointer points to the instruction that caused the #NM.

8.2.9 #DF—Double-Fault Exception (Vector 8)

A #DF exception can occur when a second exception occurs during the handling of a prior (first) exception or interrupt handler.

Usually, the first and second exceptions can be handled sequentially without resulting in a #DF. In this case, the first exception is considered *benign*, as it does not harm the ability of the processor to handle the second exception.

In some cases, however, the first exception adversely affects the ability of the processor to handle the second exception. These exceptions contribute to the occurrence of a #DF, and are called *contributory*

exceptions. If a contributory exception is followed by another contributory exception, a double-fault exception occurs. Likewise, if a page fault is followed by another page fault or a contributory exception, a double-fault exception occurs.

Table 8-3 shows the conditions under which a #DF occurs. Page faults are either benign or contributory, and are listed separately. See the “Class” column in Table 8-2 on page 238 for information on whether an exception is benign or contributory.

Table 8-3. Double-Fault Exception Conditions

First Interrupting Event	Second Interrupting Event
Contributory Exceptions • Divide-by-Zero-Error Exception • Invalid-TSS Exception • Segment-Not-Present Exception • Stack Exception • General-Protection Exception	 Invalid-TSS Exception Segment-Not-Present Exception Stack Exception General-Protection Exception
Page Fault Exception	Page Fault Exception Invalid-TSS Exception Segment-Not-Present Exception Stack Exception General-Protection Exception

If a third interrupting event occurs while transferring control to the #DF handler, the processor shuts down. Only an NMI, RESET, or INIT can restart the processor in this case. However, if the processor shuts down as it is executing an NMI handler, the processor can only be restarted with RESET or INIT.

#DF cannot be disabled.

Error Code Returned. Zero.

Program Restart. #DF is an abort-type exception. The saved instruction pointer is undefined, and the program cannot be restarted.

8.2.10 Coprocessor-Segment-Overrun Exception (Vector 9)

This interrupt vector is reserved. It is for a discontinued exception originally used by processors that supported external x87-instruction coprocessors. On those processors, the exception condition is caused by an invalid-segment or invalid-page access on an x87-instruction coprocessor-instruction operand. On current processors, this condition causes a general-protection exception to occur.

Error Code Returned. Not applicable.

Program Restart. Not applicable.

8.2.11 #TS—Invalid-TSS Exception (Vector 10)

A #TS exception occurs when an invalid reference is made to a segment selector as part of a task switch. A #TS also occurs during a privilege-changing control transfer (through a call gate or an interrupt gate), if a reference is made to an invalid stack-segment selector located in the TSS. Table 8-4 lists the conditions under which a #TS occurs and the error code returned by the exception mechanism.

#TS cannot be disabled.

Error Code Returned. See Table 8-4 for a list of error codes returned by the #TS exception.

Program Restart. #TS is a fault-type exception. If the exception occurs before loading the segment selectors from the TSS, the saved instruction pointer points to the instruction that caused the #TS. However, most #TS conditions occur due to errors with the loaded segment selectors. When an error is found with a segment selector, the hardware task-switch mechanism completes loading the new task state from the TSS, and then triggers the #TS exception mechanism. In this case, the saved instruction pointer points to the first instruction in the new task.

In long mode, a #TS cannot be caused by a task switch, because the hardware task-switch mechanism is disabled. A #TS occurs only as a result of a control transfer through a gate descriptor that results in an invalid stack-segment reference using an SS selector in the TSS. In this case, the saved instruction pointer always points to the control-transfer instruction that caused the #TS.

Table 8-4. Invalid-TSS Exception Conditions

Selector Reference	Error Condition	Error Code
Task-State Segment	TSS limit check on a task switch	TSS Selector Index
	TSS limit check on an inner-level stack pointer	
LDT Segment	LDT does not point to GDT	LDT Selector Index
	LDT reference outside GDT	
	GDT entry is not an LDT descriptor	
	LDT descriptor is not present	
Code Segment	CS reference outside GDT or LDT	CS Selector Index
	Privilege check (conforming DPL > CPL)	
	Privilege check (non-conforming DPL ≠ CPL)	
	Type check (CS not executable)	
Data Segment	Data segment reference outside GDT or LDT	DS, ES, FS or GS Selector Index
	Type check (data segment not readable)	
Stack Segment	SS reference outside GDT or LDT	SS Selector Index
	Privilege check (stack segment descriptor DPL ≠ CPL)	
	Privilege check (stack segment selector RPL ≠ CPL)	
	Type check (stack segment not writable)	

8.2.12 #NP—Segment-Not-Present Exception (Vector 11)

An #NP occurs when an attempt is made to load a segment or gate with a clear present bit, as described in the following situations:

- Using the MOV, POP, LDS, LES, LFS, or LGS instructions to load a segment selector (DS, ES, FS, and GS) that references a segment descriptor containing a clear present bit (descriptor.P=0).
- Far transfer to a CS that is not present.
- Referencing a gate descriptor containing a clear present bit.
- Referencing a TSS descriptor containing a clear present bit. This includes attempts to load the TSS descriptor using the LTR instruction.
- Attempting to load a descriptor containing a clear present bit into the LDTR using the LLDT instruction.
- Loading a segment selector (CS, DS, ES, FS, or GS) as part of a task switch, with the segment descriptor referenced by the segment selector having a clear present bit. In long mode, an #NP cannot be caused by a task switch, because the hardware task-switch mechanism is disabled.

When loading a stack-segment selector (SS) that references a descriptor with a clear present bit, a stack exception (#SS) occurs. For information on the #SS exception, see the next section, "#SS—Stack Exception (Vector 12)."

#NP cannot be disabled.

Error Code Returned. The segment-selector index of the segment descriptor causing the #NP exception.

Program Restart. #NP is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that loaded the segment selector resulting in the #NP. See Section "Exceptions During a Task Switch" on page 253 for a description of the consequences when this exception occurs during a task switch.

8.2.13 #SS—Stack Exception (Vector 12)

An #SS exception can occur in the following situations:

- Implied stack references in which the stack address is not in canonical form. Implied stack references include all push and pop instructions, and any instruction using RSP or RBP as a base register.
- Attempting to load a stack-segment selector that references a segment descriptor containing a clear present bit (descriptor.P=0).
- Any stack access that fails the stack-limit check.

#SS cannot be disabled.

Error Code Returned. The error code depends on the cause of the #SS, as shown in Table 8-5 on page 246:

Table 8-5. Stack Exception Error Codes

Stack Exception Cause	Error Code
Stack-segment descriptor present bit is clear	SS Selector Index
Stack-limit violation	0
Stack reference using a non-canonical address	0

Program Restart. #SS is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #SS. See Section “Exceptions During a Task Switch” on page 253 for a description of the consequences when this exception occurs during a task switch.

8.2.14 #GP—General-Protection Exception (Vector 13)

Table 8-6 describes the general situations that can cause a #GP exception. The table is not an exhaustive, detailed list of #GP conditions, but rather a guide to the situations that can cause a #GP. If an invalid use of an AMD64 architectural feature results in a #GP, the specific cause of the exception is described in detail in the section describing the architectural feature.

#GP cannot be disabled.

Error Code Returned. As shown in Table 8-6, a selector index is reported as the error code if the #GP is due to a segment-descriptor access. In all other cases, an error code of 0 is returned.

Program Restart. #GP is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #GP. See Section “Exceptions During a Task Switch” on page 253 for a description of the consequences when this exception occurs during a task switch.

Table 8-6. General-Protection Exception Conditions

Error Condition	Error Code
Any segment privilege-check violation, while loading a segment register.	Selector Index
Any segment type-check violation, while loading a segment register.	
Loading a null selector into the CS, SS, or TR register.	
Accessing a gate-descriptor containing a null segment selector.	
Referencing an LDT descriptor or TSS descriptor located in the LDT.	
Attempting a control transfer to a busy TSS (except IRET).	
In 64-bit mode, loading a non-canonical base address into the GDTR or IDTR.	
In long mode, accessing a system or call-gate descriptor whose extended type field is not 0.	
In long mode, accessing a system descriptor containing a non-canonical base address.	
In long mode, accessing a gate descriptor containing a non-canonical offset.	
In long mode, accessing a gate descriptor that does not point to a 64-bit code segment.	
In long mode, accessing a 16-bit gate descriptor.	
In long mode, attempting a control transfer to a TSS or task gate.	

Table 8-6. General-Protection Exception Conditions (continued)

Error Condition	Error Code
Any segment limit-check or non-canonical address violation (except when using the SS register).	0
Accessing memory using a null segment register.	
Writing memory using a read-only segment register.	
Attempting to execute an SSE instruction specifying an unaligned memory operand.	
Attempting to execute code that is past the CS segment limit or at a non-canonical RIP.	
Executing a privileged instruction while CPL > 0.	
Executing an instruction that is more than 15 bytes long.	
Writing a 1 into any register field that is reserved, must be zero (MBZ).	
Using WRMSR to write a read-only MSR.	
Using WRMSR to write a non-canonical value into an MSR that must be canonical.	
Using WRMSR to set an invalid type encoding in an MTRR or the PAT MSR.	0
Enabling paging while protected mode is disabled.	
Setting CR0.NW=1 while CR0.CD=0.	
Any long-mode consistency-check violation.	

8.2.15 #PF—Page-Fault Exception (Vector 14)

A #PF exception can occur during a memory access in any of the following situations:

- A page-translation-table entry or physical page involved in translating the memory access is not present in physical memory. This is indicated by a cleared present bit (P=0) in the translation-table entry.
- An attempt is made by the processor to load the instruction TLB with a translation for a non-executable page.
- The memory access fails the paging-protection checks (user/supervisor, read/write, or both).
- A reserved bit in one of the page-translation-table entries is set to 1. A #PF occurs for this reason only when CR4.PSE=1 *or* CR4.PAE=1.
- A data access to a user-mode address caused a protection key violation.

#PF cannot be disabled.

CR2 Register. The virtual (linear) address that caused the #PF is stored in the CR2 register. The legacy CR2 register is 32 bits long. The CR2 register in the AMD64 architecture is 64 bits long, as shown in Figure 8-1 on page 248. In AMD64 implementations, when either software or a page fault causes a write to the CR2 register, only the low-order 32 bits of CR2 are used in legacy mode; the processor clears the high-order 32 bits.

Figure 8-1. Control Register 2 (CR2)

Error Code Returned. The page-fault error code is pushed onto the page-fault exception-handler stack. See Section “Page-Fault Error Code” on page 254 for a description of this error code.

Program Restart. #PF is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #PF. See Section “Exceptions During a Task Switch” on page 253 for a description of what can happen if this exception occurs during a task switch.

8.2.16 #MF—x87 Floating-Point Exception-Pending (Vector 16)

The #MF exception is used to handle unmasked x87 floating-point exceptions. An #MF occurs when *all* of the following conditions are true:

- CR0.NE=1.
- An unmasked x87 floating-point exception is pending. This is indicated by an exception bit in the x87 floating-point status-word register being set to 1
- The corresponding mask bit in the x87 floating-point control-word register is cleared to 0.
- The FWAIT/WAIT instruction or any waiting floating-point instruction is executed.

If there is an exception mask bit (in the FPU control word) set, the exception is not reported. Instead, the x87-instruction unit responds in a default manner and execution proceeds normally.

The x87 floating-point exceptions reported by the #MF exception are (including mnemonics):

- IE—Invalid-operation exception (also called #I), which is either:
  - IE alone—Invalid arithmetic-operand exception (also called #IA), or
  - SF and IE together—x87 Stack-fault exception (also called #IS).
- DE—Denormalized-operand exception (also called #D).
- ZE—Zero-divide exception (also called #Z).
- OE—Overflow exception (also called #O).
- UE—Underflow exception (also called #U).
- PE—Precision exception (also called #P or inexact-result exception).

Error Code Returned. None. Exception information is provided by the x87 status-word register. See “x87 Floating-Point Programming” in Volume 1 for more information on using this register.

Program Restart. #MF is a fault-type exception. The #MF exception is not precise, because multiple instructions and exceptions can occur before the #MF handler is invoked. Also, the saved instruction

pointer does not point to the instruction that caused the exception resulting in the #MF. Instead, the saved instruction pointer points to the x87 floating-point instruction or FWAIT/WAIT instruction that is about to be executed when the #MF occurs. The address of the *last instruction* that caused an x87 floating-point exception is in the x87 instruction-pointer register. See “x87 Floating-Point Programming” in Volume 1 for information on accessing this register.

Masking. Each type of x87 floating-point exception can be masked by setting the appropriate bits in the x87 control-word register. See “x87 Floating-Point Programming” in Volume 1 for more information on using this register.

#MF can also be disabled by clearing the CR0.NE bit to 0. See Section “Numeric Error (NE) Bit” on page 44 for more information on using this bit.

8.2.17 #AC—Alignment-Check Exception (Vector 17)

An #AC exception occurs when an unaligned-memory data reference is performed while alignment checking is enabled.

After a processor reset, #AC exceptions are disabled. Software enables the #AC exception by setting the following register bits:

- CR0.AM=1.
- RFLAGS.AC=1.

When the above register bits are set, an #AC can occur only when CPL=3. #AC never occurs when CPL < 3.

Table 8-7 lists the data types and the alignment boundary required to *avoid* an #AC exception when the mechanism is enabled.

Table 8-7. Data-Type Alignment

Supported Data Type	Required Alignment (Byte Boundary)
Word	2
Doubleword	4
Quadword	8
Bit string	2, 4 or 8 (depends on operand size)
256-bit media	16
128-bit media	16
64-bit media	8
Segment selector	2
32-bit near pointer	4
32-bit far pointer	2
48-bit far pointer	4

Table 8-7. Data-Type Alignment (continued)

Supported Data Type	Required Alignment (Byte Boundary)
x87 Floating-point single-precision	4
x87 Floating-point double-precision	8
x87 Floating-point extended-precision	8
x87 Floating-point save areas	2 or 4 (depends on operand size)

Error Code Returned. Zero.

Program Restart. #AC is a fault-type exception. The saved instruction pointer points to the instruction that caused the #AC.

8.2.18 #MC—Machine-Check Exception (Vector 18)

The #MC exception is model specific. Processor implementations are not required to support the #MC exception, and those implementations that do support #MC can vary in how the #MC exception mechanism works.

The exception is enabled by setting CR4.MCE to 1. The machine-check architecture can include model-specific masking for controlling the reporting of some errors. Refer to Chapter 9, “Machine Check Architecture,” for more information.

Error Code Returned. None. Error information is provided by model-specific registers (MSRs) defined by the machine-check architecture.

Program Restart. #MC is an abort-type exception. If the RIPV flag (RIP valid) is set in the MCG_Status MSR, the saved CS and rIP point to the instruction at which the interrupted process thread can be restarted. If RIPV is clear, there is no reliable way to restart the program. If the EIPV flag (EIP valid) is set in the MCG_Status MSR, the saved CS and rIP point to the process thread that caused the error. If EIPV is clear, the CS:rIP of the instruction causing the failure is not known or the machine check is not related to a specific instruction.

8.2.19 #XF—SIMD Floating-Point Exception (Vector 19)

The #XF exception is used to handle unmasked SSE floating-point exceptions. A #XF exception occurs when all of the following conditions are true:

- A SSE floating-point exception occurs. The exception causes the processor to set the appropriate exception-status bit in the MXCSR register to 1.
- The exception-mask bit in the MXCSR that corresponds to the SSE floating-point exception is clear (=0).
- CR4.OSXMMEXCPT=1, indicating that the operating system supports handling of SSE floating-point exceptions.

The exception-mask bits are used by software to specify the handling of SSE floating-point exceptions. When the corresponding mask bit is cleared to 0, an exception occurs under the control of the CR4.OSXMMEXCPT bit. However, if the mask bit is set to 1, the SSE floating-point unit responds in a default manner and execution proceeds normally.

The CR4.OSXMMEXCPT bit specifies the interrupt vector to be taken when an unmasked SSE floating-point exception occurs. When CR4.OSXMMEXCPT=1, the #XF interrupt vector is taken when an exception occurs. When CR4.OSXMMEXCPT=0, the #UD (undefined opcode) interrupt vector is taken when an exception occurs.

The SSE floating-point exceptions reported by the #XF exception are (including mnemonics):

- IE—Invalid-operation exception (also called #I).
- DE—Denormalized-operand exception (also called #D).
- ZE—Zero-divide exception (also called #Z).
- OE—Overflow exception (also called #O).
- UE—Underflow exception (also called #U).
- PE—Precision exception (also called #P or inexact-result exception).

Each type of SSE floating-point exception can be masked by setting the appropriate bits in the MXCSR register. #XF can also be disabled by clearing the CR4.OSXMMEXCPT bit to 0.

Error Code Returned. None. Exception information is provided by the SSE floating-point MXCSR register. See “Instruction Reference” in Volume 4 for more information on using this register.

Program Restart. #XF is a fault-type exception. Unlike the #MF exception, the #XF exception is precise. The saved instruction pointer points to the instruction that caused the #XF.

8.2.20 #CP—Control-Protection Exception (Vector 21)

A #CP exception is generated when shadow stacks are enabled (CR4.CET=1) and any of the following situations occur:

- For RET or IRET instructions, the return addresses on the shadow stack and the data stack do not match.
- An invalid supervisor shadow stack token is encountered by the CALL, RET, IRET, SETSSBSY or RSTORSSP instructions or during the delivery of an interrupt or exception.
- For inter-privilege RET and IRET instructions, the SSP is not 8-byte aligned, or the previous SSP from shadow stack is not 4-byte aligned or, in legacy or compatibility mode, is not less than 4GB.
- A task switch initiated by IRET where the incoming SSP is not aligned to 4 bytes or is not less than 4GB.

Error Code Returned. The #CP error code is pushed onto the control-protection exception-handler stack. See Section “Control-Protection Error Code” on page 254 for a description of this error code.

Program Restart. #CP is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #CP. See Section “Exceptions During a Task Switch” on page 253 for a description of what can happen if this exception occurs during a task switch.

8.2.21 #HV—Hypervisor Injection Exception (Vector 28)

The #HV exception may be injected by the hypervisor into a secure guest VM to notify the VM of pending events. See Section 15.36.16 for details.

8.2.22 #VC—VMM Communication Exception (Vector 29)

The #VC exception is generated when certain events occur inside a secure guest VM. See Section 15.35.5 for more details.

8.2.23 #SX—Security Exception (Vector 30)

The #SX exception is generated by security-sensitive events under SVM. See Section 15.28 for details.

8.2.24 User-Defined Interrupts (Vectors 32–255)

User-defined interrupts can be initiated either by system logic or software. They occur when:

- System logic signals an external interrupt request to the processor. The signaling mechanism and the method of communicating the interrupt vector to the processor are implementation dependent.
- Software executes an INT*n* instruction. The INT*n* instruction operand provides the interrupt vector number.

Both methods can be used to initiate an interrupt into vectors 0 through 255. However, because vectors 0 through 31 are defined or reserved by the AMD64 architecture, software should not use vectors in this range for purposes other than their defined use.

Error Code Returned. None.

Program Restart. The saved instruction pointer depends on the interrupt source:

- External interrupts are recognized on instruction boundaries. The saved instruction pointer points to the instruction immediately following the boundary where the external interrupt was recognized.
- If the interrupt occurs as a result of executing the INT*n* instruction, the saved instruction pointer points to the instruction after the INT*n*.

Masking. The ability to mask user-defined interrupts depends on the interrupt source:

- External interrupts can be masked using the RFLAGS.IF bit. Setting RFLAGS.IF to 1 enables external interrupts, while clearing RFLAGS.IF to 0 inhibits them.
- Software interrupts (initiated by the INT*n* instruction) cannot be disabled.

8.3 Exceptions During a Task Switch

An exception can occur during a task switch while loading a segment selector. Page faults can also occur when accessing a TSS. In these cases, the hardware task-switch mechanism completes loading the new task state from the TSS, and then triggers the appropriate exception mechanism. No other checks are performed. When this happens, the saved instruction pointer points to the first instruction in the new task.

In long mode, an exception cannot occur during a task switch, because the hardware task-switch mechanism is disabled.

8.4 Error Codes

The processor exception-handling mechanism reports error and status information for some exceptions using an error code. The error code is pushed onto the stack by the exception mechanism during the control transfer into the exception handler. The error code formats are described in the following sections.

8.4.1 Selector-Error Code

Figure 8-2 shows the format of the selector-error code.

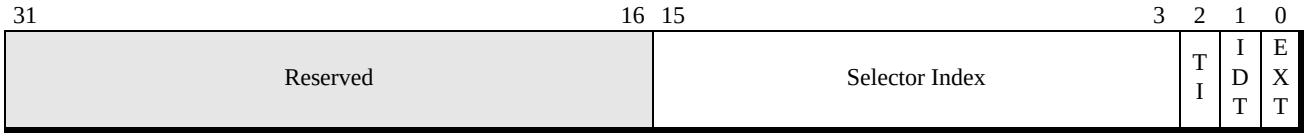


Figure 8-2. Selector Error Code

The information reported by the selector-error code includes:

- *EXT*—Bit 0. If this bit is set to 1, the exception source is external to the processor. If cleared to 0, the exception source is internal to the processor.
- *IDT*—Bit 1. If this bit is set to 1, the error-code selector-index field references a gate descriptor located in the interrupt-descriptor table (IDT). If cleared to 0, the selector-index field references a descriptor in either the global-descriptor table (GDT) or local-descriptor table (LDT), as indicated by the TI bit.
- *TI*—Bit 2. If this bit is set to 1, the error-code selector-index field references a descriptor in the LDT. If cleared to 0, the selector-index field references a descriptor in the GDT. The TI bit is relevant only when the IDT bit is cleared to 0.
- *Selector Index*—Bits 15:3. The selector-index field specifies the index into either the GDT, LDT, or IDT, as specified by the IDT and TI bits.

Some exceptions return a zero in the selector-error code.

8.4.2 Page-Fault Error Code

Figure 8-4 shows the format of the page-fault error code.

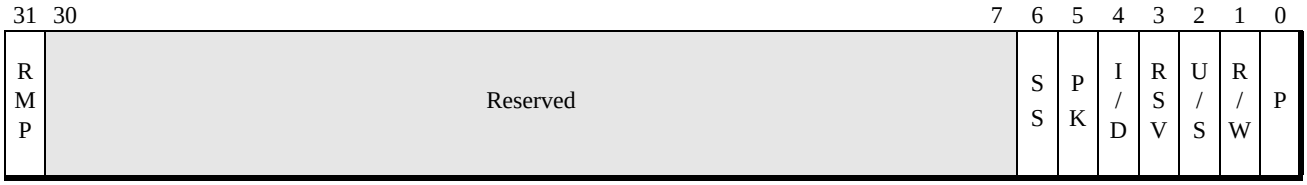


Figure 8-3. Page-Fault Error Code

The information reported by the page-fault error code includes:

- *P*—Bit 0. If this bit is cleared to 0, the page fault was caused by a not-present page. If this bit is set to 1, the page fault was caused by a page-protection violation.
- *R/W*—Bit 1. If this bit is cleared to 0, the access that caused the page fault is a memory read. If this bit is set to 1, the memory access that caused the page fault was a write. This bit does not necessarily indicate the cause of the page fault was a read or write violation.
- *U/S*—Bit 2. If this bit is cleared to 0, an access in supervisor mode (CPL=0, 1, or 2) caused the page fault. If this bit is set to 1, an access in user mode (CPL=3) caused the page fault. This bit does not necessarily indicate the cause of the page fault was a privilege violation.
- *RSV*—Bit 3. If this bit is set to 1, the page fault is a result of the processor reading a 1 from a reserved field within a page-translation-table entry. This type of page fault occurs only when CR4.PSE=1 or CR4.PAE=1. If this bit is cleared to 0, the page fault was not caused by the processor reading a 1 from a reserved field.
- *I/D*—Bit 4. If this bit is set to 1, it indicates that the access that caused the page fault was an instruction fetch. Otherwise, this bit is cleared to 0. This bit is only defined if no-execute feature is enabled (EFER.NXE=1 && CR4.PAE=1).
- *PK*—Bit 5. If this bit is set to 1, it indicates that a data access to a user-mode address caused a protection key violation. This fault only occurs if memory protection keys are enabled (CR4.PKE=1).
- *SS*—Bit 6. If this bit is set to 1, the page fault was caused by a shadow stack access. This bit is only set when the shadow stack feature is enabled (CR4.CET=1).
- *RMP*—Bit 31. If this bit is set to 1, the page fault is a result of the processor encountering an RMP violation. This type of page fault only occurs when SYSCFG[SecureNestedPagingEn]=1. If this bit is cleared to 0, the page fault was not caused by an RMP violation. See section 15.36.10 for additional information.

8.4.3 Control-Protection Error Code

Figure 8-4 shows the format of the #CP error code.

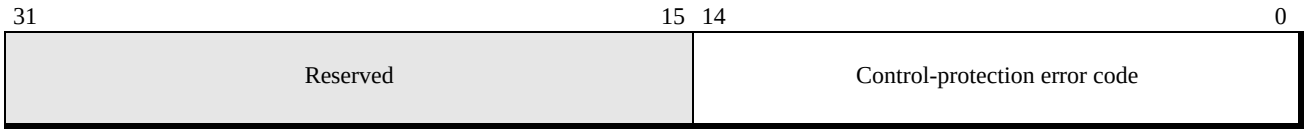


Figure 8-4. Control-Protection Error Code

The control-protection error codes are defined below:

Table 8-8. Control-Protection Error Codes

Error Code Value	Name	Cause
1	NEAR-RET	A RET (near) instruction encountered a return address mismatch.
2	FAR-RET/IRET	A RET (far) or IRET instruction encountered a return address mismatch.
3	RSTORSSP	An RSTORSSP instruction encountered an invalid shadow stack restore token.
4	SETSSBSY	A SETSSBSY instruction encountered an invalid supervisor shadow stack token.

8.5 Priorities

To allow for consistent handling of multiple-interrupt conditions, simultaneous interrupts are prioritized by the processor. The AMD64 architecture defines priorities between groups of interrupts, and interrupt prioritization within a group is implementation dependent. Table 8-9 shows the interrupt priorities defined by the AMD64 architecture.

When simultaneous interrupts occur, the processor transfers control to the highest-priority interrupt handler. Lower-priority interrupts from external sources are held pending by the processor, and they are handled after the higher-priority interrupt is handled. Lower-priority interrupts that result from internal sources are discarded. Those interrupts reoccur when the high-priority interrupt handler completes and transfers control back to the interrupted instruction. Software interrupts are discarded as well, and reoccur when the software-interrupt instruction is restarted.

Table 8-9. Simultaneous Interrupt Priorities

Interrupt Priority	Interrupt Condition	Interrupt Vector
(High) 0	Processor Reset	—
	Machine-Check Exception	18
1	External Processor Initialization (INIT)	—
	SMI Interrupt	
	External Clock Stop (Stpclk)	

Table 8-9. Simultaneous Interrupt Priorities (continued)

Interrupt Priority	Interrupt Condition	Interrupt Vector
2	Data, and I/O Breakpoint (Debug Register)	1
	Single-Step Execution Instruction Trap (RFLAGS.TF=1)	
3	Non-Maskable Interrupt	2
4	Maskable External Interrupt (INTR)	32–255
5	Instruction Breakpoint (Debug Register)	1
	Code-Segment-Limit Violation[1]	13
	Instruction-Fetch Page Fault[1]	14
6	Invalid Opcode Exception[1]	6
	Device-Not-Available Exception	7
	Instruction-Length Violation (> 15 Bytes)	13
7	Divide-by-zero Exception	0
	Invalid-TSS Exception	10
	Segment-Not-Present Exception	11
	Stack Exception	12
	General-Protection Exception	13
	Data-Access Page Fault	14
	Floating-Point Exception-Pending Exception	16
	Alignment-Check Exception	17
	SIMD Floating-Point Exception	19
	Control-Protection Exception	21
	Hypervisor Injection Exception	28
	VMM Communication Exception	29

Note:

1. This reflects the relative priority for faults encountered when fetching the first byte of an instruction. In the fetching and decoding of subsequent bytes of an instruction, if those bytes span the segment limit or cross into a non-executable or not-present page, the fetch will result in a #GP(0) fault or #PF as appropriate, preventing those bytes from being accessed. However, if the instruction can be determined to be invalid based just on the bytes preceding that boundary, a #UD fault may take priority. This behavior is model-dependent.

8.5.1 Floating-Point Exception Priorities

Floating-point exceptions (SSE and x87 floating-point) can be handled in one of two ways:

- Unmasked exceptions are reported in the appropriate floating-point status register, and a software-interrupt handler is invoked. See Section “#MF—x87 Floating-Point Exception-Pending (Vector 16)” on page 248 and Section “#XF—SIMD Floating-Point Exception (Vector 19)” on page 250 for more information on the floating-point interrupts.

- Masked exceptions are also reported in the appropriate floating-point status register. Instead of transferring control to an interrupt handler, however, the processor handles the exception in a default manner and execution proceeds.

If the processor detects more than one exception while executing a single floating-point instruction, it prioritizes the exceptions in a predictable manner. When responding in a default manner to masked exceptions, it is possible that the processor acts only on the high-priority exception and ignores lower-priority exceptions. In the case of vector (SIMD) floating-point instructions, priorities are set on sub-operations, not across all operations. For example, if the processor detects and acts on a QNaN operand in one sub-operation, the processor can still detect and act on a denormal operand in another sub-operation.

When reporting SSE floating-point exceptions before taking an interrupt or handling them in a default manner, the processor first classifies the exceptions as follows:

- *Input exceptions* include SNaN operand (#I), invalid operation (#I), denormal operand (#D), or zero-divide (#Z). Using a NaN operand with a maximum, minimum, compare, or convert instruction is also considered an input exception.
- *Output exceptions* include numeric overflow (#O), numeric underflow (#U), and precision (#P).

Using the above classification, the processor applies the following procedure to report the exceptions:

1. The exceptions for all sub-operations are prioritized.
2. The exception conditions for all sub-operations are logically ORed together to form a single set of exceptions covering all operations. For example, if two sub-operations produce a denormal result, only one denormal exception is reported.
3. If the set of exceptions includes any *unmasked* input exceptions, all input exceptions are reported in MCXSR, and no output exceptions are reported. Otherwise, all input and output exceptions are reported in MCXSR.
4. If any exceptions are unmasked, control is transferred to the appropriate interrupt handler.

Table 8-10 on page 257 lists the priorities for simultaneous floating-point exceptions.

Table 8-10. Simultaneous Floating-Point Exception Priorities

Exception Priority	Exception Condition	
(High) 0	SNaN Operand	#I
	NaN Operand of Maximum, Minimum, Compare, and Convert Instructions (Vector Floating-Point)	
	Stack Overflow (x87 Floating-Point)	
	Stack Underflow (x87 Floating-Point)	
1	QNaN Operand	—

Table 8-10. Simultaneous Floating-Point Exception Priorities (continued)

Exception Priority	Exception Condition	
2	Invalid Operation (Remaining Conditions)	#I
	Zero Divide	#Z
3	Denormal Operand	#D
4	Numeric Overflow	#O
	Numeric Underflow	#U
5 (Low)	Precision	#P

8.5.2 External Interrupt Priorities

The AMD64 architecture allows software to define up to 15 external interrupt-priority classes. Priority classes are numbered from 1 to 15, with priority-class 1 being the lowest and priority-class 15 the highest. The organization of these priority classes is implementation dependent. A typical method is to use the upper four bits of the interrupt vector number to define the priority. Thus, interrupt vector 53h has a priority of 5 and interrupt vector 37h has a priority of 3.

A new control register (CR8) is introduced by the AMD64 architecture for managing priority classes. This register, called the *task-priority register* (TPR), uses its four low-order bits to specify a task priority. The remaining 60 bits are reserved and must be written with zeros. Figure 8-5 shows the format of the TPR.

The TPR is available only in 64-bit mode.

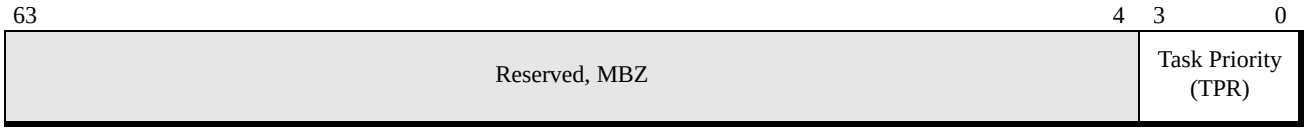


Figure 8-5. Task Priority Register (CR8)

System software can use the TPR register to temporarily block low-priority interrupts from interrupting a high-priority task. This is accomplished by loading TPR with a value corresponding to the highest-priority interrupt that is to be blocked. For example, loading TPR with a value of 9 (1001b) blocks all interrupts with a priority class of 9 or less, while allowing all interrupts with a priority class of 10 or more to be recognized. Loading TPR with 0 enables all external interrupts. Loading TPR with 15 (1111b) disables all external interrupts. The TPR is cleared to 0 on reset.

System software reads and writes the TPR using a MOV CR8 instruction. The MOV CR8 instruction requires a privilege level of 0. Programs running at any other privilege level cannot read or write the TPR, and an attempt to do so results in a general-protection exception (#GP).

A serializing instruction is not required after loading the TPR, because a new priority level is established when the MOV instruction completes execution. For example, assume two sequential TPR loads are performed, in which a low value is first loaded into TPR and immediately followed by a load of a higher value. Any pending, lower-priority interrupt enabled by the first MOV CR8 is recognized between the two MOVs.

The TPR is an architectural abstraction of the interrupt controller (IC), which prioritizes and manages external interrupt delivery to the processor. The IC can be an external system device, or it can be integrated on the chip like the local advanced programmable interrupt controller (APIC). Typically, the IC contains a priority mechanism similar, if not identical to, the TPR. The IC, however, is implementation dependent, and the underlying priority mechanisms are subject to change. The TPR, by contrast, is part of the AMD64 architecture.

Effect of IC on TPR. The features of the implementation-specific IC can impact the operation of the TPR. For example, the TPR might affect interrupt delivery only if the IC is enabled. Also, the mapping of an external interrupt to a specific interrupt priority is an implementation-specific behavior of the IC.

While the CR8 register provides the same functionality as the TPR at offset 80h of the local APIC, software should only use one mechanism to access the TPR. For example, updating the TPR with a write to the local APIC offset 0x80 but then reading it with a MOV CR8 is not guaranteed to return the same value that was written by the local APIC write.

8.6 Real-Mode Interrupt Control Transfers

In real mode, the IDT is a table of 4-byte entries, one entry for each of the 256 possible interrupts implemented by the system. The real mode IDT is often referred to as an *interrupt vector table*, or IVT. Table entries contain a far pointer (CS:IP pair) to an exception or interrupt handler. The base of the IDT is stored in the IDTR register, which is loaded with a value of 00h during a processor reset. Figure 8-6 on page 260 shows how the real-mode interrupt handler is located by the interrupt mechanism.

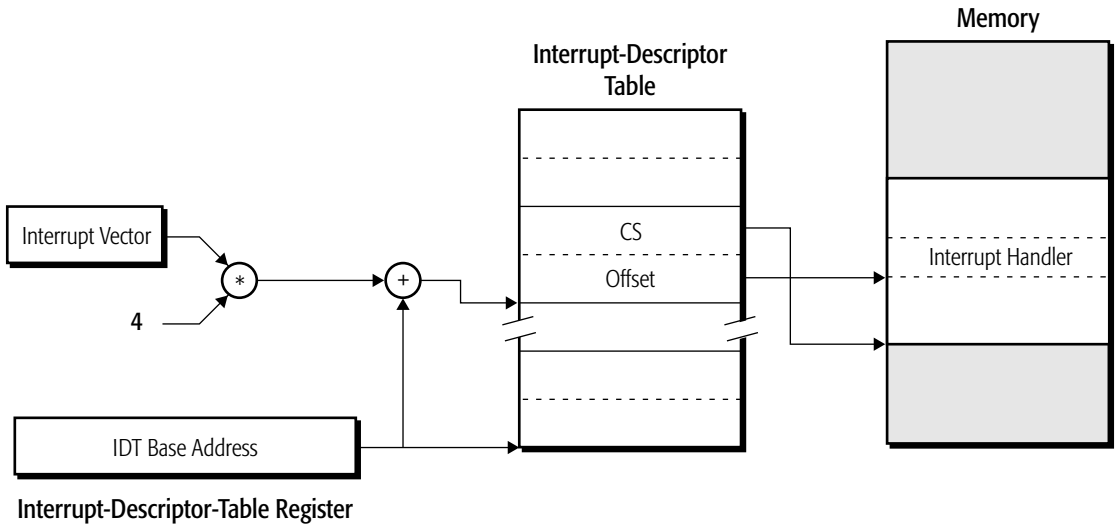


Figure 8-6. Real-Mode Interrupt Control Transfer

When an exception or interrupt occurs in real mode, the processor performs the following:

1. Pushes the FLAGS register (EFLAGS[15:0]) onto the stack.
2. Clears EFLAGS.IF to 0 and EFLAGS.TF to 0.
3. Saves the CS register and IP register (RIP[15:0]) by pushing them onto the stack.
4. Locates the interrupt-handler pointer (CS:IP) in the IDT by scaling the interrupt vector by four and adding the result to the value in the IDTR.
5. Transfers control to the interrupt handler referenced by the CS:IP in the IDT.

Figure 8-7 on page 261 shows the stack after control is transferred to the interrupt handler in real mode.

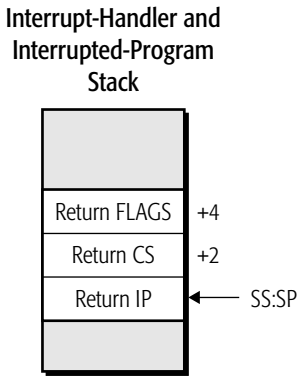


Figure 8-7. Stack After Interrupt in Real Mode

An IRET instruction is used to return to the interrupted program. When an IRET is executed, the processor performs the following:

1. Pops the saved CS value off the stack and into the CS register. The saved IP value is popped into RIP[15:0].
2. Pops the FLAGS value off of the stack and into EFLAGS[15:0].
3. Execution begins at the saved CS.IP location.

8.7 Legacy Protected-Mode Interrupt Control Transfers

In protected mode, the interrupt mechanism transfers control to an exception or interrupt handler through gate descriptors. In protected mode, the IDT is a table of 8-byte gate entries, one for each of the 256 possible interrupt vectors implemented by the system. Three gate types are allowed in the IDT:

- Interrupt gates.
- Trap gates.
- Task gates.

If a reference is made to any other descriptor type in the IDT, a general-protection exception (#GP) occurs.

Interrupt-gate control transfers are similar to CALLs and JMPs through call gates. The interrupt mechanism uses gates (interrupt, trap, and task) to establish protected entry-points into the exception and interrupt handlers.

The remainder of this chapter discusses control transfers through interrupt gates and trap gates. If the gate descriptor in the IDT is a task gate, a TSS-segment selector is referenced, and a task switch

occurs. See Chapter 12, “Task Management,” for more information on the hardware task-switch mechanism.

8.7.1 Locating the Interrupt Handler

When an exception or interrupt occurs, the processor scales the interrupt vector number by eight and uses the result as an offset into the IDT. If the gate descriptor referenced by the IDT offset is an interrupt gate or a trap gate, it contains a segment-selector and segment-offset field (see Section “Legacy Segment Descriptors” on page 88 for a detailed description of the gate-descriptor format and fields). These two fields perform the same function as the pointer operand in a far control-transfer instruction. The gate-descriptor segment-selector field points to the target code-segment descriptor located in either the GDT or LDT. The gate-descriptor segment-offset field is the instruction-pointer offset into the interrupt-handler code segment. The code-segment base taken from the code-segment descriptor is added to the gate-descriptor segment-offset field to create the interrupt-handler virtual address (linear address).

Figure 8-8 on page 263 shows how the protected-mode interrupt handler is located by the interrupt mechanism.

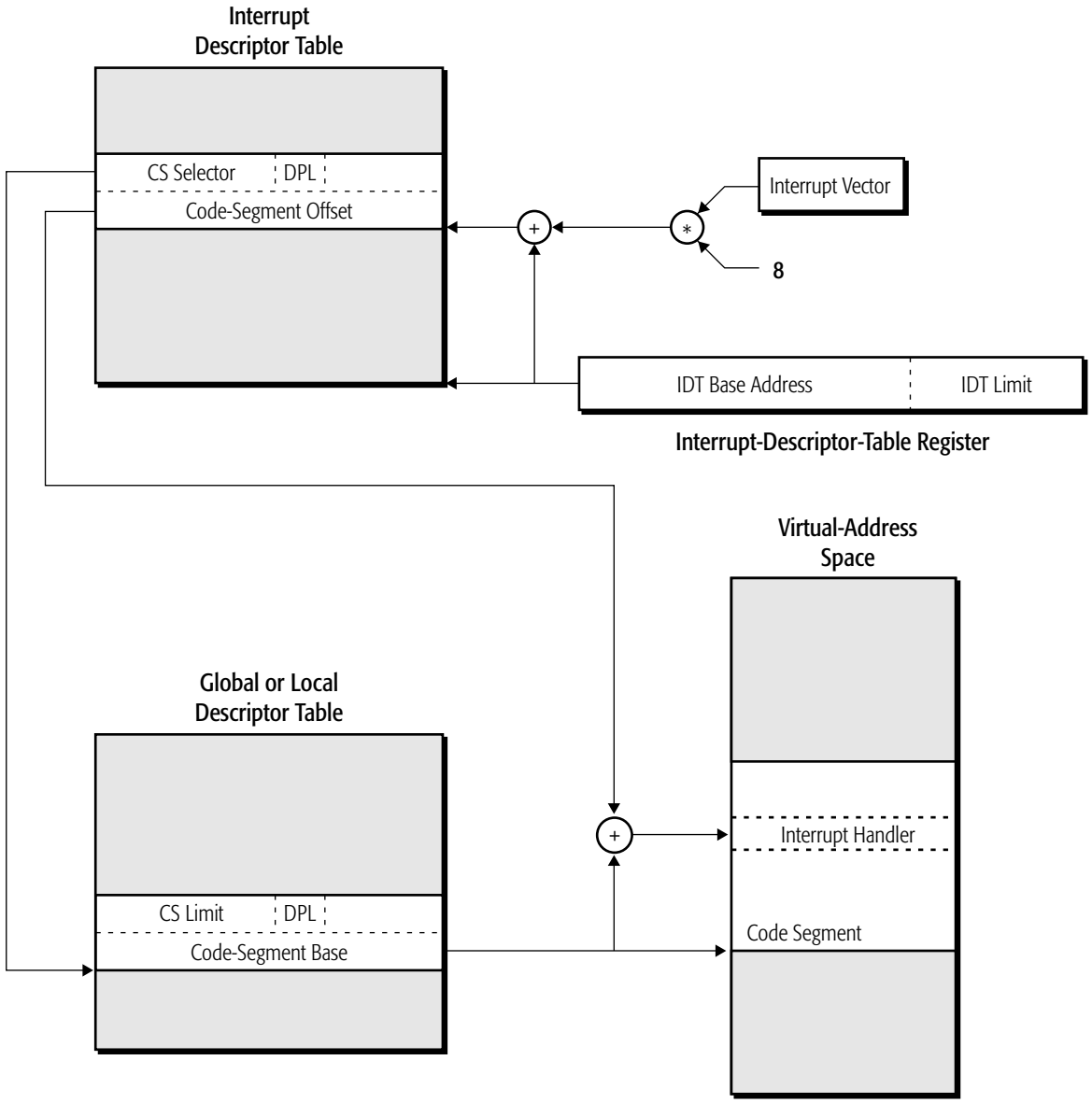


Figure 8-8. Protected-Mode Interrupt Control Transfer

8.7.2 Interrupt To Same Privilege

When a control transfer to an exception or interrupt handler at the same privilege level occurs (through an interrupt gate or a trap gate), the processor performs the following:

1. Pushes the EFLAGS register onto the stack.
2. Clears the TF, NT, RF, and VM bits in EFLAGS to 0.

3. The processor handles EFLAGS.IF based on the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, EFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, EFLAGS.IF is not modified.
4. Saves the return CS register and EIP register (RIP[31:0]) by pushing them onto the stack. The CS value is padded with two bytes to form a doubleword.
5. If the interrupt has an associated error code, the error code is pushed onto the stack.
6. The CS register is loaded from the segment-selector field in the gate descriptor, and the EIP is loaded from the offset field in the gate descriptor.
7. The interrupt handler begins executing with the instruction referenced by new CS:EIP.

Figure 8-9 shows the stack after control is transferred to the interrupt handler.

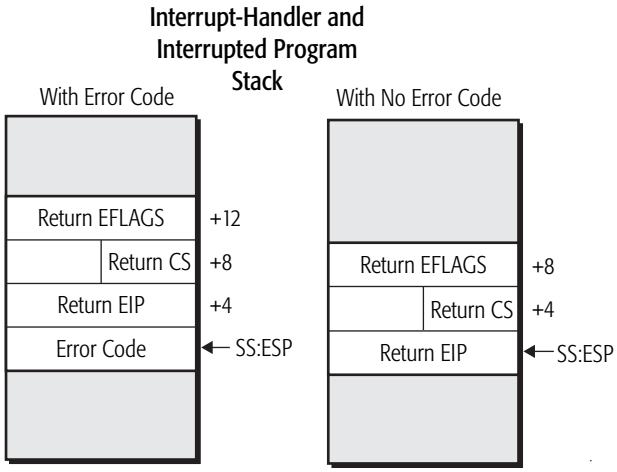


Figure 8-9. Stack After Interrupt to Same Privilege Level

8.7.3 Interrupt To Higher Privilege

When a control transfer to an exception or interrupt handler running at a higher privilege occurs (numerically lower CPL value), the processor performs a stack switch using the following steps:

1. The target CPL is read by the processor from the target code-segment DPL and used as an index into the TSS for selecting the new stack pointer (SS:ESP). For example, if the target CPL is 1, the processor selects the SS:ESP for privilege-level 1 from the TSS.
2. Pushes the return stack pointer (old SS:ESP) onto the new stack. The SS value is padded with two bytes to form a doubleword.
3. Pushes the EFLAGS register onto the new stack.
4. Clears the following EFLAGS bits to 0: TF, NT, RF, and VM.

5. The processor handles the EFLAGS.IF bit based on the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, EFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, EFLAGS.IF is not modified.
6. Saves the return-address pointer (CS:EIP) by pushing it onto the stack. The CS value is padded with two bytes to form a doubleword.
7. If the interrupt vector number has an error code associated with it, the error code is pushed onto the stack.
8. The CS register is loaded from the segment-selector field in the gate descriptor, and the EIP is loaded from the offset field in the gate descriptor.
9. The interrupt handler begins executing with the instruction referenced by new CS:EIP.

Figure 8-10 shows the new stack after control is transferred to the interrupt handler.

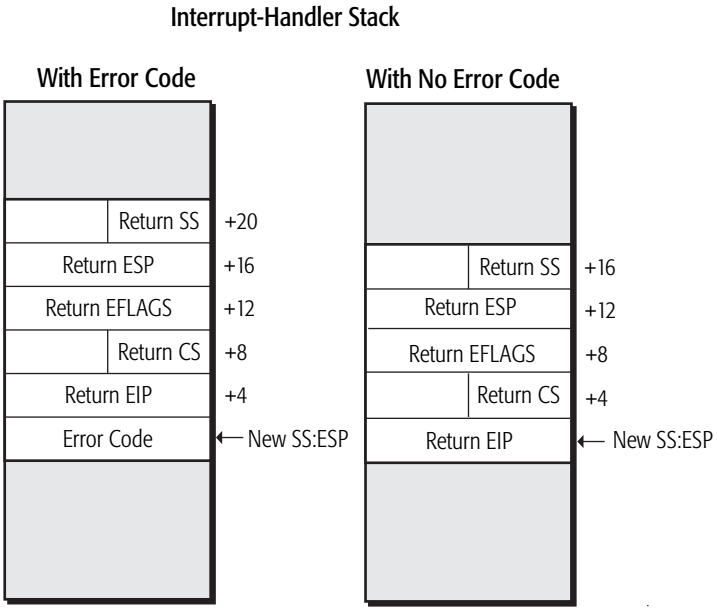


Figure 8-10. Stack After Interrupt to Higher Privilege

8.7.4 Privilege Checks

Before loading the CS register with the interrupt-handler code-segment selector (located in the gate descriptor), the processor performs privilege checks similar to those performed on call gates. The checks are performed when either conforming or nonconforming interrupt handlers are referenced:

1. The processor reads the gate DPL from the interrupt-gate or trap-gate descriptor. The gate DPL is the *minimum privilege-level* (numerically-highest value) needed by a program to access the gate. The processor compares the CPL with the gate DPL. The CPL must be numerically *less-than or equal-to* the gate DPL for this check to pass.

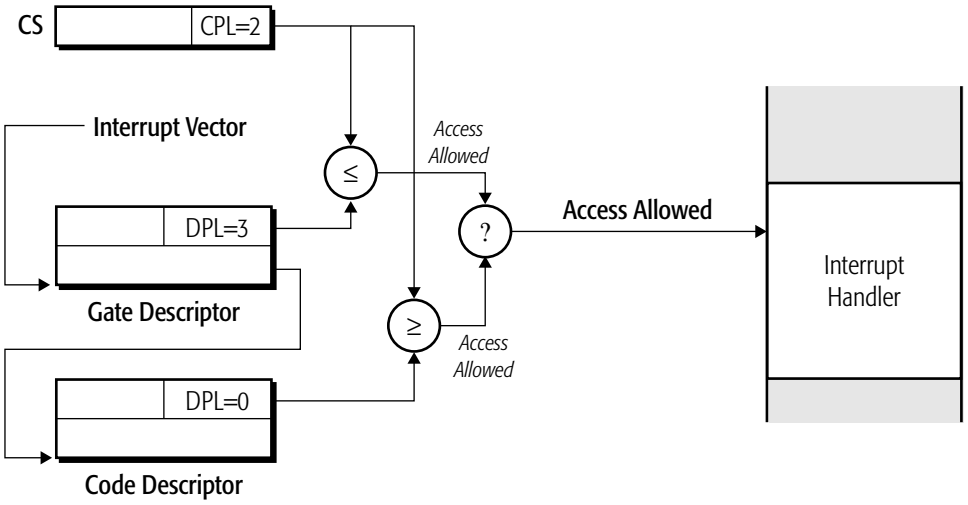
2. The processor compares the CPL with the interrupt-handler code-segment DPL. For this check to pass, the CPL must be numerically *greater-than or equal-to* the code-segment DPL. This check prevents control transfers to less-privileged interrupt handlers.

Unlike call gates, no RPL comparison takes place. This is because the gate descriptor is referenced in the IDT using the interrupt vector number rather than a selector, and no RPL field exists in the interrupt vector number.

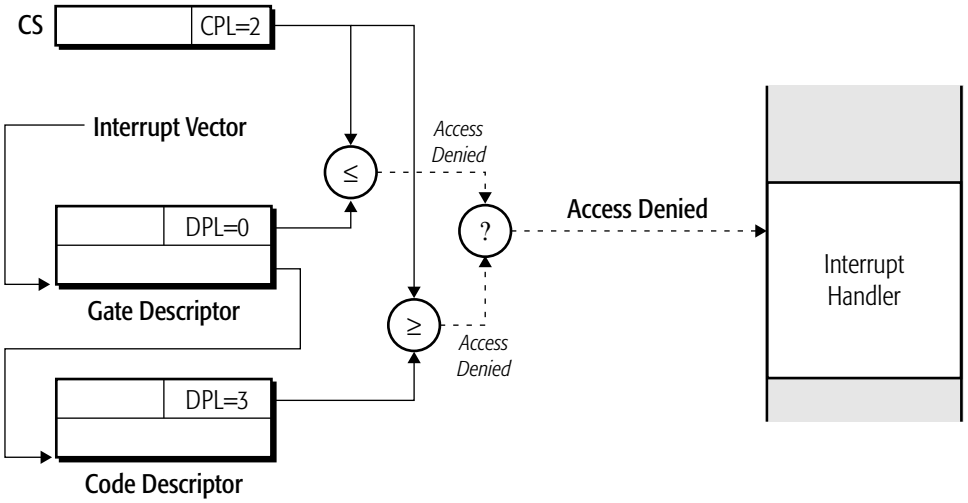
Exception and interrupt handlers should be made reachable from software running at any privilege level that requires them. If the gate DPL value is too low (requiring more privilege), or the interrupt-handler code-segment DPL is too high (runs at lower privilege), the interrupt control transfer can fail the privilege checks. Setting the gate DPL=3 and interrupt-handler code-segment DPL=0 makes the exception handler or interrupt handler reachable from any privilege level.

Figure 8-11 on page 267 shows two examples of interrupt privilege checks. In Example 1, both privilege checks pass:

- The interrupt-gate DPL is at the lowest privilege (3), which means that software running at any privilege level (CPL) can access the interrupt gate.
- The interrupt-handler code segment is at the highest-privilege level, as indicated by DPL=0. This means software running at any privilege can enter the interrupt handler through the interrupt gate.



Example 1: Privilege Check Passes



Example 2: Privilege Check Fails

Figure 8-11. Privilege-Check Examples for Interrupts

In Example 2, both privilege checks fail:

- The interrupt-gate DPL specifies that only software running at privilege-level 0 can access the gate. The current program does not have a high enough privilege level to access the interrupt gate, since its CPL is set at 2.

- The interrupt handler has a lower privilege (DPL=3) than the currently-running software (CPL=2). Transitions from more-privileged software to less-privileged software are not allowed, so this privilege check fails as well.

Although both privilege checks fail, only one such failure is required to deny access to the interrupt handler.

8.7.5 Returning From Interrupt Procedures

A return to an interrupted program should be performed using the IRET instruction. An IRET is a far return to a different code segment, with or without a change in privilege level. The actions of an IRET in both cases are described in the following sections.

IRET, Same Privilege. Before performing the IRET, the stack pointer must point to the return EIP. If there was an error code pushed onto the stack as a result of the exception or interrupt, that error code should have been popped off the stack earlier by the handler. The IRET reverses the actions of the interrupt mechanism:

1. Pops the return pointer off of the stack, loading both the CS register and EIP register (RIP[31:0]) with the saved values. The return code-segment RPL is read by the processor from the CS value stored on the stack to determine that an equal-privilege control transfer is occurring.
2. Pops the saved EFLAGS image off of the stack and into the EFLAGS register.
3. Transfers control to the return program at the target CS:EIP.

IRET, Less Privilege. If an IRET changes privilege levels, the return program must be at a lower privilege than the interrupt handler. The IRET in this case causes a stack switch to occur:

1. The return pointer is popped off of the stack, loading both the CS register and EIP register (RIP[31:0]) with the saved values. The return code-segment RPL is read by the processor from the CS value stored on the stack to determine that a lower-privilege control transfer is occurring.
2. The saved EFLAGS image is popped off of the stack and loaded into the EFLAGS register.
3. The return-program stack pointer is popped off of the stack, loading both the SS register and ESP register (RSP[31:0]) with the saved values.
4. Control is transferred to the return program at the target CS:EIP.

8.7.6 Shadow Stack Support for Interrupts and Exceptions

The operation of the shadow stack for an interrupt control transfer depends whether or not the interrupt handler runs at the same privilege or at a higher privilege level than the CPL when the interrupt or exception occurred.

Interrupt Control Transfer to Same Privilege Level. When a control transfer to a interrupt handler at the same privilege occurs, and the shadow stack feature is enabled for the current CPL, the processor pushes the interrupted procedure's CS and LIP (CS.base + EIP) onto the current shadow stack.

Interrupt Control Transfer to Higher Privilege Level. When a control transfer to a interrupt handler at a higher privilege occurs, the actions taken by the processor depend on the CPL when the interrupt or exception occurred.

- If the CPL = 3:

  - if shadow stacks are enabled in user-mode, the current SSP is saved to MSR PL3_SSP.

  - if shadow stacks are enabled in supervisor mode, a new SSP is loaded from MSR PLn_SSP (where n = the target CPL 0, 1 or 2).

  - the shadow stack token located at the base of the new shadow stack is checked, and if valid, the token's busy bit is set to 1.

  Note: the CS and LIP are not pushed onto the new shadow stack.

- If the CPL =1 or 2, and shadow stacks are enabled in supervisor mode:

  - the new SSP is loaded from MSR PLn_SSP (where n = the target CPL 0 or 1).

  - the shadow stack token located at the base of the new shadow stack is checked, and if valid, the token's busy bit is set to 1.

  - the CS, LIP and old SSP are pushed onto the new shadow stack.

For a detailed description of shadow stack operations, see Section “Shadow Stacks” on page 647.

8.8 Virtual-8086 Mode Interrupt Control Transfers

This section describes interrupt control transfers as they relate to virtual-8086 mode. Virtual-8086 mode is not supported by long mode. Therefore, the control-transfer mechanism described here is not applicable to long mode.

When a software interrupt occurs (not external interrupts, INT1, or INT3) while the processor is running in virtual-8086 mode (EFLAGS.VM=1), the control transfer that occurs depends on three system controls:

- *EFLAGS.IOPL*—This field controls interrupt handling based on the CPL. See Section “I/O Privilege Level Field (IOPL) Field” on page 54 for more information on this field.

  Setting IOPL<3 redirects the interrupt to the general-protection exception (#GP) handler.
- *CR4.VME*—This bit enables virtual-mode extensions. See Section “Virtual-8086 Mode Extensions (VME)” on page 48 for more information on this bit.
- *TSS Interrupt-Redirection Bitmap*—The TSS interrupt-redirection bitmap contains 256 bits, one for each possible INT*n* vector (software interrupt). When CR4.VME=1, the bitmap is used by the processor to direct interrupts to the handler provided by the currently-running 8086 program (bitmap entry is 0), or to the protected-mode operating-system interrupt handler (bitmap entry is 1). See Section “Legacy Task-State Segment” on page 367 for information on the location of this field within the TSS.

If IOPL<3, CR4.VME=1, and the corresponding interrupt redirection bitmap entry is 0, the processor uses the virtual-interrupt mechanism. See Section “Virtual Interrupts” on page 279 for more information on this mechanism.

Table 8-11 summarizes the actions of the above system controls on interrupts taken when the processor is running in virtual-8086 mode.

Table 8-11. Virtual-8086 Mode Interrupt Mechanisms

EFLAGS.IOPL	CR4.VME	TSS Interrupt Redirection Bitmap Entry	Interrupt Mechanism
0, 1, or 2	0	—	General-Protection Exception
	1	1	
	1	0	Virtual Interrupt
3	0	—	Protected-Mode Handler
	1	1	
	1	0	Virtual 8086 Handler

8.8.1 Protected-Mode Handler Control Transfer

Control transfers to protected-mode handlers from virtual-8086 mode differ from standard protected-mode transfers in several ways. The processor follows these steps in making the control transfer:

1. Reads the CPL=0 stack pointer (SS:ESP) from the TSS.
2. Pushes the GS, FS, DS, and ES selector registers onto the stack. Each push is padded with two bytes to form a doubleword.
3. Clears the GS, FS, DS, and ES selector registers to 0. This places a null selector in each of the four registers
4. Pushes the return stack pointer (old SS:ESP) onto the new stack. The SS value is padded with two bytes to form a doubleword.
5. Pushes the EFLAGS register onto the new stack.
6. Clears the following EFLAGS bits to 0: TF, NT, RF, and VM.
7. Handles EFLAGS.IF based on the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, EFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, EFLAGS.IF is not modified.
8. Pushes the return-address pointer (CS:EIP) onto the stack. The CS value is padded with two bytes to form a doubleword.
9. If the interrupt has an associated error code, pushes the error code onto the stack.
10. Loads the segment-selector field from the gate descriptor into the CS register, and loads the offset field from the gate descriptor into the EIP register.

11. Begins execution of the interrupt handler with the instruction referenced by the new CS:EIP.

Figure 8-12 shows the new stack after control is transferred to the interrupt handler with an error code.

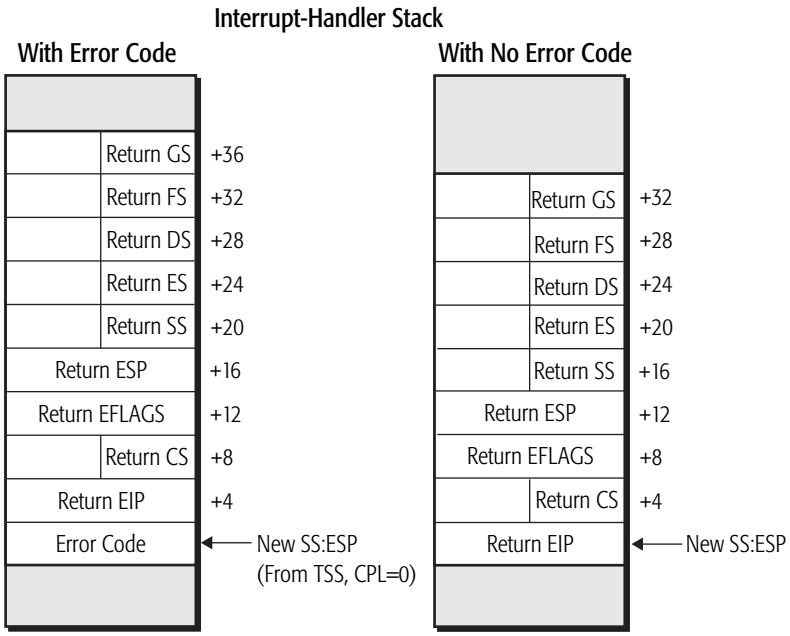


Figure 8-12. Stack After Virtual-8086 Mode Interrupt to Protected Mode

An IRET from privileged protected-mode software (CPL=0) to virtual-8086 mode reverses the stack-build process. After the return pointer, EFLAGS, and return stack-pointer are restored, the processor restores the ES, DS, FS, and GS registers by popping their values off the stack.

8.8.2 Virtual-8086 Handler Control Transfer

When a control transfer to an 8086 handler occurs from virtual-8086 mode, the processor creates an interrupt-handler stack identical to that created when an interrupt occurs in real mode (see Figure 8-7 on page 261). The processor performs the following actions during a control transfer:

1. Pushes the FLAGS register (EFLAGS[15:0]) onto the stack.
2. Clears the EFLAGS.IF and EFLAGS.RF bits to 0.
3. Saves the CS register and IP register (RIP[15:0]) by pushing them onto the stack.
4. Locates the interrupt-handler pointer (CS:IP) in the 8086 IDT by scaling the interrupt vector by four and adding the result to the virtual (linear) address 0. The value in the IDTR is not used.
5. Transfers control to the interrupt handler referenced by the CS:IP in the IDT.

An IRET from the 8086 handler back to virtual-8086 mode reverses the stack-build process.

8.9 Long-Mode Interrupt Control Transfers

The long-mode architecture expands the legacy interrupt-mechanism to support 64-bit operating systems and applications. These changes include:

- All interrupt handlers are 64-bit code and operate in 64-bit mode.
- The size of an interrupt-stack push is fixed at 64 bits (8 bytes).
- The interrupt-stack frame is aligned on a 16-byte boundary.
- The stack pointer, SS:RSP, is pushed unconditionally on interrupts, rather than conditionally based on a change in CPL.
- The SS selector register is loaded with a null selector as a result of an interrupt, if the CPL changes.
- The IRET instruction behavior changes, to unconditionally pop SS:RSP, allowing a null SS to be popped.
- A new interrupt stack-switch mechanism, called the interrupt-stack table or IST, is introduced.
- When shadow stacks are enabled, a new shadow stack-switch mechanism, called the Interrupt SSP Table or ISST, is introduced.

8.9.1 Interrupt Gates and Trap Gates

Only long-mode interrupt and trap gates can be referenced in long mode (64-bit mode and compatibility mode). The legacy 32-bit interrupt-gate and 32-bit trap-gate types (0Eh and 0Fh, as described in Section “System Descriptors” on page 99) are redefined in long mode as 64-bit interrupt-gate and 64-bit trap-gate types. 32-bit and 16-bit interrupt-gate and trap-gate types do not exist in long mode, and software is prohibited from using task gates. If a reference is made to any gate other than a 64-bit interrupt gate or a 64-bit trap gate, a general-protection exception (#GP) occurs.

The long-mode gate types are 16 bytes (128 bits) long. They are an extension of the legacy-mode gate types, allowing a full 64-bit segment offset to be stored in the descriptor. See Section “Legacy Segment Descriptors” on page 88 for a detailed description of the gate-descriptor format and fields.

8.9.2 Locating the Interrupt Handler

When an interrupt occurs in long mode, the processor multiplies the interrupt vector number by 16 and uses the result as an offset into the IDT. The gate descriptor referenced by the IDT offset contains a segment-selector and a 64-bit segment-offset field. The gate-descriptor segment-offset field contains the complete virtual address for the interrupt handler. The gate-descriptor segment-selector field points to the target code-segment descriptor located in either the GDT or LDT. The code-segment descriptor is only used for privilege-checking purposes and for placing the processor in 64-bit mode. The code segment-descriptor base field, limit field, and most attributes are ignored.

Figure 8-13 shows how the long-mode interrupt handler is located by the interrupt mechanism.

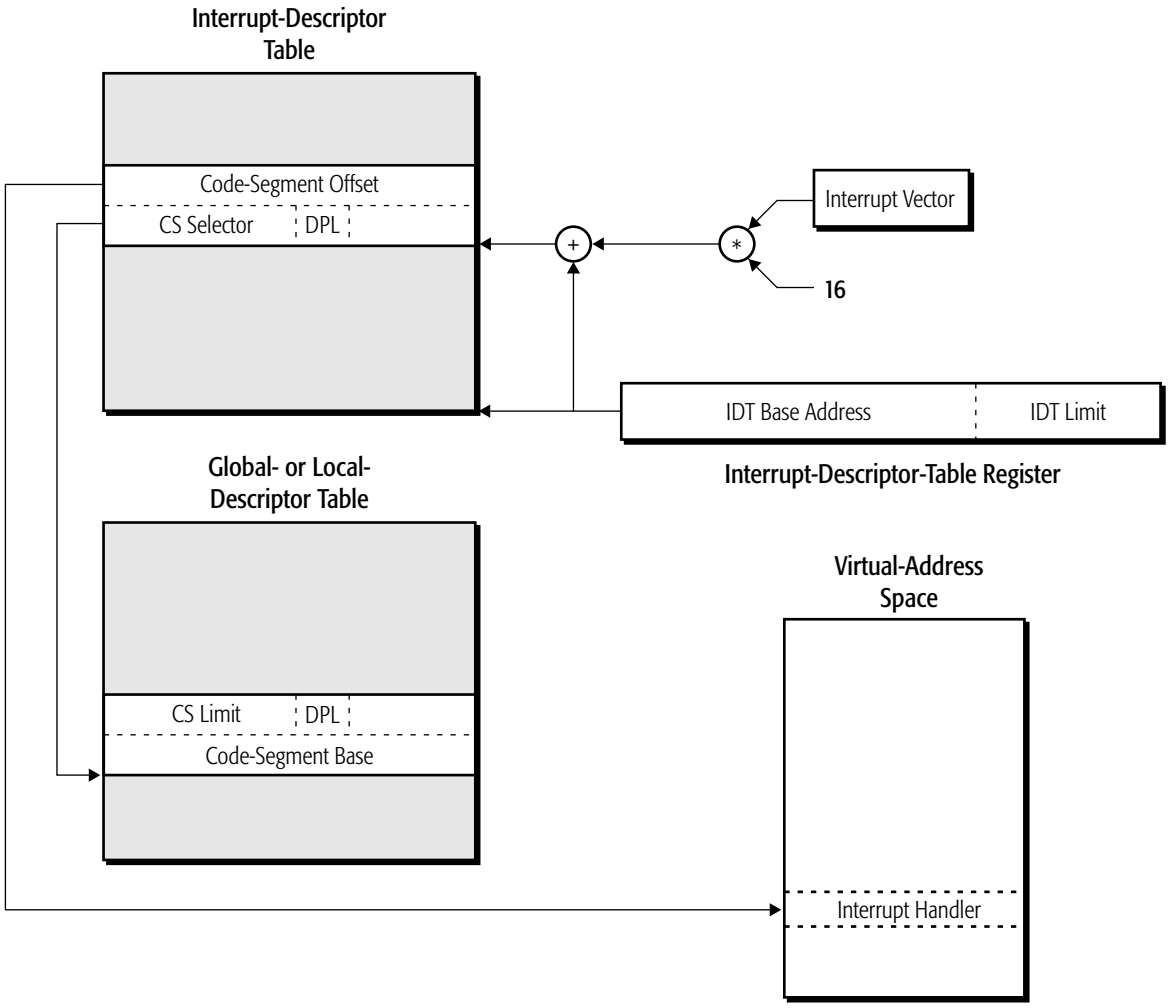


Figure 8-13. Long-Mode Interrupt Control Transfer

8.9.3 Interrupt Stack Frame

In long mode, the return-program stack pointer (SS:RSP) is always pushed onto the interrupt-handler stack, regardless of whether or not a privilege change occurs. Although the SS register is not used in 64-bit mode, SS is pushed to allow returns into compatibility mode. Pushing SS:RSP unconditionally presents operating systems with a consistent interrupt-stack-frame size for all interrupts, except for error codes. Interrupt service-routine entry points that handle interrupts generated by non-error-code interrupts can push an error code on the stack for consistency.

In long mode, when a control transfer to an interrupt handler occurs, the processor performs the following:

1. Aligns the new interrupt-stack frame by masking RSP with FFFF_FFFF_FFFF_FFF0h.
2. If IST field in interrupt gate is not 0, reads IST pointer into RSP.
3. If a privilege change occurs, the target DPL is used as an index into the long-mode TSS to select a new stack pointer (RSP).
4. If a privilege change occurs, SS is cleared to zero indicating a null selector.
5. Pushes the return stack pointer (old SS:RSP) onto the new stack. The SS value is padded with six bytes to form a quadword.
6. Pushes the 64-bit RFLAGS register onto the stack. The upper 32 bits of the RFLAGS image on the stack are written as zeros.
7. Clears the TF, NT, and RF bits in RFLAGS bits to 0.
8. Handles the RFLAGS.IF bit according to the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, RFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, RFLAGS.IF is not modified.
9. Pushes the return CS register and RIP register onto the stack. The CS value is padded with six bytes to form a quadword.
10. If the interrupt vector number has an error code associated with it, pushes the error code onto the stack. The error code is padded with four bytes to form a quadword.
11. Loads the segment-selector field from the gate descriptor into the CS register. The processor checks that the target code-segment is a 64-bit mode code segment.
12. Loads the offset field from the gate descriptor into the target RIP. The interrupt handler begins execution when control is transferred to the instruction referenced by the new RIP.

Figure 8-14 on page 275 shows the stack after control is transferred to the interrupt handler.

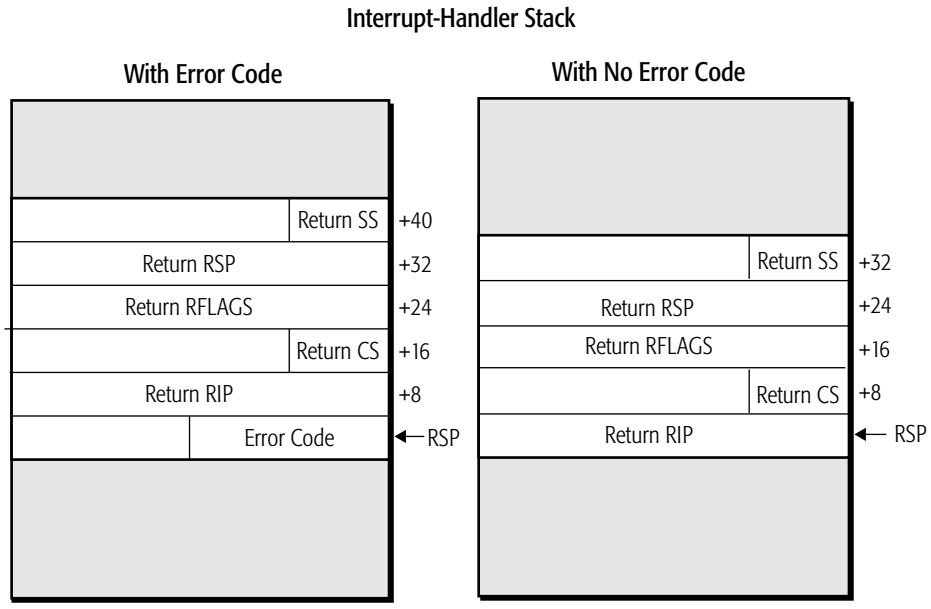


Figure 8-14. Long-Mode Stack After Interrupt—Same Privilege

Interrupt-Stack Alignment. In legacy mode, the interrupt-stack pointer can be aligned at any address boundary. Long mode, however, aligns the stack on a 16-byte boundary. This alignment is performed by the processor in hardware before pushing items onto the stack frame. The previous RSP is saved unconditionally on the new stack by the interrupt mechanism. A subsequent IRET instruction automatically restores the previous RSP.

Aligning the stack on a 16-byte boundary allows optimal performance for saving and restoring the 16-byte XMM registers. The interrupt handler can save and restore the XMM registers using the faster 16-byte aligned loads and stores (MOVAPS), rather than unaligned loads and stores (MOVUPS). Although the RSP alignment is always performed in long mode, it is only of consequence when the interrupted program is already running at CPL=0, and it is generally used only within the operating-system kernel. The operating system should put 16-byte aligned RSP values in the TSS for interrupts that change privilege levels.

Stack Switch. In long mode, the stack-switch mechanism differs slightly from the legacy stack-switch mechanism (see Section “Interrupt To Higher Privilege” on page 264). When stacks are switched during a long-mode privilege-level change resulting from an interrupt, a new SS descriptor is *not* loaded from the TSS. Long mode only loads an inner-level RSP from the TSS. However, the SS selector is loaded with a null selector, allowing nested control transfers, including interrupts, to be handled properly in 64-bit mode. The SS.RPL is set to the new CPL value. See Section “Nested IRETs to 64-Bit Mode Procedures” on page 278 for additional information.

The interrupt-handler stack that results from a privilege change in long mode looks identical to a long-mode stack when no privilege change occurs. Figure 8-15 shows the stack after the switch is performed and control is transferred to the interrupt handler.

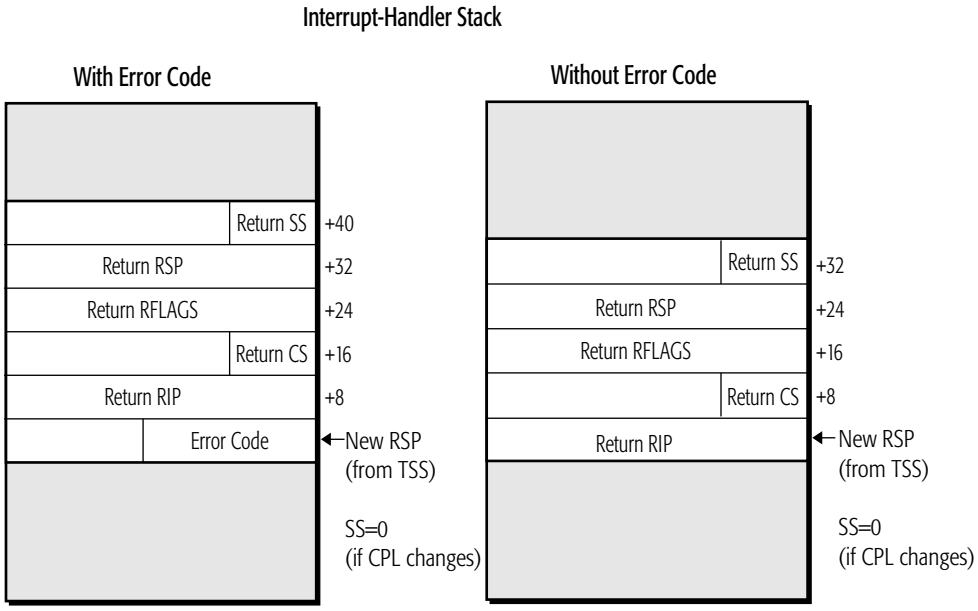


Figure 8-15. Long-Mode Stack After Interrupt—Higher Privilege

8.9.4 Interrupt-Stack Table

In long mode, a new interrupt-stack table (IST) mechanism is introduced as an alternative to the modified legacy stack-switch mechanism described above. The IST mechanism provides a method for specific interrupts, such as NMI, double-fault, and machine-check, to always execute on a known-good stack. In legacy mode, interrupts can use the hardware task-switch mechanism to set up a known-good stack by accessing the interrupt service routine through a task gate located in the IDT. However, the hardware task-switch mechanism is not supported in long mode.

When enabled, the IST mechanism unconditionally switches stacks. It can be enabled on an individual interrupt vector basis using a new field in the IDT gate-descriptor entry. This allows some interrupts to use the modified legacy mechanism, and others to use the IST mechanism. The IST mechanism is only available in long mode.

The IST mechanism uses new fields in the 64-bit TSS format and the long-mode interrupt-gate and trap-gate descriptors:

- Figure 12-8 on page 373 shows the format of the 64-bit TSS and the location of the seven IST pointers. The 64-bit TSS offsets from 24h to 5Bh provide space for seven IST pointers, each of which are 64 bits (8 bytes) long.

- The long-mode interrupt-gate and trap-gate descriptors define a 3-bit IST-index field in bits 2:0 of byte +4. Figure 4-24 on page 102 shows the format of long-mode interrupt-gate and trap-gate descriptors and the location of the IST-index field.

To enable the IST mechanism for a specific interrupt, system software stores a non-zero value in the interrupt gate-descriptor IST-index field. If the IST index is zero, the modified legacy stack-switching mechanism (described in the previous section) is used.

Figure 8-16 shows how the IST mechanism is used to create the interrupt-handler stack. When an interrupt occurs and the IST index is non-zero, the processor uses the index to select the corresponding IST pointer from the TSS. The IST pointer is loaded into the RSP to establish a new stack for the interrupt handler. The SS register is loaded with a null selector if the CPL changes and the SS.RPL is set to the new CPL value. After the stack is loaded, the processor pushes the old stack pointer, RFLAGS, the return pointer, and the error code (if applicable) onto the stack. Control is then transferred to the interrupt handler.

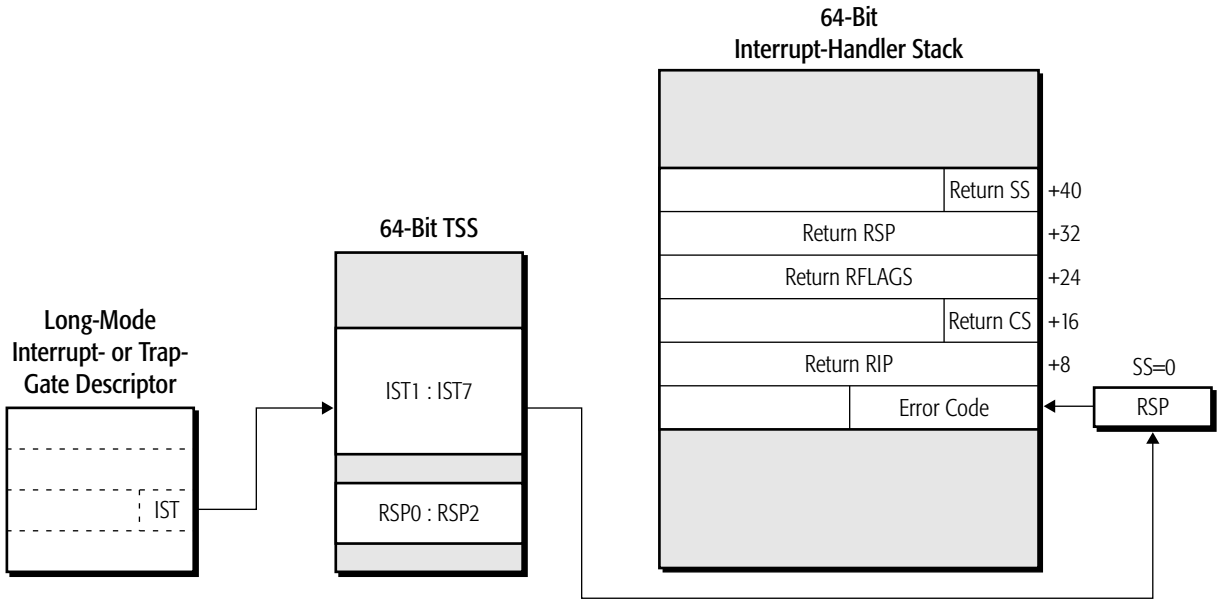


Figure 8-16. Long-Mode IST Mechanism

Software must make sure that an interrupt or exception handler using an IST pointer doesn't take another exception using the same IST pointer, as this will result in the first stack exception frame being overwritten.

8.4.9.1 Interrupt Shadow Stack Table.

When the shadow stack feature is enabled in long mode (EFER.LMA=1), a mechanism similar to the IST is provided to switch shadow stacks. This mechanism, the Interrupt Shadow Stack Table (ISST) is described in Section “Shadow Stacks” on page 647.

8.9.5 Returning From Interrupt Procedures

As with legacy mode, a return to an interrupted program in long mode should be performed using the IRET instruction. However, in long mode, the IRET semantics are different from legacy mode:

- *In 64-bit mode*, IRET pops the return-stack pointer unconditionally off the interrupt-stack frame and into the SS:RSP registers. This reverses the action of the long-mode interrupt mechanism, which saves the stack pointer whether or not a privilege change occurs. IRET also allows a null selector to be popped off the stack and into the SS register. See Section “Nested IRETs to 64-Bit Mode Procedures” on page 278 for additional information.
- *In compatibility mode*, IRET behaves as it does in legacy mode. The SS:ESP is popped off the stack only if a control transfer to less privilege (numerically greater CPL) is performed. Otherwise, it is assumed that a stack pointer is not present on the interrupt-handler stack.

The long-mode interrupt mechanism always uses a 64-bit stack when saving values for the interrupt handler, and the interrupt handler is always entered in 64-bit mode. To work properly, an IRET used to exit the 64-bit mode interrupt-handler requires a series of eight-byte pops off the stack. This is accomplished by using a 64-bit operand-size prefix with the IRET instruction. The default stack size assumed by an IRET in 64-bit mode is 32 bits, so a 64-bit REX prefix is needed by 64-bit mode interrupt handlers.

Nested IRETs to 64-Bit Mode Procedures. In long mode, an interrupt causes a null selector to be loaded into the SS register if the CPL changes (this is the same action taken by a far CALL in long mode). If the interrupt handler performs a far call, or is itself interrupted, the null SS selector is pushed onto the stack frame, and another null selector is loaded into the SS register. Using a null selector in this way allows the processor to properly handle returns nested within 64-bit-mode procedures and interrupt handlers.

The null selector enables the processor to properly handle nested returns to 64-bit mode (which do not use the SS register), and returns to compatibility mode (which do use the SS register). Normally, an IRET that pops a null selector into the SS register causes a general-protection exception (#GP) to occur. However, in long mode, the null selector indicates the existence of nested interrupt handlers and/or privileged software in 64-bit mode. Long mode allows an IRET to pop a null selector into SS from the stack under the following conditions:

- The target mode is 64-bit mode.
- The target CPL<3.

In this case, the processor does not load an SS descriptor, and the null selector is loaded into SS without causing a #GP exception.

8.10 Virtual Interrupts

The term *virtual interrupts* includes two classes of extensions to the interrupt-handling mechanism:

- *Virtual-8086 Mode Extensions (VME)*—These allow virtual interrupts and interrupt redirection in virtual-8086 mode. VME has no effect on protected-mode programs.
- *Protected-Mode Virtual Interrupts (PVI)*—These allow virtual interrupts in protected mode when CPL=3. Interrupt redirection is not available in protected mode. PVI has no effect on virtual-8086-mode programs.

Because virtual-8086 mode is not supported in long mode, VME extensions are not supported in long mode. PVI extensions are, however, supported in long mode.

8.10.1 Virtual-8086 Mode Extensions

The virtual-8086-mode extensions (VME) enable performance enhancements for 8086 programs running as protected tasks in virtual-8086 mode. These extensions are enabled by setting CR4.VME (bit 0) to 1. The extensions enabled by CR4.VME are:

- Virtualizing control and notification of maskable external interrupts with the EFLAGS VIF (bit 19) and VIP (bit 20) bits.
- Selective interception of software interrupts (INT*n* instructions) using the TSS interrupt redirection bitmap (IRB).

Background. Legacy-8086 programs expect to have full access to the EFLAGS interrupt flag (IF) bit, allowing programs to enable and disable maskable external interrupts. When those programs run in virtual-8086 mode under a multitasking protected-mode environment, it can disrupt the operating system if programs enable or disable interrupts for their own purposes. This is particularly true if interrupts associated with one program can occur during execution of another program. For example, a program could request that an area of memory be copied to disk. System software could suspend the program before external hardware uses an interrupt to acknowledge that the block has been copied. System software could subsequently start a second program which enables interrupts. This second program could receive the external interrupt indicating that the memory block of the first program has been copied. If that were to happen, the second program would probably be unprepared to handle the interrupt properly.

Access to the IF bit must be managed by system software on a task-by-task basis to prevent corruption of system resources. In order to completely manage the IF bit, system software must be able to interrupt all instructions that can read or write the bit. These instructions include STI, CLI, PUSHF, POPF, INT*n*, and IRET. These instructions are part of an instruction class that is *IOPL-sensitive*. The processor takes a general-protection exception (#GP) whenever an IOPL-sensitive instruction is executed and the EFLAGS.IOPL field is less than the CPL. Because all virtual-8086 programs run at CPL=3, system software can interrupt all instructions that modify the IF bit by setting IOPL<3.

System software maintains a virtual image of the IF bit for each virtual-8086 program by emulating the actions of IOPL-sensitive instructions that modify the IF bit. When an external maskable-interrupt occurs, system software checks the state of the IF image for the current virtual-8086 program to determine whether the program is masking interrupts. If the program is masking interrupts, system software saves the interrupt information until the virtual-8086 program attempts to re-enable interrupts. When the virtual-8086 program unmasks interrupts with an IOPL-sensitive instruction, system software traps the action with the #GP handler.

The performance of a processor can be significantly degraded by the overhead of trapping and emulating IOPL-sensitive instructions, and the overhead of maintaining images of the IF bit for each virtual-8086 program. This performance loss can be eliminated by running virtual-8086 programs with IOPL set to 3, thus allowing changes to the real IF flag from any privilege level. Unfortunately, this can leave critical system resources unprotected.

In addition to the performance problems caused by virtualizing the IF bit, software interrupts (INT*n* instructions) cannot be masked by the IF bit or virtual copies of the IF bit. The IF bit only affects maskable external interrupts. Software interrupts in virtual-8086 mode are normally directed to the real mode interrupt vector table (IVT), but it can be desirable to redirect certain interrupts to the protected-mode interrupt-descriptor table (IDT).

The virtual-8086-mode extensions are designed to support both external interrupts and software interrupts, with mechanisms that preserve high performance without compromising protection. Virtualization of external interrupts is supported using two bits in the EFLAGS register: the virtual-interrupt flag (VIF) bit and the virtual-interrupt pending (VIP) bit. Redirection of software interrupts is supported using the interrupt-redirection bitmap (IRB) in the TSS. A separate TSS can be created for each virtual-8086 program, allowing system software to control interrupt redirection independently for each virtual-8086 program.

VIF and VIP Extensions for External Interrupts. When VME extensions are enabled, the IF-modifying instructions normally trapped by system software are allowed to execute. However, instead of modifying the IF bit, they modify the EFLAGS VIF bit. This leaves control over maskable interrupts to the system software. It can also be used as an indicator to system software that the virtual-8086 program is able to, or is expecting to, receive external interrupts.

When an unmasked external interrupt occurs, the processor transfers control from the virtual-8086 program to a protected-mode interrupt handler. If the interrupt handler determines that the interrupt is for the virtual-8086 program, it can check the state of the VIF bit in the EFLAGS value pushed on the stack for the virtual-8086 program. If the VIF bit is set (indicating the virtual-8086 program attempted to unmask interrupts), system software can allow the interrupt to be handled by the appropriate virtual-8086 interrupt handler.

If the VIF bit is clear (indicating the virtual-8086 program attempted to mask interrupts) and the interrupt is for the virtual-8086 program, system software can hold the interrupt pending. System software holds an interrupt pending by saving appropriate information about the interrupt, such as the interrupt vector, and setting the virtual-8086 program's VIP bit in the EFLAGS image on the stack. When the virtual-8086 program later attempts to set IF, the previously set VIP bit causes a general-

protection exception (#GP) to occur. System software can then pass the saved interrupt information to the virtual-8086 interrupt handler.

To summarize, when the VME extensions are enabled (CR4.VME=1), the VIF and VIP bits are set and cleared as follows:

- *VIF Bit*—This bit is set and cleared by the processor in virtual-8086 mode in response to an attempt by a virtual-8086 program to set and clear the EFLAGS.IF bit. VIF is used by system software to determine whether a maskable external interrupt should be passed on to the virtual-8086 program, emulated by system software, or held pending. VIF is also cleared during software interrupts through interrupt gates, with the original VIF value preserved in the EFLAGS image on the stack.
- *VIP Bit*—System software sets and clears this bit in the EFLAGS image saved on the stack after an interrupt. It can be set when an interrupt occurs for a virtual-8086 program that has a clear VIF bit. The processor examines the VIP bit when an attempt is made by the virtual-8086 program to set the IF bit. If VIP is set when the program attempts to set IF, a general-protection exception (#GP) occurs *before* execution of the IF-setting instruction. System software must clear VIP to avoid repeated #GP exceptions when returning to the interrupted instruction.

The VIF and VIP bits can be used by system software to minimize the overhead associated with managing maskable external interrupts because virtual copies of the IF flag do not have to be maintained by system software. Instead, VIF and VIP are maintained during context switches along with the remaining EFLAGS bits.

Table 8-12 on page 283 shows how the behavior of instructions that modify the IF bit are affected by the VME extensions.

Interrupt Redirection of Software Interrupts. In virtual-8086 mode, software interrupts (INT*n* instructions) are trapped using a #GP exception handler if the IOPL is less than 3 (the CPL for virtual-8086 mode). This allows system software to interrupt and emulate 8086-interrupt handlers. System software can set the IOPL to 3, in which case the INT*n* instruction is vectored through a gate descriptor in the protected-mode IDT. System software can use the gate to control access to the virtual-8086 mode interrupt vector table (IVT), or to redirect the interrupt to a protected-mode interrupt handler.

When VME extensions are enabled, for INT*n* instructions to execute normally, vectoring directly to a virtual-8086 interrupt handler through the virtual-8086 IVT (located at address 0 in the virtual-address space of the task). For security or performance reasons, however, it can be necessary to intercept INT*n* instructions on a vector-specific basis to allow servicing by protected-mode interrupt handlers. This is performed by using the interrupt-redirection bitmap (IRB), located in the TSS and enabled when CR4.VME=1. The IRB is available only in virtual-8086 mode.

Figure 12-6 on page 368 shows the format of the TSS, with the interrupt redirection bitmap located near the top. The IRB contains 256 bits, one for each possible software-interrupt vector. The most-significant bit of the IRB controls interrupt vector 255, and is located immediately before the IOPB base. The least-significant bit of the IRB controls interrupt vector 0.

The bits in the IRB function as follows:

- When set to 1, the INT*n* instruction behaves as if the VME extensions are not enabled. The interrupt is directed through the IDT to a protected-mode interrupt handler if IOPL=3. If IOPL<3, the INT*n* causes a #GP exception.
- When cleared to 0, the INT*n* instruction is directed through the IVT for the virtual-8086 program to the corresponding virtual-8086 interrupt handler.

Only software interrupts can be redirected using the IRB mechanism. External interrupts are asynchronous events that occur outside the context of a virtual-8086 program. Therefore, external interrupts require system-software intervention to determine the appropriate context for the interrupt. The VME extensions described in Section “VIF and VIP Extensions for External Interrupts” on page 280 are provided to assist system software with external-interrupt intervention.

8.10.2 Protected Mode Virtual Interrupts

The protected-mode virtual-interrupt (PVI) bit in CR4 enables support for interrupt virtualization in protected mode. When enabled, the processor maintains program-specific VIF and VIP bits similar to the manner defined by the virtual-8086 mode extensions (VME). However, unlike VME, only the STI and CLI instructions are affected by the PVI extension. When a program is running at CPL=3, it can use STI and CLI to set and clear its copy of the VIF flag without causing a general-protection exception. The last section of Table 8-12 on page 283 describes the behavior of instructions that modify the IF bit when PVI extensions are enabled.

The interrupt redirection bitmap (IRB) defined by the VME extensions is not supported by the PVI extensions.

8.10.3 Effect of Instructions that Modify EFLAGS.IF

Table 8-12 on page 283 shows how the behavior of instructions that modify the IF bit are affected by the VME and PVI extensions. The table columns specify the following:

- *Operating Mode*—the processor mode in effect when the instruction is executed.
- *Instruction*—the IF-modifying instruction.
- *IOPL*—the value of the EFLAGS.IOPL field.
- *VIP*—the value of the EFLAGS.VIP bit.
- *#GP*—indicates whether the conditions in the first four columns cause a general-protection exception (#GP) to occur.
- *Effect on IF Bit*—indicates the effect the conditions in the first four columns have on the EFLAGS.IF bit and the image of EFLAGS.IF on the stack.
- *Effect on VIF Bit*—indicates the effect the conditions in the first four columns have on the EFLAGS.VIF bit and the image of EFLAGS.VIF on the stack.

Table 8-12. Effect of Instructions that Modify the IF Bit

Operating Mode	Instruction	IOPL	VIP	#GP	Effect on IF Bit	Effect on VIF Bit
Real Mode *CR0.PE=0* *EFLAGS.VM=0* *CR4.VME=0* *CR4.PVI=0*	CLI			no	IF = 0	
	STI				IF = 1	
	PUSHF				EFLAGS.IF Stack Image = IF	
	POPF				IF = EFLAGS.IF stack image	
	INT*n*				EFLAGS.IF Stack Image = IF IF = 0	
	IRET				IF = EFLAGS.IF Stack Image	
Protected Mode *CR0.PE=1* *EFLAGS.VM=0* *CR4.VME=x* *CR4.PVI=0*	CLI	³CPL		no	IF = 0	
		<CPL		yes	—	
	STI	³CPL		no	IF = 1	
		<CPL		yes	—	
	PUSHF	x		no	EFLAGS.IF Stack Image = IF	
	POPF	³CPL			IF = EFLAGS.IF Stack Image	
		<CPL			No Change	
	INT*n* gate	x			EFLAGS.IF Stack Image = IF IF = 0	
	IRET				IF = EFLAGS.IF Stack Image	
	IRETD					

Note:

Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.
"x" indicates the value of the bit is a "don't care".
"—" indicates the instruction causes a general-protection exception (#GP).

Note:

1. *If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.*
2. *If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.*

Table 8-12. Effect of Instructions that Modify the IF Bit (continued)

Operating Mode	Instruction	IOPL	VIP	#GP	Effect on IF Bit	Effect on VIF Bit
Virtual-8086 Mode *CR0.PE=1* *EFLAGS.VM=1* *CR4.VME=0* *CR4.PVI=x*	CLI	3		no	IF = 0	
		< 3		yes	—	
	STI	3		no	IF = 1	
		< 3		yes	—	
	PUSHF	3		no	EFLAGS.IF Stack Image = IF	
		< 3		yes	—	
	POPF	3		no	IF = EFLAGS.IF Stack Image	
		< 3		yes	—	
	INT*n* gate	3		no	EFLAGS.IF Stack Image = IF IF = 0	
		< 3		yes	—	
	IRET	3		no	IF = EFLAGS.IF Stack Image	
		< 3		yes	—	
	IRETD	3		no	IF = EFLAGS.IF Stack Image	
		< 3		yes	—	

Note:
Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.
"x" indicates the value of the bit is a "don't care".
"—" indicates the instruction causes a general-protection exception (#GP).

Note:
1. *If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.*
2. *If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.*

Table 8-12. Effect of Instructions that Modify the IF Bit (continued)

Operating Mode	Instruction	IOPL	VIP	#GP	Effect on IF Bit	Effect on VIF Bit
Virtual-8086 Mode *with VME Extensions* *CR0.PE=1* *EFLAGS.VM=1* *CR4.VME=1* *CR4.PVI=x*	CLI	3	x	no	IF = 0	No Change
		<3			No Change	VIF = 0
	STI	3	x	no	IF = 1	No Change
		<3	0	no	No Change	VIF = 1
			1	yes	—	
	PUSHF	3	x	no	EFLAGS.IF Stack Image = IF	Not Pushed
		<3			Not Pushed	EFLAGS.IF Stack Image = VIF
	PUSHFD	3	x	no	EFLAGS.IF Stack Image = IF	EFLAGS.VIF Stack Image = VIF
		<3		yes	—	
	POPF	3	x	no	IF = EFLAGS.IF Stack Image	No Change
		<3	0	no	No Change	VIF = EFLAGS.IF Stack Image
			1	yes	—	
	POPFD	3	x	no	IF = EFLAGS.IF Stack Image	No Change
		<3		yes	—	
	INT*n* gate	3	x	no	EFLAGS.IF Stack Image = IF IF = 0	No Change
		<3			No Change	EFLAGS.IF Stack Image = VIF VIF = 0
	IRET	3	x	no	IF = EFLAGS.IF Stack Image	No Change
		<3	0	no	No Change	VIF = EFLAGS.IF Stack Image
			1	no[1]	No Change	VIF = EFLAGS.IF Stack Image
				yes[2]	—	
	IRETD	3	x	no	IF = EFLAGS.IF Stack Image	VIF = EFLAGS.IF Stack Image
		<3		yes	—	

Note:

Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.
"x" indicates the value of the bit is a "don't care".
"—" indicates the instruction causes a general-protection exception (#GP).

Note:

1. *If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.*
2. *If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.*

Table 8-12. Effect of Instructions that Modify the IF Bit (continued)

Operating Mode	Instruction	IOPL	VIP	#GP	Effect on IF Bit	Effect on VIF Bit
Protected Mode *with PVI Extensions* *CR0.PE=1* *EFLAGS.VM=0* *CR4.VME=x* *CR4.PVI=1* *CPL=3*	CLI	3	x	no	IF = 0	No Change
		<3			No Change	VIF = 0
	STI	3	x	no	IF = 1	No Change
		<3	0	no	No Change	VIF = 1
			1	yes	—	
	PUSHF	x	x	no	EFLAGS.IF Stack Image = IF	Not Pushed
	PUSHFD					EFLAGS.VIF Stack Image = VIF
	POPF				IF = EFLAGS.IF Stack Image	No Change
	POPFD					VIF = 0
	INT*n* gate				EFLAGS.IF Stack Image = IF IF = 0 (if interrupt gate)	No Change
	IRET				IF = EFLAGS.IF Stack Image	No Change
	IRETD					VIF = EFLAGS.VIF Stack Image

Note:

Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.
"x" indicates the value of the bit is a "don't care".
"—" indicates the instruction causes a general-protection exception (#GP).

Note:

1. *If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.*
2. *If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.*

9 Machine Check Architecture

The AMD64 Machine Check Architecture (MCA) plays a vital role in the reliability, availability, and serviceability (RAS) of AMD processors, as well as the RAS of the computer systems in which they are embedded. MCA defines the facilities by which processor and system hardware errors are logged and reported to system software. This allows system software to serve a strategic role in recovery from and diagnosis of hardware errors.

Error checking hardware is configured and information about detected error conditions is conveyed via an architecturally-defined set of registers. The system programming interface of MCA is described below in Section 9.3 “Machine Check Architecture MSRs” on page 291.

9.1 Introduction

All computer systems are susceptible to errors—results that are contrary to the system design. Errors can be categorized as *soft* or *hard*. Soft errors are caused by transient interference and are not necessarily indicative of any damage to the computer circuitry. These external events include noise from electromagnetic radiation and the incursion of sub-atomic particles that cause bit cell storage capacitors to change state.

Hard errors are repeatable malfunctions that are generally attributable to physical damage to computer circuitry. Damage may be caused by external forces (for example, voltage surges) or wear processes inherent in the circuit technology. Damaged circuit elements can manifest symptoms similar to those that are caused by soft error processes. An increase in the frequency of errors attributable to one circuit element may indicate that the element has sustained damage or is wearing-out and may, in the future, cause a hard error.

9.1.1 Reliability, Availability, and Serviceability

This section describes the concepts of reliability, availability, and serviceability (RAS) and shows how they are interrelated.

The rate at which errors occur in a computer system is a measure of the system’s *reliability*. *Availability* is the percentage of time that the system is available to do useful work. Errors that prevent a computer system from continued operation result in *down-time*, that is, periods of unavailability. Down-time includes the amount of time required to restore the system to operation. This may include the time to diagnose a failure, determine the field replaceable unit (FRU) containing the faulty circuitry, carry out the repair action required to replace the identified FRU, and restart the system. This time directly impacts the system’s availability and is a measure of the system’s *serviceability*.

The availability of a computer system can be increased without decreasing performance or significantly increasing cost through the judicious addition of data and control path redundancy in concert with dedicated error-checking hardware. Together, redundancy and error checking *detect* and

often *correct* hardware errors. When errors are corrected by hardware, system operation continues without any perceptible disruption or loss in performance.

Another important technique that can prevent down-time is *error containment*. Error containment limits the propagation of an erroneous data. This enhances system availability by limiting the effects of errors to a subset of software or hardware resources. System software may either correct the error and resume the interrupted program or, if the error cannot be corrected, terminate software processes that cannot continue due to the error.

Error logging enhances serviceability by providing information that is used to identify the FRU that contains the failed circuitry. The mechanical design of the computer system can enhance serviceability (and thus availability) by making the task of physically replacing a failed FRU quicker and easier.

9.1.2 Error Detection, Logging, and Reporting

Error detection requires specific error-checking hardware that compares the actual result of some data transfer or transformation to the expected result. Any disparity indicates that an error has occurred. Error detection is controlled through implementation-specific means. Disabling detection is normally only appropriate when hardware is being debugged in the laboratory.

When an error is detected, hardware autonomously acts to either correct the error or contain the propagation of the corrupting effects of an uncorrected error. For some error sources, hardware action can be disabled by software through the MCA interface.

As hardware acts to correct or contain a detected error, it gathers information about the error to aid in recovery, diagnosis, and repair. The architecture provides software control of error *logging* and *reporting*. The following describes the characteristics of each:

- Logging

  Logging involves saving information about the error in specific MCA registers. If the error reporting bank associated with the error source is enabled, logging occurs; if disabled, error information is generally discarded (there are implementation-specific exceptions).

- Reporting

  An uncorrected error may be reported to system software via a machine-check exception, if error reporting for the specific error source is enabled.

Reporting is the hardware-initiated action of interrupting the processor using a machine-check exception (#MC). Reporting for each specific error type can be enabled or disabled by system software though the MCA register interface. Even if reporting for an error type is disabled, logging may continue.

Disabling reporting can negatively impact both error containment and error recovery (see the next section) and should be avoided.

Hardware categorizes errors into three classes. These are:

- *corrected*

- *uncorrected*
- *deferred*

The following sections describe the characteristics of each of these error classes:

If an error can be corrected by hardware, no immediate action by software is required. In this case, information is logged, if enabled, to aid in later diagnosis and possible repair.

If correction is not possible, the error is classified as uncorrected. The occurrence of an uncorrected error requires immediate action by system software to either correct the error and resume the interrupted program or, if software-based correction is not possible, to determine the extent of the impact of the uncorrected error to any executing instruction stream or the architectural state of the processor or system and take actions to contain the error condition by terminating corrupted software processes.

For errors that are not corrected, but have no immediate impact on the architectural state of the system, processor core, or any current thread of execution, the error may be classified by hardware as a deferred error. Information about deferred errors is logged, if enabled, but not reported via a machine-check exception. Instead hardware monitors the error and escalates the error classification to uncorrected at the point in time where the error condition is about to impact the execution of an instruction stream or cause the corruption of the processor core or system architectural state.

This escalation results in a #MC exception, assuming that reporting for that error source is enabled. If software can correct the error, it may be possible to resume the affected program. If not, software can terminate the affected program rather than bringing down the entire system. This is referred to as *error localization*.

A common example of deferred error processing and localization is the conversion of globally uncorrected DRAM errors to process-specific consumed memory errors. In this example, uncorrected ECC-protected data that has not yet been consumed by any processor core is tagged as “poison.” Hardware reports the uncorrected data as a localized error via a #MC exception when it is about to be used (“consumed”) by an instruction execution stream.

In contrast, an error that cannot be contained and is of such severity that it has compromised the continued operation of a processor core requires immediate action to terminate system processing and may result in a hardware-enforced *shutdown*. In the shutdown state, the execution of instructions by that processor core is halted. See Section 8.2.9 “#DF—Double-Fault Exception (Vector 8)” on page 242 for a description of the shutdown processor state.

If supported, system software can chose to configure and enable hardware to generate an interrupt when a deferred error is first detected. Corrected errors may be counted as they are logged. If supported and enabled, exceeding a software-configured count threshold may be signalled via an interrupt. These notification mechanisms are independent of machine-check reporting.

Specific details on hardware error detection, logging, and reporting are implementation-dependent and are described in the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.

9.1.3 Error Recovery

When errors cannot be corrected by hardware, *error recovery* comes into play. Error recovery, as defined by MCA, always involves software intervention. Logged information about the uncorrected error condition that caused the exception allows system software to take actions to either correct the error and resume the interrupted execution stream or terminate software processes (or higher-level software constructs) that are known to be affected by the uncorrected error.

From a system perspective, all errors are either recoverable or unrecoverable. The following outlines the characteristics of each:

- *Recoverable*—Hardware has determined that the architectural state of the processor experiencing the uncorrected error has not been compromised. Software execution can continue if system software can determine the extent of the error and take actions to either:
  - *correct* the error and *resume* the interrupted stream of execution or,
  - if this is not possible, *terminate* software processes that have incurred a loss of architectural state and *continue* other software processes that are unaffected by the error.
- *Unrecoverable*—Hardware has determined that the architectural state of the processor experiencing the uncorrected error has been corrupted. Software execution cannot reliably continue.

  Software saves any diagnostic information that it may be able to gather and halts.

The fact that an error is recoverable does not mean that recovery software will be able to resume program execution. If it is unable to determine the extent of the corruption or if it determines that essential state information has been lost, it may only be able to save information about the error and halt processing.

System software has many options to recover from an uncorrected error. The following is a partial list of possible actions that system software might take:

- If it can be determined that the corruption caused by the uncorrected error is contained within a software process, software can kill the process.
- If the uncorrected error has corrupted the architectural state of a virtual machine, the VMM can rebuild the container (using only hardware resources that are known to be good) and reboot the guest operating system.
- If the uncorrected error is a part of a block of data being transferred to or from an I/O device, the data transfer can be flushed and retried or terminated with an error.
- If the uncorrected error is due to a hard link failure, software can reconfigure the network to route information around the failed link.
- If the uncorrected error is in a cache and the cache line containing the uncorrected (known bad) data is in the shared state, software can invalidate the line so that it will be reloaded from memory or another cache that has the line in the owned state.

Many more error scenarios are recoverable depending on the effectiveness of hardware error containment, the logging capabilities of the system, and the sophistication of the recovery software that acts on the information conveyed through the MCA reporting structure.

If recovery software is unable to restore a valid system architectural state at some level of software abstraction (process, guest operating system, virtual machine, or virtual machine monitor), the uncorrected error is considered *system fatal*. In this situation, system software must halt the execution of instructions. A system reset is required to restore the system to a known-good architectural state.

9.2 Determining Machine-Check Architecture Support

Support for the machine-check architecture is implementation-dependent. System software executes the CPUID instruction to determine whether a processor implements the machine-check exception (#MC) and the global MCA MSRs. The CPUID Fn0000_0001_EDX[MCE] feature bit indicates support for the machine-check exception and the CPUID Fn0000_0001_EDX[MCA] feature bit indicates support for the base set of global machine-check MSRs.

Once system software determines that the base set of MCA MSRs is available, it determines the implemented number of machine-check reporting banks by reading the machine-check capabilities register (MCG_CAP), which is the first of the global MCA MSRs.

For a processor implementation to provide an architecturally compliant MCA interface, it must provide support for the machine-check exception, the global machine-check MSRs, the watchdog timer (see “CPU Watchdog Timer Register” on page 295.), and at least one bank of the machine-check reporting registers.

Support for the deferred reporting and software-based containment of uncorrected data errors is indicated by the feature bit CPUID Fn8000_0007_EBX[SUCCOR]. See “Machine-Check Recovery” on page 298.

Support for recoverable MCA overflow conditions is indicated by feature bit CPUID Fn8000_0007_EBX[McaOverflowRecov]. See the discussion of recoverable status overflow in Section 9.3.2.1 “MCA Overflow” on page 297.

Implementation-specific information concerning the machine-check mechanism can be found in the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product. For more information on using the CPUID instruction, see Section 3.3, “Processor Feature Identification,” on page 71.

9.3 Machine Check Architecture MSRs

The AMD64 Machine-Check Architecture defines the set of model-specific registers (MCA MSRs) used to log and report hardware errors. These registers are:

- Global status and control registers:
  - Machine-check global-capabilities register (MCG_CAP)

  - Machine-check global-status register (MCG_STATUS)
  - Machine-check global-control register (MCG_CTL)
  - Machine-check exception configuration register (MCA_INTR_CFG)
- One or more error-reporting register banks, each containing:
  - Machine-check control register (MC*i*_CTL)
  - Machine-check status register (MC*i*_STATUS)
  - Machine-check address register (MC*i*_ADDR)
  - At least one machine-check miscellaneous error-information register (MC*i*_MISC0)

  Each error-reporting register bank is associated with a specific processor unit (or group of processor units).
- CPU Watchdog Timer register (CPU_WATCHDOG_TIMER)

The error-reporting registers retain their values through a warm reset. (A warm reset occurs while power to the processor is stable. This in contrast to a cold reset, which occurs during the application of power after a period of power loss.) This preservation of error information allows the platform firmware or other system-boot software to recover and report information associated with the error when the processor is forced into a shutdown state.

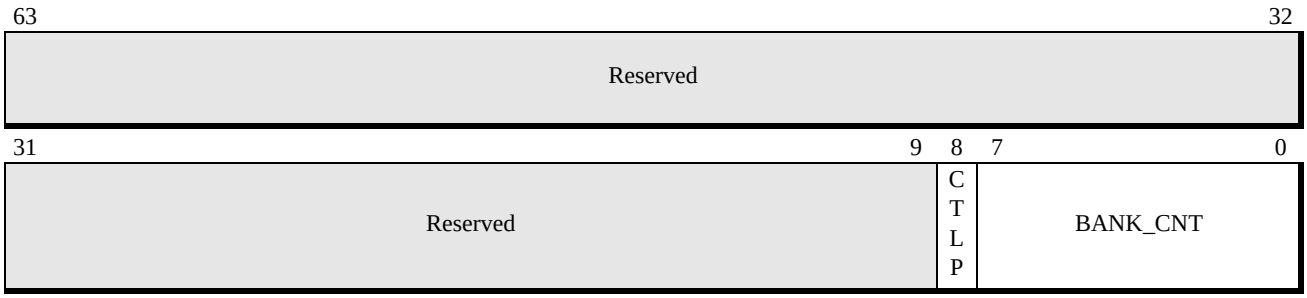
The RDMSR and WRMSR instructions are used to read and write the machine-check MSRs. See “Machine-Check MSRs” on page 671 for a listing of the machine-check MSR numbers and their reset values. The following sections describe each MCA MSR and its function.

9.3.1 Global Status and Control Registers

The global status and control MSRs are the MCG_CAP, MCG_STATUS, MCG_CTL and MCA_INTR_CFG registers.

9.3.1.1 Machine-Check Global-Capabilities Register

Figure 9-1 shows the format of the machine-check global-capabilities register (MCG_CAP). MCG_CAP is a read-only register that specifies the machine-check mechanism capabilities supported by the processor implementation.



Bits	Mnemonic	Description	Access type
63:9	—	Reserved	R
8	CTLP	MCG_CTL register present	R
7:0	BANK_CNT	Number of reporting banks	R

Figure 9-1. MCG_CAP Register

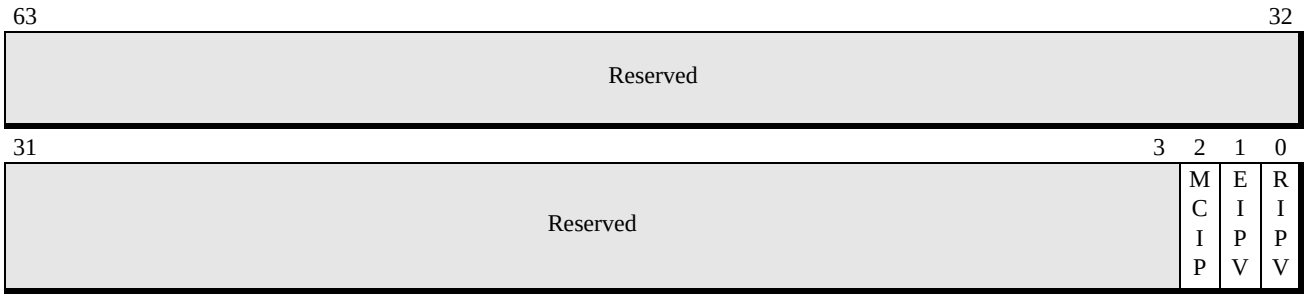
The fields within the MCG_CAP register are:

- *BANK_CNT (MCi Bank Count)*—Bits 7:0. This field specifies how many error-reporting register banks are supported by the processor implementation.
- *CTLP(MCG_CTL Register Present)*—Bit 8. This bit specifies whether or not the Machine-Check Global-Control (MCG_CTL) Register is supported by the processor. When the bit is set to 1, the register is supported. When the bit is cleared to 0, the register is unsupported. The MCG_CTL register is described on page 294.

All remaining bits in the MCG_CAP register are reserved. Writing values to the MCG_CAP register produces undefined results.

9.3.1.2 Machine-Check Global-Status Register

Figure 9-2 shows the format of the machine-check global-status register (MCG_STATUS). MCG_STATUS provides basic information about the processor state after the occurrence of a machine-check error.



Bits	Mnemonic	Description	R/W
63:3	—	Reserved	
2	MCIP	Machine Check In-Progress	R/W
1	EIPV	Error IP Valid Flag	R/W
0	RIPV	Restart IP Valid Flag	R/W

Figure 9-2. MCG_STATUS Register

The fields within the MCG_STATUS register are:

- *Restart-IP Valid (RIPV)*—Bit 0. When this bit is set to 1, the interrupted program can be reliably restarted at the instruction addressed by the instruction pointer pushed onto the stack by the machine-check error mechanism. If this bit is cleared to 0, the interrupted program cannot be reliably restarted.
- *Error-IP Valid (EIPV)*—Bit 1. When this bit is set to 1, the process thread that was interrupted by the #MC is responsible for the machine-check error, which occurred at the privilege level indicated by the CS selector that was pushed on the stack by the exception. If this bit is cleared to 0, it is possible that the interrupted thread is not responsible for the machine-check error.
- *Machine Check In-Progress (MCIP)*—Bit 2. When this bit is set to 1, it indicates that a machine-check error is in progress. If another machine-check error occurs while this bit is set, the processor enters the shutdown state. The processor sets this bit whenever a machine check exception is generated. Software is responsible for clearing it after the machine check exception is handled.

All remaining bits in the MCG_STATUS register are reserved.

9.3.1.3 Machine-Check Global-Control Register

Figure 9-3 shows the format of the machine-check global-control register (MCG_CTL). MCG_CTL is used by software to enable or disable the logging and reporting of machine-check errors from the implemented error-reporting banks. Depending on the implementation, detected errors from some error sources associated with a reporting bank that is disabled are still logged. Setting all bits to 1 in this register enables all implemented error-reporting register banks to log errors.

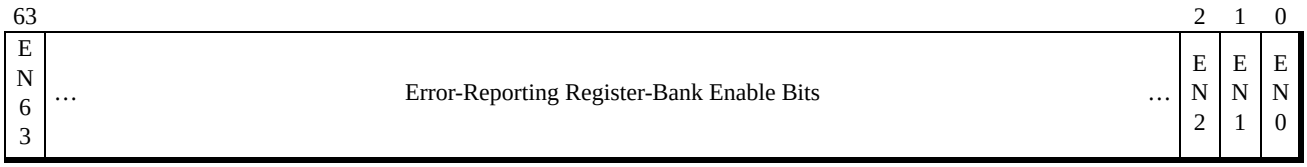
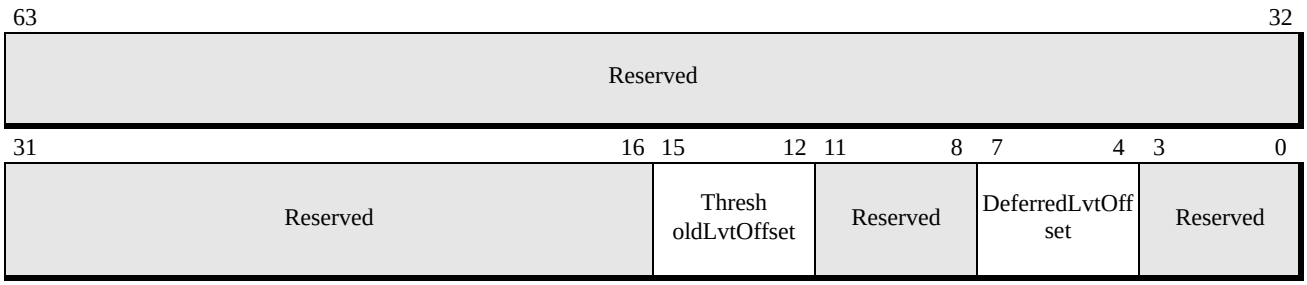


Figure 9-3. MCG_CTL Register

9.3.1.4 Machine-Check Exception Configuration Register

In addition to the standard Machine Check exception (#MC) signaling, certain machine check events can raise an interrupt via the APIC LVT mechanism. Configuration of these interrupts are provided, in part, in register MCA_INTR_CFG located at MSR C000_0410, see Figure 9-4. See Section 16.4 “Local Interrupts” on page 608 for details on configuring LVT entries.



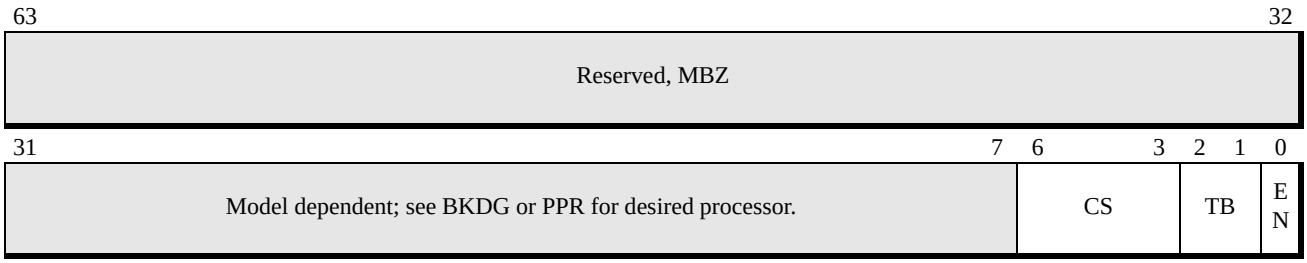
Bits	Mnemonic	Description	R/W
63:16	—	Reserved	R
15:12	ThresholdLvtOffset	For error thresholding interrupts, specifies the address of the LVT entry in the APIC registers as follows: LVT address = (ThresholdLvtOffset < < 4) + 500h (see APIC[530:500]).	R/W
11:8	—	Reserved	R
7:4	DeferredLvtOffset	For deferred error interrupts, specifies the address of the LVT entry in the APIC registers as follows: LVT address = (DeferredLvtOffset < < 4) + 500h (see APIC[530:500]).	R/W
3:0	—	Reserved	R

Figure 9-4. MCA_INTR_CFG Register

9.3.1.5 CPU Watchdog Timer Register

The CPU watchdog timer is used to generate a machine check condition when an instruction does not complete within a time period specified by the CPU Watchdog Timer register. The timer restarts the count each time an instruction completes, when enabled by the *CPU Watchdog Timer Enable* bit. The time period is determined by the *Count Select* and *Time Base* fields. The timer does not count during halt or stop-grant.

The format of the CPU watchdog timer is shown in Figure 9-5.



Bits	Mnemonic	Description	R/W
63:7	Reserved	Reserved, Must be Zero	
6:3	CS	CPU Watchdog Timer Count Select	R/W
2:1	TB	CPU Watchdog Timer Time Base	R/W
0	EN	CPU Watchdog Timer Enable	R/W

Figure 9-5. CPU Watchdog Timer Register Format

CPU Watchdog Timer Enable (EN) - Bit 0. This bit specifies whether the CPU Watchdog Timer is enabled. When the bit is set to 1, the timer increments and generates a machine check when the timer expires. When cleared to 0, the timer does not increment and no machine check is generated.

CPU Watchdog Timer Time Base (TB) - Bits 2:1. Specifies the time base for the time-out period indicated in the *Count Select* field. The allowable time base values are provided in Table 9-1.

Table 9-1. CPU Watchdog Timer Time Base

TB[1:0]	Time Base
00b	1 millisecond
01b	1 microsecond
10b	Reserved
11b	Reserved

CPU Watchdog Timer Count Select (CS) - Bits 6:3. Specifies the time period required for the CPU Watchdog Timer to expire. The time period is this value times the time base specified in the *Time Base* field. The allowable values are shown in Table 9-2.

Table 9-2. CPU Watchdog Timer Count Select

CS[3:0]	Value
0000b	4095
0001b	2047
0010b	1023
0011b	511
0100b	255

Table 9-2. CPU Watchdog Timer Count Select (continued)

0101b	127
0110b	63
0111b	31
1000b	8191
1001b	16383
1010b–1111b	Reserved

9.3.2 Error-Reporting Register Banks

Each error-reporting register bank contains the following registers:

- Machine-check control register (MC*i*_CTL).
- Machine-check status register (MC*i*_STATUS).
- Machine-check address register (MC*i*_ADDR).
- Machine-check miscellaneous error-information register 0 (MC*i*_MISC0).

The *i* in each register name corresponds to the number of a supported register bank. Each error-reporting register bank is normally associated with a specific execution unit. The number of error-reporting register banks is implementation-specific. For more information, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.

Software reads the MCG_CAP register to determine the number of supported register banks. The first error-reporting register (MC0_CTL) always starts with MSR address 400h, followed by MC0_STATUS (401h), MC0_ADDR (402h), and MC0_MISC0 (403h). The addresses of any additional error-reporting MSRs are assigned sequentially starting at 404h through the remaining supported register banks.

9.3.2.1 MCA Overflow

If an error occurs within an error reporting bank while the status register for that bank contains valid data (MC*i*_STATUS[VAL] = 1), an MCA overflow condition results. In this situation, information about the new error will either be discarded or will replace the information about the prior error.

Hardware sets the MC*i*_STATUS[OVER] bit to indicate this condition has occurred and follows a set of rules to determine whether to overwrite the previously logged error information or discard the new error information. These rules are shown in Table 9-3 below.

Table 9-3. Error Logging Priorities

		Previous Error Type		
		Corrected	Deferred	Uncorrected
Current Error Type	**Corrected**	Discard Current	Discard Current	Discard Current
	Deferred	Overwrite Previous	Discard Current	Discard Current
	Uncorrected	Overwrite Previous	Overwrite Previous	Discard Current

Note(s):

1. *Logging a deferred error has priority over the retention of information concerning a prior corrected error.*
2. *Logging an uncorrected error has priority over the retention of information concerning either a prior deferred or corrected error.*
3. *Valid Information concerning an uncorrected error is not overwritten by any subsequent errors.*

If the VAL bit is not set, hardware writes the appropriate logging registers based on the type of error (writing the MC*i*_STATUS register last) and then sets the VAL bit to indicate to software that the information currently contained in the MC*i*_STATUS register is valid. Software clears the VAL bit after reading the contents of this register (after reading and saving valid information stored in any of the other logging registers) to indicate to hardware that it has saved the information, making the registers available to log the next error.

If survivable MCA overflow is supported by the implementation (as indicated by CPUID Fn8000_0007_EBX[McaOverflowRecov] = 1), the state of the MC*i*_STATUS[PCC] bit indicates whether system execution can continue. If a particular processor does not support survivable MCA overflow and overflow occurs, software must halt instruction execution on that processor core regardless of the state of the PCC bit because critical information may have been lost as a result of the overflow. See the description of the Machine-Check Status registers below for more information on the PCC bit.

9.3.2.2 Machine-Check Recovery

Machine Check Recovery is a feature allowing recovery of the system when the hardware cannot correct an error. Machine Check Recovery is supported when CPUID Fn8000_0007_EBX[SUCCOR]=1.

When Machine Check Recovery is supported and an uncorrected error has been detected that the hardware can contain to the task or process to which the machine check has been delivered, it logs a context-synchronous uncorrectable error (MCi_STATUS[UC]=1, MCi_STATUS[PCC]=0). The rest of the system is unaffected and may continue running if supervisory software can terminate only the affected process context.

9.3.2.3 Machine-Check Control Registers

The machine-check control registers (MC*i*_CTL), as shown in Figure 9-6, contain an enable bit for each error source within an error-reporting register bank. Setting an enable bit to 1 enables error reporting for the specific feature controlled by the bit, and clearing the bit to 0 disables error reporting

for the feature. It is recommended that the value FFFF_FFFF_FFFF_FFFFh be programmed into each MC*i*_CTL register.

Disabling the reporting of errors from error sources that are capable of detecting uncorrected errors can compromise future error recovery and is not recommended. Other implementation-specific values are documented in the product's *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual*.

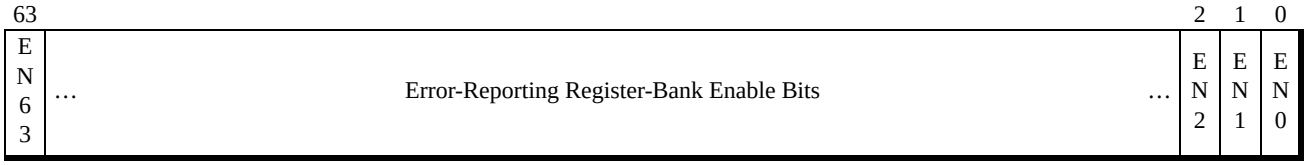
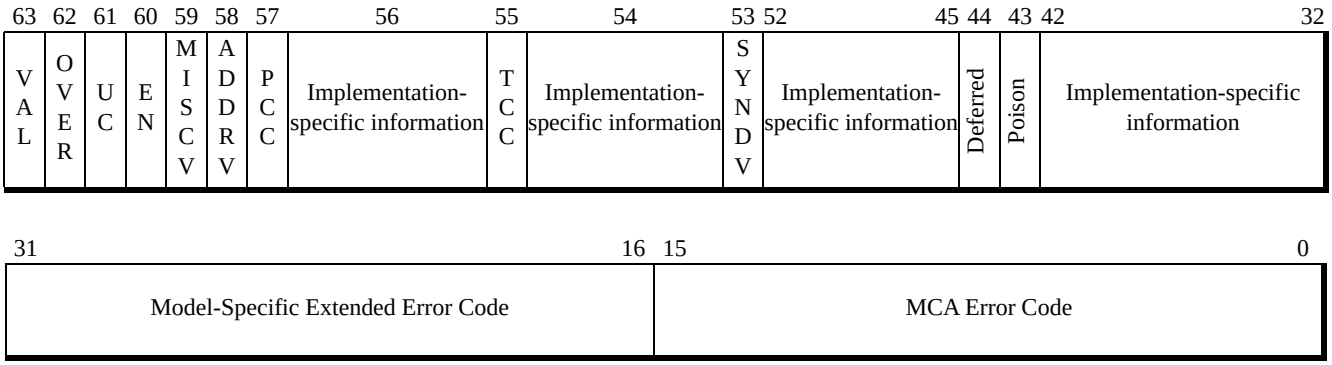


Figure 9-6. MC*i*_CTL Register

9.3.2.4 Machine-Check Status Registers

Each error-reporting register bank includes a machine-check status register (MC*i*_STATUS) that the processor uses to log error information. Hardware writes the status register bits when an error is detected, and sets the VAL bit of the register to 1, indicating that the status information is valid. Error reporting for the error source associated with the detected error *does not* need to be enabled in the MC*i*_CTL Register for the processor to write the status register. Error reporting must be enabled for the error to be reported via a #MC exception. Software is responsible for clearing the status register after the exception has been handled. Attempting to write a value other than 0 to an MC*i*_STATUS register in an implemented MCA bank may raise a general-protection (#GP) exception.

Figure 9-7 on page 300 shows the format of the MC*i*_STATUS register.



Bits	Mnemonic	Description	R/W
63	VAL	Valid	R/W*
62	OVER	Status Register Overflow	R/W*
61	UC	Uncorrected Error	R/W*
60	EN	Error Condition Enabled	R/W*
59	MISCV	Miscellaneous-Error Register Valid	R/W*
*System software can only clear this bit to 0.			
58	ADDRV	Error-Address Register Valid	R/W*
57	PCC	Processor-Context Corrupt	R/W*
56		Implementation-specific information	R/W*
55	TCC	Task-Context Corrupt	R/W*
54		Implementation-specific information	R/W*
53	SYNDV	Syndrome Register Valid	R/W*
52:45		Implementation-specific information	R/W*
44	Deferred	Deferred error	R/W*
43	Poison	Poisoned data consumed	R/W*
42:32		Implementation-specific information	R/W*
31:16		Model-Specific Extended Error Code	R/W*
15:0		MCA Error Code	R/W*

Figure 9-7. MC*i*_STATUS Register

The fields within the MC*i*_STATUS register are:

- *MCA Error Code*—Bits 15:0. This field encodes information about the error, including:
  - The type of transaction that caused the error.
  - The memory-hierarchy level involved in the error.
  - The type of request that caused the error.
  - Other information concerning the transaction type.

  See the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product for information on the format and encoding of the MCA error code.

- *Model-Specific Extended Error Code*—Bits 31:16. This field encodes model-specific information about the error. For further information, see the documentation for particular implementations of the architecture.
- *Implementation-specific Information*—Bits 56, 54:45, 42:32. These bit ranges hold model-specific error information. Software should not rely on the field definitions in these ranges being consistent between processor implementations. For details see the BKDG or PPR for desired implementations of the architecture.
- *Poison*—Bit 43. When set to 1, this bit indicates that the uncorrected error condition being reported is due to the attempted use of data that was previous detected as in error (and could not be corrected) and marked as known-bad.
- *Deferred*—Bit 44. When set to 1, this bit indicates that hardware has determined that the error condition being logged has not affected the execution of any instruction stream and that action by system software to prevent or correct an error is not required. No machine-check exception is signalled. Hardware will monitor the error and log an uncorrected error when the execution of any thread of execution is impacted.
- *SYNDV*—Bit 53. When set to 1, this bit indicates that the contents of the corresponding syndrome register (MCA_SYND) are valid. When this bit is cleared, the contents of MCA_SYND are not valid.
- *TCC*—Bit 55. When set to 1, this bit indicates that the hardware context of the process thread to which the error was reported may have been corrupted. Continued operation of the thread may have unpredictable results. When this bit is cleared, the hardware context of the process thread to which the error was reported is not corrupted and recovery of the process thread is possible. This bit is only meaningful when MCA_STATUS[PCC]=0.
- *PCC*—Bit 57. When set to 1, this bit indicates that the processor state is likely to be corrupt due to an uncorrected error. In this case, it is possible that software cannot reliably continue execution. When this bit is cleared, the processor state is not corrupted and recovery is still possible. If the PCC bit is set in any error bank, the processor will clear RIPV and EIPV in the MCG_STATUS register.
- *ADDRV*—Bit 58. When set to 1, this bit indicates that the contents of the corresponding error-reporting address register (MC*i*_ADDR) are valid. When this bit is cleared, the contents of MC*i*_ADDR are not valid.
- *MISCV*—Bit 59. When set to 1, this bit indicates that additional information about the error is saved in the corresponding error-reporting miscellaneous register (MC*i*_MISC0). When cleared, this bit indicates that the contents of the MC*i*_MISC0 register are not valid.
- *EN*—Bit 60. When set to 1, this bit indicates that the error condition is enabled in the corresponding error-reporting control register (MC*i*_CTL). Errors disabled by MC*i*_CTL do not cause a machine-check exception.
- *UC*—Bit 61. When set to 1, this bit indicates that the logged error status is for an uncorrected error. When cleared, the error class is determined by looking at the Deferred bit; the error is a Corrected error if the Deferred bit is clear or a Deferred error if the Deferred bit is set. (See Section 9.1.2, “Error Detection, Logging, and Reporting,” on page 288, for more detail on these error classes.)

- *OVER*—Bit 62. This bit is set to 1 by the processor if the VAL bit is already set to 1 as the processor attempts to write error information into MC*i*_STATUS. In this situation, the machine-check mechanism handles the contents of MC*i*_STATUS as follows:
  - For processor implementations that log errors for disabled reporting banks, status for an enabled error replaces status for a disabled error.
  - Status for a deferred error replaces status for a corrected error.
  - Status for an uncorrected error replaces status for a corrected or deferred error.
  - Status for an enabled uncorrected error is never replaced.

  See Section 9.3.2.1 “MCA Overflow” on page 297 for more information on this field.
- *VAL*—Bit 63. This bit is set to 1 by the processor if the contents of MC*i*_STATUS are valid. Software should clear the VAL bit after reading the MC*i*_STATUS register, otherwise a subsequent machine-check error sets the OVER bit as described above.

When a machine-check error occurs, the processor writes an error code into the appropriate MC*i*_STATUS register MCA error-code field. The MC*i*_STATUS[VAL] bit is set to 1, indicating that the MC*i*_STATUS register contents are valid.

MCA error-codes are used to report errors in the memory hierarchy, the system bus, and the system-interconnection logic. Error-codes are divided into subfields that are used to describe the cause of an error. The information is implementation-specific. For further information, see the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.

9.3.2.5 Machine-Check Address Registers

Each error-reporting register bank includes a machine-check address register (MC*i*_ADDR) that the processor uses to report the address or location associated with the logged error. The address field can hold a virtual (linear) address, a physical address, or a value indicating an internal physical location, depending on the type of error. For further information, see the documentation for particular implementations of the architecture. The contents of this register are valid only if the ADDRV bit in the corresponding MC*i*_STATUS register is set to 1.

9.3.2.6 Machine-Check Miscellaneous-Error Information Register 0 (MC*i*_MISC0)

Each error-reporting register bank includes the Machine-Check Miscellaneous 0 register that the processor uses to report additional error information.

In some implementations, the MC*i*_MISC0 register is used for error thresholding. Thresholding is a mechanism provided by hardware to:

- count detected errors, and
- (optionally) generate an APIC-based interrupt when a programmed number of errors has been counted.

Processor hardware counts detected errors and ensures that multiple error sources do not share the same thresholding register. Software can use corrected error counts to help predict which components might soon fail (begin generating uncorrectable errors) and schedule their replacement.

Threshold counters increment for error sources that are enabled for logging.

9.3.2.7 Additional Machine-Check Miscellaneous-Error Information Registers (MC*i*_MISC*j*)

The MC*i*_MISC0[BLKP] field is used to point to any additional MC*i*_MISC*j* registers, where j > 0. If this field is zero, no additional MC*i*_MISC registers are implemented. If this field is one, and CPUID Fn8000_0007_EBX[ScalableMca]=1, refer to Section 9.3.3, “Machine-Check Architecture Extension Registers,” on page 305 for addresses of additional MC*i*_MISC*j* registers.

If the MC*i*_MISC0[BLKP] field is non-zero and CPUID Fn8000_0007_EBX[ScalableMca]=0, up to 8 additional MC*i*_MISC*j* registers can be implemented for the error-reporting bank *i* (for a total of 9). These registers are allocated in contiguous blocks of 8, with MC*i*_MISC1 addressed by:

MC*i*_MISC1 address = C000_0400h + (MC*i*_MISC0[BLKP] << 3)

This is illustrated in Figure 9-8 below.

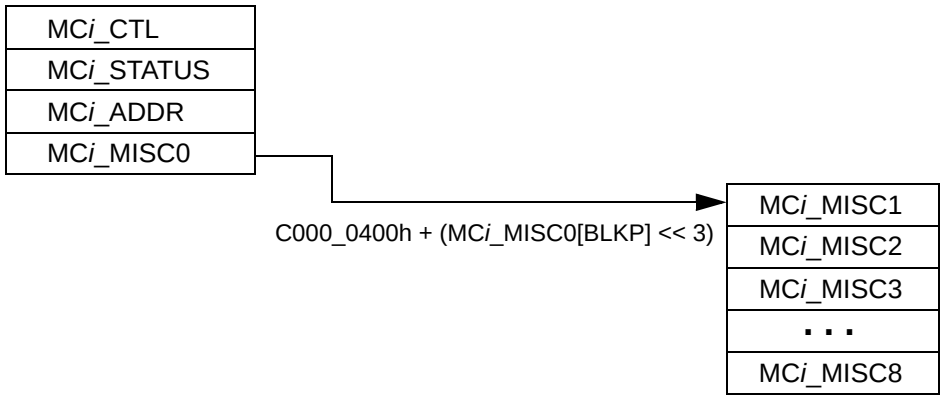
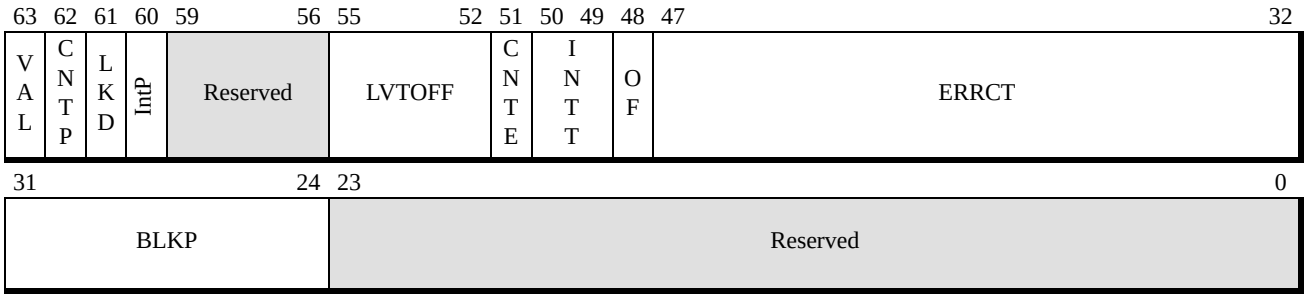


Figure 9-8. MC*i*_MISC1 Addressing

The format of implemented MC*i*_MISC*j* registers depends upon their use and use can vary from one implementation to another. Figure 9-9 below illustrates the format of a miscellaneous error information register when used as an error thresholding register.

All miscellaneous error information registers will contain the VAL field in bit position 63. MC*i*_MISC0 must contain the BLKP field in bits 31:24.



Bits	Mnemonic	Description	R/W	Reset
63	VAL	Valid	R	1b
62	CNTP	Counter Present	R	1b
61	LKD	Locked	R/W	0b
60	IntP	Thresholding Interrupt Supported	R	Implementation Dependent
59:56	Reserved			
55:52	LVTOFF	LVT Offset	R/W	0000b
51	CNTE	Counter Enable	R/W	0b
50:49	INTT	Interrupt Type	R/W	00b
48	OF	Overflow	R/W	Undefined
47:32	ERRCT	Error Counter	R/W	Undefined
31:24	BLKP	Block pointer for additional MISC registers	R	Undefined
23:0	Reserved			

Figure 9-9. Miscellaneous Information Register (Thresholding Register Format)

The fields within the MC*i*_MISC*j* register are:

- *Valid (VAL)*—Bit 63. When set to 1, indicates that the counter present (CNTP) and block pointer (BLKP) fields in this register are valid.
- *Counter Present (CNTP)*—Bit 62. When set to 1, indicates the presence of a threshold counter.
- *Locked (LKD)*—Bit 61. When set to 1, indicates that the threshold counter is not available for OS use. If this is the case, writes to bits 60:0 of this register are ignored and do not generate a fault. Software must check the Locked bit before writing into the thresholding register.

  This field is write-enabled by MSR C001_0015h Hardware Configuration Register [MCSTATUSWrEn].
- *IntP (Thresholding Interrupt Supported)*—Bit 60. When set, this bit indicates that the reporting of threshold overflow via interrupt is supported. Interrupt type is determined by the setting of the INTT field.
- *LVT Offset (LVTOFF)*—Bits 55:52. This field specifies the address of the APIC LVT entry to deliver the threshold counter interrupt. Software must initialize the APIC LVT entry before enabling the threshold counter to generate the APIC interrupt; otherwise, undefined behavior may result. This field is aliased to MCA_INTR_CFG[ThresholdLvtOffest].

- *Counter Enable (CNTE)*—Bit 51. When set to 1, counting of implementation-dependent errors is enabled; otherwise, counting is disabled.
- *Interrupt Type (INTT)*—Bits 50:49. The value of this field specifies the type of interrupt signaled when the value of the overflow bit changes from 0 to 1.
  - 00b = No interrupt
  - 01b = APIC-based interrupt
  - 10b = Reserved
  - 11b = Reserved
- *Overflow (OF)*—Bit 48. The value of this field is maintained through a warm reset. This bit is set by hardware when the error counter increments to its maximum implementation-supported value (from FFFEh to FFFFh for the maximum implementation-supported value). This is defined as the threshold level. When the overflow bit is set, the interrupt selected by the interrupt type field is generated. Software must reset this bit to zero in the interrupt handler routine when they update the error counter.
- *Error Counter (ERRCT)*—Bits 47:32. This field is maintained through a warm reset. The size of the threshold counter is implementation-dependent. Implementations with less than 16 bits fill the most significant unimplemented bits with zeros.

  Software enumerates the counter bits to discover the size of the counter and the threshold level (when counter increments to the maximum count implemented). Software sets the starting error count as follows:

  Starting error count = threshold level – desired software error count to cause overflow

  The error counter is incremented by hardware when errors for the associated error counter are logged. When this counter overflows, it stays at the maximum error count (with no rollover).
- *Block pointer for additional MISC registers (BLKP)*—Bits 31:24. This field is only valid when valid (VAL) bit is set. When non-zero, this field is used to indicate the presence of additional MCi_MISC registers.

Other formats for miscellaneous information registers are implementation-dependent, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product for more details.

9.3.3 Machine-Check Architecture Extension Registers

This section describes the Machine Check Architecture Extension (MCAX). MCAX is available on a processor when CPUID Fn8000_0007_EBX[ScalableMca] returns 1.

A processor that supports MCAX supports up to 64 MCA banks per thread, with 16 MCAX registers per bank, at MSRs C000_2[3FF:000]. MSRs C000_2[FFF:400] are reserved for future use.

Software can identify an MCA bank as MCAX-capable by checking for MCA_CONFIG[Mcax]=1. An MCAX bank contains the following MCAX registers: MCA_CTL, MCA_STATUS, MCA_ADDR, MCA_MISC0, MCA_CONFIG, MCA_IPID, MCA_SYND, MCA_DESTAT,

MCA_MISC[4:1], and MCA_SYND[2:1]. See Figure 9-10 for address mapping of registers in a MCAX bank.

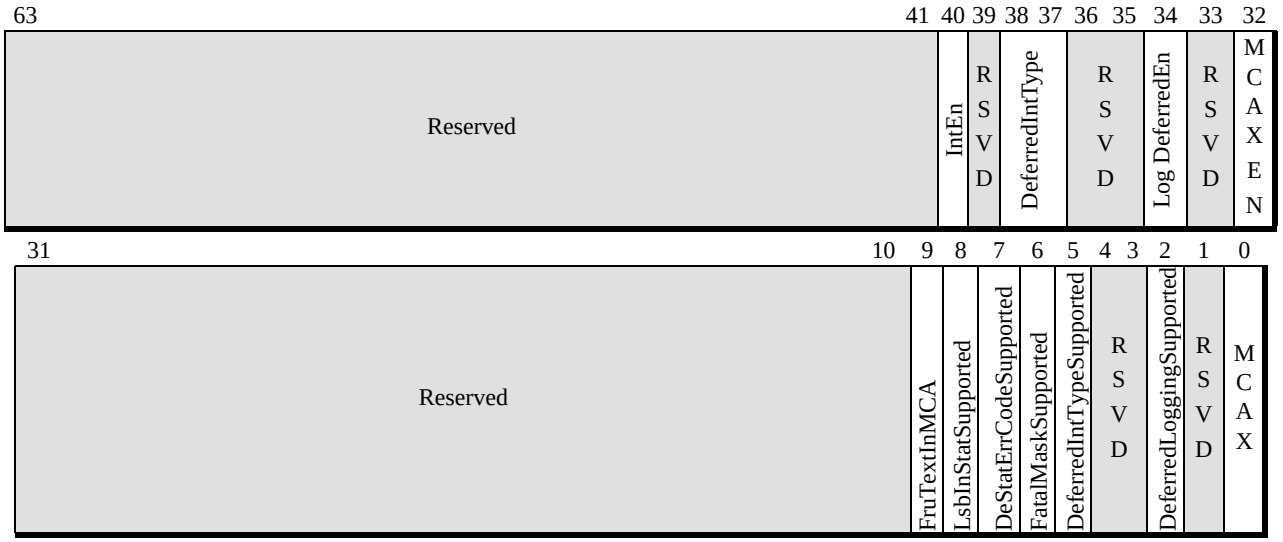
MCAX bank registers MCA_CTL, MCA_STATUS, MCA_ADDR, and MCA_MISC0 are aliased to MCA bank registers MCi_CTL, MCi_STATUS, MCi_ADDR, and MCi_MISC0 in MCA banks 0 to 31. The format of MCA_CTL, MCA_STATUS, MCA_ADDR, and MCA_MISC0 is the same as the format of the corresponding MCA bank register.

MCA bank number (decimal)	MCAX Bank registers (MSR C000_2xxx)				MCAX Bank registers (MSR C000_2xxx)							
	MCA_CTL	MCA_STATUS	MCA_ADDR	MCA_MISC0	MCA_CONFIG	MCA_IPID	MCA_SYND	Reserved	MCA_DESTAT	MCA_DEADDR	MCA_MISC [1:4]	MCA_SYND [1:2]
0	000h	001h	002h	003h	004h	005h	006h	007h	008h	009h	00Ah:00Dh	00Eh:00Fh
n	000h + n*10h	001h + n*10h	002h + n*10h	003h + n*10h	004h + n*10h	005h + n*10h	006h + n*10h	007h + n*10h	008h + n*10h	009h + n*10h	(00Ah:00Dh) + n*10h	(00Eh:00Fh) + n*10h

Figure 9-10. Address Mapping of Registers in MCAX Bank

9.3.3.1 MCA Configuration Register

The MCA_CONFIG register holds configuration information for the MCA bank.



Bits	Mnemonic	Description	R/W
63:41	Reserved		
40	IntEn	Interrupt Enable	RW
39	Reserved		
38:37	DeferredIntType	Deferred Error Interrupt Type	RW
36:35	Reserved		
34	LogDeferredEn	Deferred Error Logging Enable	RW
33	Reserved		
32	McaxEn	MCAX Enable	RW
31:10	Reserved		
9	FruTextInMCA	MCA FruText Supported	RW
8	LsbInStatSupported	Address LSB in MCA_STATUS Supported	R
7	DeStatErrCodeSupported	Deferred Error Status Error Code Supported	R
6	FatalMaskSupported	System fatal error event supported.	R
5	DeferredIntTypeSupported	Deferred Interrupt Type Supported	R
4:3	Reserved		
2	DeferredLoggingSupported	Deferred Error Logging Supported	R
1	Reserved		
0	MCAX	MCAX Capable	R

Figure 9-11. MCA_CONFIG Register

IntEn—Bits 40. When this bit is set to 0, an interrupt will not occur for corrected errors. When this bit is set to 1 and IntPresent=1, this bank will generate an interrupt on a corrected error to the interrupt vector configured in MCA_INTR_CFG[ThresholdLvtOffset].

DeferredIntType—Bits 38:37. When MCA_CONFIG[McaxEn]=1, specifies the type of interrupt to generate on a deferred error.

00b	No interrupt
01b	APIC based interrupt
10b	SMI
11b	Reserved

LogDeferredEn—Bit 34. Enable logging of deferred errors in MCA_STATUS. 0=Log deferred errors only in MCA_DESTAT and MCA_DEADDR. 1=Log deferred errors in MCA_STATUS and MCA_ADDR in addition to MCA_DESTAT and MCA_DEADDR. This bit does not affect logging of deferred errors in MCA_SYND or MCA_MISCx.

McaxEn—Bit 32. Enable MCAX feature set. 1=System software acknowledges support for the MCAX feature set. 0=System software has not acknowledged support for the MCAX feature set; all uncorrected errors and fatal errors cause a system fatal event.

FruTextInMca—Bit 9. 1=MCA_SYND1 and MCA_SYND2 contain an ASCII-formatted Field Replaceable Unit (FRU) identifier for the error logged in MCA_STATUS or MCA_DESTAT. 0=MCA_SYND1 and MCA_SYND2 contain implementation-specific information.

LsbInStatSupported—Bit 8. 1=MCA_STATUS[29:24] and MCA_DESTAT[29:24] contain the least significant valid bit of the address logged in MCA_ADDR and MCA_DEADDR.

DeStatErrCodeSupported—Bit 7. 1=MCA_DESTAT contains an MCA Error Code and a Model-specific Extended Error Code for the error logged in MCA_DESTAT.

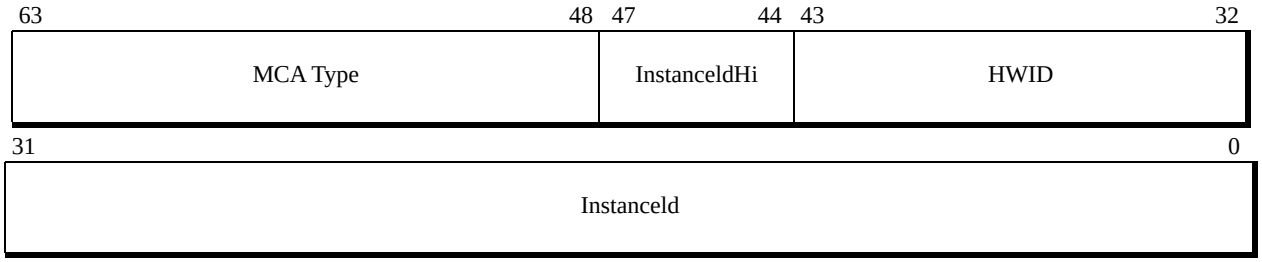
DeferredIntTypeSupported—Bit 5. 1=MCA_CONFIG[DeferredIntType] controls the type of interrupt generated on a deferred error. Deferred errors are supported in this bank only if MCA_CONFIG[DeferredLoggingSupported]=1.

DeferredLoggingSupported—Bit 2. 1= MCA_CONFIG[LogDeferredEn] in this bank controls the logging behavior for deferred errors. 0= MCA_CONFIG[LogDeferredEn] is not supported.

MCAX—Bit 0. MCAX: Set by hardware. This MCA bank provides Machine Check Architecture Extensions.

9.3.3.2 MCA IP Identification

This register holds information which identifies the specific bank type: the McaType value, HWID, and the InstanceID. By providing those values as part of the MCA, the bank type information is available to the machine check handler for parsing and for diagnosis.



Bits	Mnemonic	Description	R/W
63:48	McaType	MCA bank type.	RW
47:44	InstanceIdHi	The high bits of the Instance ID.	RW
43:32	Hwid	Hardware ID value.	RW
31:0	InstanceID	The low bits of the Instance ID.	RW

Figure 9-12. MCA_IPID Register

McaType—Bits 63:48. MCA bank type. This field is used to identify the MCA bank type in conjunction with MCA_IPID[Hwid]. For more information, see the Processor Programming Reference Manual applicable to your product.

InstanceIdHi—Bits 47:44. The high bits of the Instance identification for the MCA bank. This field is used to identify the MCA bank instance in conjunction with MCA_IPID[InstanceId]. For more information, see the Processor Programming Reference Manual applicable to your product.

Hwid—Bits 43:32. MCA bank hardware identification. For more information, see the Processor Programming Reference Manual applicable to your product.

InstanceId—Bits 31:0. The Instance identification for the MCA bank. This field is used to identify the MCA bank instance.

9.3.3.3 MCA Syndrome Register

The MCA_SYND register stores a syndrome associated with the error logged in MCA_STATUS or MCA_DESTAT. The “syndrome” may include syndrome values associated with an error correcting code or other information about the error. The contents of this register are valid if MCA_STATUS[SYNDV] bit is set to 1 or MCA_DESTAT[SYNDV] bit is set to 1.

The format and contents of this register are implementation dependent. For more information, see the Processor Programming Reference Manual applicable to your product.

9.3.3.4 MCA Deferred Error Status Register

This register holds status information for deferred errors. This register ensures that software will see a deferred error, even when a later error of higher severity occurs.

If the error being logged is a deferred error, then the error will be logged to MCA_DESTAT. See Section 9.3.2.4, “Machine-Check Status Registers,” on page 299 for more detailed descriptions of the register fields.

When the deferred error has been processed by the deferred error handler, MCA_DESTAT should be cleared. If MCA_STATUS also contains a deferred error, MCA_STATUS should be cleared.

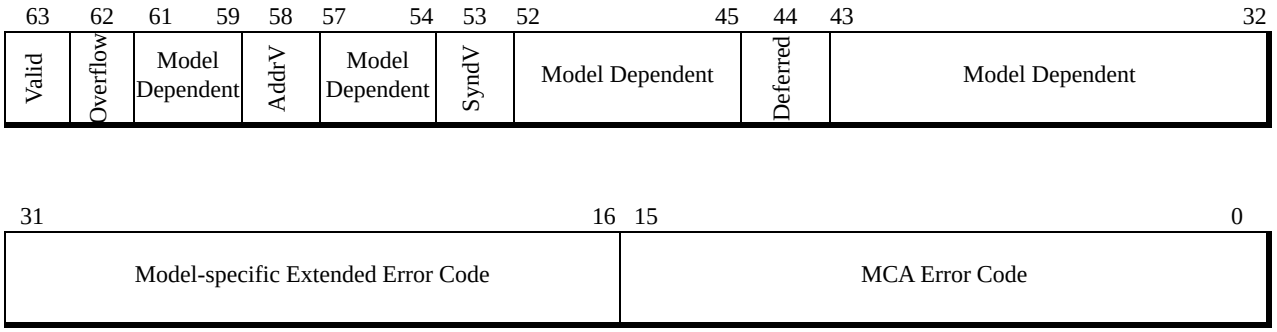


Figure 9-13. MCA_DESTAT Register

Bits	Mnemonic	Description	R/W
63	Valid	A valid error is contained in this register.	R/W
62	Overflow	One or more deferred errors was not logged.	R/W
61:59		Model Dependent.	
58	AddrV	An address is contained in MCA_DEADDR.	R/W
57:54		Model Dependent.	
53	SyndV	Syndrome valid. MCA_SYND contains error syndrome information associated with this deferred error.	R/W
52:45		Model Dependent.	
44	Deferred	Error is a deferred error.	R/W
43:32		Model Dependent.	
31:16	Model-Specific Extended Error Code	This field encodes model-specific information about the error. Valid if MCA_CONFIG[DeStatErrCodeSupported]=1, else Reserved.	R/W
15:0	MCA Error Code	This field encodes information about the error logged in MCA_DESTAT. Valid if MCA_CONFIG[DeStatErrCodeSupported]=1, else Reserved.	R/W

9.3.3.5 MCA Deferred Error Address Register

The MCA_DEADDR register provides the address associated with an error logged in MCA_DESTAT.

The format of this register is the same as MCA_ADDR.

The register is only meaningful if MCA_DESTAT[Valid]=1 and MCA_DESTAT[ADDRV]=1.

9.3.3.6 MCA Miscellaneous Registers 1 - 4

Set by hardware.

The format of MCA_MISC[4-1] is the same as the format of MCA_MISC0.

9.3.3.7 MCA Syndrome Registers 1 - 2

The MCA_SYND[2-1] registers store information associated with the error in MCA_STATUS or MCA_DESTAT. The contents of these registers are valid if MCA_STATUS[SYNDV] is set or MCA_DESTAT[SYNDV] is set.

The format and contents of this register is implementation dependent. For more information, see the Processor Programming Reference Manual applicable to your product.

9.4 Initializing the Machine-Check Mechanism

Following a processor reset, all machine-check error-reporting enable bits are disabled. System software must enable these bits before machine-check errors can be reported. Generally, system software should initialize the machine-check mechanism using the following process:

- Execute the CPUID instruction and verify that the processor supports the machine-check exception (MCE) and machine-check registers (MCA). Software should not proceed with initializing the machine-check mechanism if the machine-check registers are not supported.
- If the machine-check registers are supported, system software should take the following steps:
  - Check to see if the CTLP bit in the MCG_CAP register is set to 1. If it is, then the MCG_CTL register is supported by the processor. If the MCG_CTL register is supported, software should set its enable bits to 1 for the machine-check features it uses. Software can load MCG_CTL with all 1s to enable all available machine-check reporting banks.
  - Read the COUNT field from the MCG_CAP register to determine the number of error-reporting register banks supported by the processor. For each error-reporting register bank, software should set the enable bits to 1 in the MC*i*_CTL register for the error types it wants the processor to report. Software can write each MC*i*_CTL with all 1s to enable all error-reporting mechanisms.

    Not enabling reporting banks that may be involved in the reporting of uncorrected errors can lead to the loss of system reliability and error recoverability.
  - Check the VAL bit on each implemented MC*i*_STATUS register. It is possible that valid error-status information has already been logged in the MC*i*_STATUS registers at the time software is attempting to initialize them. The status can reflect errors logged prior to a warm reset or errors recorded during the system power-up and boot process. Before clearing the MC*i*_STATUS registers, software should examine their contents and log any errors found.
  - After saving any valid error information contained in the MC*i*_STATUS, MC*i*_ADDR, and any implemented miscellaneous error information registers for each implemented reporting bank, software should clear all status fields in the MC*i*_STATUS register for each bank by writing all 0s to the register.
- As a final step in the initialization process, system software should enable the machine-check exception by setting CR4[MCE] to 1.

A machine-check condition that occurs while CR4[MCE] is cleared will result in the processor core entering the shutdown state.

9.5 Using MCA Features

System software can detect and handle logged errors using three methods:

1. Polling

   Software can periodically examine the machine-check status registers for errors, and save any error information found. Uncorrected errors found during polling will require some type of immediate response to initiate recovery or shutdown.

2. Enabling machine-check reporting

   When reporting is enabled, any uncorrected error that occurs causes control to be transferred to the machine-check exception handler. The exception handler can be designed for a specific processor implementation or can be generalized to work on multiple implementations.

3. Setting up and enabling interrupts for deferred and corrected errors

   In many implementations, MCA hardware can be configured to generate an interrupt hardware on the detection of a deferred error or when a programmed corrected error threshold is reached.

These methods are not mutually exclusive.

9.5.1 Determining the Scope of Detected Errors

Table 9-4 details the actions that recovery software should take and the level of recovery possible based on status information returned in the MCi_STATUS and MCG_STATUS registers.

Table 9-4. Error Scope

MC*i*_STATUS				Error Scope
PCC	TCC	UC	Deferred	
1	—	1	—	System fatal error. Error has corrupted the processor core architectural state. System processing must be terminated.
0	0	1	—	Recoverable error. If software can correct the error, the interrupted program can be resumed.
0	1	1	—	Containable error. The interrupted instruction stream cannot be resumed. System-level recovery may be possible if software can localize the error and terminate any affected software processes.
0	0	0	1	Deferred error. Immediate software action is not required. A latent error has been discovered, but not yet consumed. Error handling software may attempt to correct this data error, or prevent access by processes which map the data, or make the physical resource containing the data inaccessible.
0	0	0	0	Hardware corrected error. No software action is required. Error information should be saved for analysis.

9.5.2 Handling Machine Check Exceptions

The processor uses the interrupt control-transfer mechanism to invoke an exception handler after a machine-check exception occurs. This requires system software to initialize the interrupt-descriptor table (IDT) with either an interrupt gate or a trap gate that references the interrupt handler. See “Legacy Protected-Mode Interrupt Control Transfers” on page 261 and “Long-Mode Interrupt Control Transfers” on page 272 for more information on interrupt control transfers.

At a minimum, the machine-check exception handler must be capable of logging errors for later examination. This can be a sufficient implementation for some handlers. More thorough exception-handler implementations can analyze the error to determine if it is unrecoverable, and whether it can be recovered in software.

Machine-check exception handlers that attempt recovery must be thorough in their analysis and their corrective actions. The following guidelines should be used when writing such a handler:

- The status registers in all the enabled error-reporting register banks must be examined to identify the cause of the machine-check exception. Read the COUNT field from MCG_CAP to determine the number of status registers supported by the processor.
- Check the valid bit in each status register (MC*i*_STATUS[VAL]). The MC*i*_STATUS register does not need to be examined when its valid bit is clear.
- Check the valid MC*i*_STATUS registers to see if error recovery is possible. Error recovery is not possible when:
  - The processor-context corrupt bit (MC*i*_STATUS[PCC]) is set to 1.
  - The error-overflow status bit (MC*i*_STATUS[OVER]) is set and the processor does not support recoverable MC*i*_STATUS overflow (as indicated by feature bit CPUID Fn8000_0007_EBX[McaOverflowRecov] = 0).
  - The processor does not support Machine Check Recovery as indicated by feature bit CPUID Fn8000_0007_EBX[SUCCOR].

  If error recovery is not possible, the handler should log the error information and return to the system software responsible for shutting down the processor core.
- Check the MC*i*_STATUS[UC] bit to see if the processor corrected the error. If UC is set, the processor did not correct the error and the exception handler must correct the error before restarting the interrupted program.
  - If MCA Recovery is supported:
    - Check MCA_STATUS[TCC].
    - If TCC is set, the context of the process executing on the interrupted thread may be corrupt and the thread cannot be recovered. The rest of the system is unaffected; it is possible to terminate only the affected process thread.
    - If TCC is clear, the context of the process thread executing on the interrupted logical core is not corrupt. Recovery of the process thread may be possible, but only if the uncorrected error condition is first corrected by software; otherwise, the interrupted process thread must be terminated.

If the handler cannot correct the error or the MCG_STATUS[RIPV] bit is cleared, it should not return control to the interrupted program, but should log the error information and terminate the software process that was about to consume the uncorrected data. If the error has compromised the state of a guest operating system, the guest should be restarted. If the state of the virtual machine has been corrupted, the virtual machine must be reinitialized.

- When identifying the error condition, portable exception handlers should examine only the architecturally defined fields of the MC*i*_STATUS register.
- If the MCG_STATUS[RIPV] bit is set, the interrupted program can be restarted reliably at the instruction pointer address pushed onto the exception handler stack. If RIPV = 0, the interrupted program cannot be restarted reliably at that location, although it can be restarted at that location for debugging purposes.
- When logging errors, particularly those that are not recoverable, check the MCG_STATUS[EIPV] bit to see if the instruction-pointer address pushed onto the exception handler stack is related to the process thread interrupted by the machine-check exception. If EIPV = 0, the address is not guaranteed to be related to the interrupted process thread.
- Before exiting the machine-check handler, clear the MCG_STATUS[MCIP] bit. MCIP indicates a machine-check exception occurred. If this bit is set when another machine-check exception occurs, the processor enters the shutdown state.
- When an exception handler is able to, at a minimum, successfully log an error condition, the MC*i*_STATUS registers should be cleared before exiting the machine-check handler. Software is responsible for clearing at least the MC*i*_STATUS[VAL] bits.
- Additional machine-check exception-handler portability can be added by having the handler use the CPUID instruction to identify the processor and its capabilities. Implementation-specific software can be added to the machine-check exception handler based on the processor information reported by CPUID.

9.5.3 Reporting Corrected Errors

Machine-check exceptions do not occur if the error is corrected by the processor. If system software wishes to detect and save information concerning corrected machine-check errors, a system-service routine must be provided to check the contents of the machine-check status registers for corrected errors. The service routine can be invoked by system software on a periodic basis, or by an error-thresholding interrupt.

A service routine that gathers error information for corrected errors should perform the following:

- Examine the status register (MC*i*_STATUS) in each of the enabled error-reporting register banks. For each MC*i*_STATUS register with a set valid bit (VAL=1), the service routine should:
  - Save the contents of the MC*i*_STATUS register.
  - Save the contents of the corresponding MC*i*_ADDR register if MC*i*_STATUS[ADDRV] = 1.
  - Save the contents of the corresponding MC*i*_MISC register if MC*i*_STATUS[MISCV] = 1.

- Once the information found in the error-reporting register banks is saved, the MC*i*_STATUS register should be cleared. This allows the processor to properly report any subsequent errors in the MC*i*_STATUS registers.
- The service routine can save the time-stamp counter with each error logged. This can help in determining how frequently errors occur. For further information, see “Time-Stamp Counter” on page 407.
- In multiprocessor configurations, the service routine can save the processor-node identifier. This can help locate a failing multiprocessor-system component, which can then be isolated from the rest of the system. For further information, see the documentation for particular implementations of the architecture.

10 System-Management Mode

System-management mode (SMM) is an operating mode designed for system-control activities like power management. Normally, these activities are transparent to conventional operating systems and applications. SMM is used by platform firmware and specialized low-level device drivers, rather than the operating system.

The SMM interrupt-handling mechanism differs substantially from the standard interrupt-handling mechanism described in Chapter 8, “Exceptions and Interrupts.” SMM is entered using a special external interrupt called the *system-management interrupt* (SMI). After an SMI is received by the processor, the processor saves the processor state in a separate address space, called *SMRAM*. The SMM-handler software and data structures are also located in the SMRAM space. Interrupts and exceptions that ordinarily cause control transfers to the operating system are disabled when SMM is entered. The processor exits SMM, restores the saved processor state, and resumes normal execution by using a special instruction, *RSM*.

In SMM, address translation is disabled and addressing is similar to real mode. SMM programs can address up to 4 Gbytes of physical memory. See “SMM Operating-Environment” on page 327 for additional information on memory addressing in SMM.

The following sections describe the components of the SMM mechanism:

- *“SMM Resources” on page 318*—this section describes SMRAM, the SMRAM save-state area used to hold the processor state, and special SMRAM save-state entries used in support of SMM.
- *“Using SMM” on page 327*—this section describes the mechanism of entering and exiting SMM. It also describes SMM memory allocation, addressing, and interrupts and exceptions.

Of these mechanisms, only the format of the SMRAM save-state area differs between the AMD64 architecture and the legacy architecture.

Note: *Model-independent aspects of SMM operation are described here; see the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual of a given processor family for possible model-specific details.*

10.1 SMM Differences

There are functional differences between the SMM support in the AMD64 architecture and the SMM support found in previous architectures. These are:

- The SMRAM state-save area layout is changed to hold the 64-bit processor state.
- The initial processor state upon entering SMM is expanded to reflect the 64-bit nature of the processor.
- New conditions exist that can cause a processor shutdown while in SMM.

- The auto-halt restart and I/O-instruction restart entries in the SMRAM state-save area are one byte each instead of two bytes each.
- SMRAM caching considerations are modified because the legacy FLUSH# external signal (writeback, if modified, and invalidate) is not supported on implementations of the AMD64 architecture.
- Some previous AMD x86 processors saved and restored the CR2 register in the SMRAM state-save area. This register is not saved by the SMM implementation in the AMD64 architecture. SMM handlers that save and restore CR2 must perform the operation in software.

10.2 SMM Resources

The SMM resources supported by the processor consist of SMRAM, the SMRAM state-save area, and special entries within the SMRAM state-save area. In addition to the save-state area, SMRAM includes space for the SMM handler.

10.2.1 SMRAM

SMRAM is the memory-address space accessed by the processor when in SMM. The default size of SMRAM is 64 Kbytes and can range in size between 32 Kbytes and 4 Gbytes. System logic can use physically separate SMRAM and main memory, directing memory transactions to SMRAM after recognizing SMM is entered, and redirecting memory transactions back to system memory after recognizing SMM is exited. When separate SMRAM and main memory are used, the system designer needs to provide a method of mapping SMRAM into main memory so that the SMI handler and data structures can be loaded.

Figure 10-1 on page 319 shows the default SMRAM memory map. The default SMRAM code-segment (CS) has a base address of 0003_0000h (the base address is automatically scaled by the processor using the CS-selector register, which is set to the value 3000h). This default SMRAM-base address is known as *SMBASE*. A 64-Kbyte memory region, addressed from 0003_0000h to 0003_FFFFh, makes up the default SMRAM memory space. The top 32 Kbytes (0003_8000h to 0003_FFFFh) must be supported by system logic, with physical memory covering that entire address range. The top 512 bytes (0003_FE00h to 0003_FFFFh) of this address range are the default *SMM state-save area*. The default entry point for the SMM interrupt handler is located at 0003_8000h.

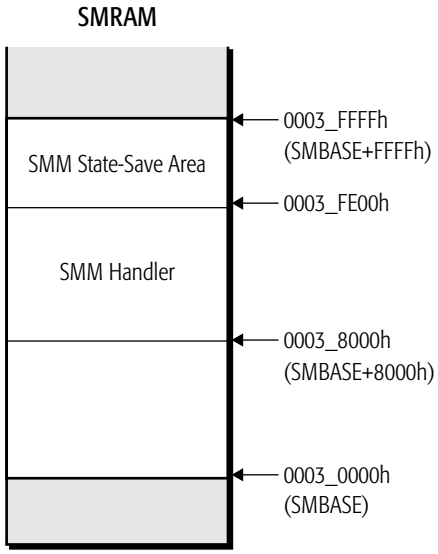


Figure 10-1. Default SMRAM Memory Map

10.2.2 SMBASE Register

The format of the SMBASE register is shown in Figure 10-2. SMBASE is an internal processor register that holds the value of the SMRAM-base address. SMBASE is set to 30000h after a processor reset.

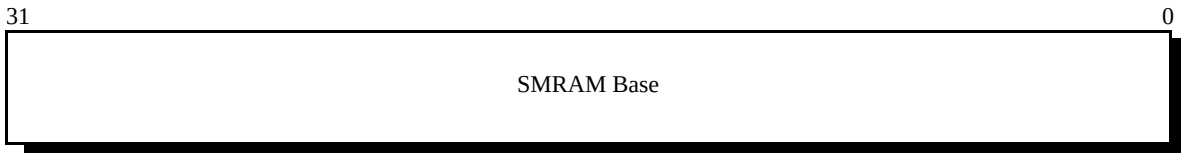


Figure 10-2. SMBASE Register

In some operating environments, relocation of SMRAM to a higher memory area can provide more low memory for legacy software. SMBASE relocation is supported when the SMM-base relocation bit in the SMM-revision identifier (bit 17) is set to 1. In processors implementing the AMD64 architecture, SMBASE relocation is always supported.

Software can only modify SMBASE (relocate the SMRAM-base address) by entering SMM, modifying the SMBASE image stored in the SMRAM state-save area, and exiting SMM. The SMM-

handler entry point must be loaded at the new memory location specified by SMBASE+8000h. The next time SMM is entered, the processor saves its state in the new state-save area at SMBASE+0FE00h, and begins executing the SMM handler at SMBASE+8000h. The new SMBASE address is used for every SMM until it is changed, or a hardware reset occurs.

When SMBASE is used to relocate SMRAM to an address above 1 Mbyte, 32-bit address-size-override prefixes must be used to access this memory. This is because addressing in SMM behaves as it does in real mode, with a 16-bit default operand size and address size. The values in the 16-bit segment-selector registers are left-shifted four bits to form a 20-bit segment-base address. Without using address-size overrides, the maximum computable address is 10FFEFh.

Because SMM memory-addressing is similar to real-mode addressing, the SMBASE address must be less than 4 Gbytes.

10.2.3 SMRAM State-Save Area

When an SMI occurs, the processor saves its state in the 512-byte SMRAM state-save area during the control transfer into SMM. The format of the state-save area defined by the AMD64 architecture is shown in Table 10-1. This table shows the offsets in the SMRAM state-save area relative to the SMRAM-base address. The state-save area is located between offset 0_FE00h (SMBASE+0_FE00h) and offset 0_FFFFh (SMBASE+0_FFFFh). Software should not modify offsets specified as read-only or reserved, otherwise unpredictable results can occur.

Table 10-1. AMD64 Architecture SMM State-Save Area

Offset (Hex) from SMBASE	Contents		Size	Allowable Access
FE00h	ES	Selector	Word	Read-Only
FE02h		Attributes	Word	
FE04h		Limit	Doubleword	
FE08h		Base	Quadword	
FE10h	CS	Selector	Word	Read-Only
FE12h		Attributes	Word	
FE14h		Limit	Doubleword	
FE18h		Base	Quadword	
FE20h	SS	Selector	Word	Read-Only
FE22h		Attributes	Word	
FE24h		Limit	Doubleword	
FE28h		Base	Quadword	

Note:
1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

Table 10-1. AMD64 Architecture SMM State-Save Area (continued)

Offset (Hex) from SMBASE	Contents		Size	Allowable Access
FE30h	DS	Selector	Word	Read-Only
FE32h		Attributes	Word	
FE34h		Limit	Doubleword	
FE38h		Base	Quadword	
FE40h	FS	Selector	Word	Read-Only
FE42h		Attributes	Word	
FE44h		Limit	Doubleword	
FE48h		Base	Quadword	
FE50h	GS	Selector	Word	Read-Only
FE52h		Attributes	Word	
FE54h		Limit	Doubleword	
FE58h		Base	Quadword	
FE60h–FE63h	GDTR	Reserved	4 Bytes	Read-Only
FE64h		Limit	Word	
FE66h–FE67h		Reserved	2 Bytes	
FE68h		Base	Quadword	
FE70h	LDTR	Selector	Word	Read-Only
FE72h		Attributes	Word	
FE74h		Limit	Doubleword	
FE78h		Base	Quadword	
FE80h–FE83h	IDTR	Reserved	4 Bytes	Read-Only
FE84h		Limit	Word	
FE86h–FE87h		Reserved	2 Bytes	
FE88h		Base	Quadword	
FE90h	TR	Selector	Word	Read-Only
FE92h		Attributes	Word	
FE94h		Limit	Doubleword	
FE98h		Base	Quadword	
FEA0h	I/O Instruction Restart RIP		Quadword	Read-Only
FEA8h	I/O Instruction Restart RCX		Quadword	Read-Only
FEB0h	I/O Instruction Restart RSI		Quadword	Read-Only
FEB8h	I/O Instruction Restart RDI		Quadword	Read-Only
FEC0h	I/O Instruction Restart Dword		Doubleword	Read-Only
FEC4h–FEC7h	Reserved		4 Bytes	—

Note:

1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

Table 10-1. AMD64 Architecture SMM State-Save Area (continued)

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FEC8h	I/O Instruction Restart	Byte	Read/Write
FEC9h	Auto-Halt Restart	Byte	
FECAh—FECFh	Reserved	6 Bytes	—
FED0h	EFER	Quadword	Read-Only
FED8h	SVM Guest	Quadword	Read-Only
FEE0h	SVM Guest VMCB Physical Address	Quadword	
FEE8h	SVM Guest Virtual Interrupt	Quadword	
FEF0h—FEFBh	Reserved	12 Bytes	—
FEFCh	SMM-Revision Identifier[1]	Doubleword	Read-Only
FF00h	SMBASE	Doubleword	Read/Write
FF04h—FF17h	Reserved	20 Bytes	—
FF18h	SSP	Quadword	Read/Write
FF20h	SVM Guest PAT	Quadword	Read-Only
FF28h	SVM Host EFER	Quadword	
FF30h	SVM Host CR4	Quadword	
FF38h	SVM Host CR3	Quadword	
FF40h	SVM Host CR0	Quadword	
FF48h	CR4	Quadword	Read-Only
FF50h	CR3	Quadword	
FF58h	CR0	Quadword	
FF60h	DR7	Quadword	Read-Only
FF68h	DR6	Quadword	
FF70h	RFLAGS	Quadword	Read/Write
FF78h	RIP	Quadword	Read/Write
FF80h	R15	Quadword	
FF88h	R14	Quadword	
FF90h	R13	Quadword	
FF98h	R12	Quadword	
FFA0h	R11	Quadword	
FFA8h	R10	Quadword	
FFB0h	R9	Quadword	
FFB8h	R8	Quadword	

Note:
1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

Table 10-1. AMD64 Architecture SMM State-Save Area (continued)

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FFC0h	RDI	Quadword	Read/Write
FFC8h	RSI	Quadword	
FFD0h	RBP	Quadword	
FFD8h	RSP	Quadword	
FFE0h	RBX	Quadword	
FFE8h	RDX	Quadword	
FFF0h	RCX	Quadword	
FFF8h	RAX	Quadword	

Note:
1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

A number of other registers are not saved or restored automatically by the SMM mechanism. See "Saving Additional Processor State" on page 329 for information on using these registers in SMM.

As a reference for legacy processor implementations, the legacy SMM state-save area format is shown in Table 10-2. *Implementations of the AMD64 architecture do not use this format.*

Table 10-2. Legacy SMM State-Save Area (Not used by AMD64 Architecture)

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FE00h—FEF7h	Reserved	248 Bytes	—
FEF8h	SMBASE	Doubleword	Read/Write
FEFCh	SMM-Revision Identifier	Doubleword	Read-Only
FF00h	I/O Instruction Restart	Word	Read/Write
FF02h	Auto-Halt Restart	Word	
FF04h—FF87h	Reserved	132 Bytes	—
FF88h	GDT Base	Doubleword	Read-Only
FF8Ch—FF93h	Reserved	Quadword	—
FF94h	IDT Base	Doubleword	Read-Only
FF98h—FFA7h	Reserved	16 Bytes	—

Note:
1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

Table 10-2. Legacy SMM State-Save Area (Not used by AMD64 Architecture) (continued)

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FFA8h	ES	Doubleword	Read-Only
FFACh	CS	Doubleword	
FFB0h	SS	Doubleword	
FFB4h	DS	Doubleword	
FFB8h	FS	Doubleword	
FFBCh	GS	Doubleword	
FFC0h	LDT Base	Doubleword	Read-Only
FFC4h	TR	Doubleword	
FFC8h	DR7	Doubleword	Read-Only
FFCCh	DR6	Doubleword	
FFD0h	EAX	Doubleword	Read/Write
FFD4h	ECX	Doubleword	
FFD8h	EDX	Doubleword	
FFDCh	EBX	Doubleword	
FFE0h	ESP	Doubleword	
FFE4h	EBP	Doubleword	
FFE8h	ESI	Doubleword	
FFECh	EDI	Doubleword	
FFF0h	EIP	Doubleword	Read/Write
FFF4h	EFLAGS	Doubleword	Read/Write
FFF8h	CR3	Doubleword	Read-Only
FFFCh	CR0	Doubleword	

Note:
1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

10.2.4 SMM-Revision Identifier

The SMM-revision identifier specifies the SMM version and the available SMM extensions implemented by the processor. Software reads the SMM-revision identifier from offset FEFCh in the SMM state-save area of SMRAM. This offset location is compatible with earlier versions of SMM. Software must not write to this location. Doing so can produce undefined results. Figure 10-3 on page 325 shows the format of the SMM-revision identifier.

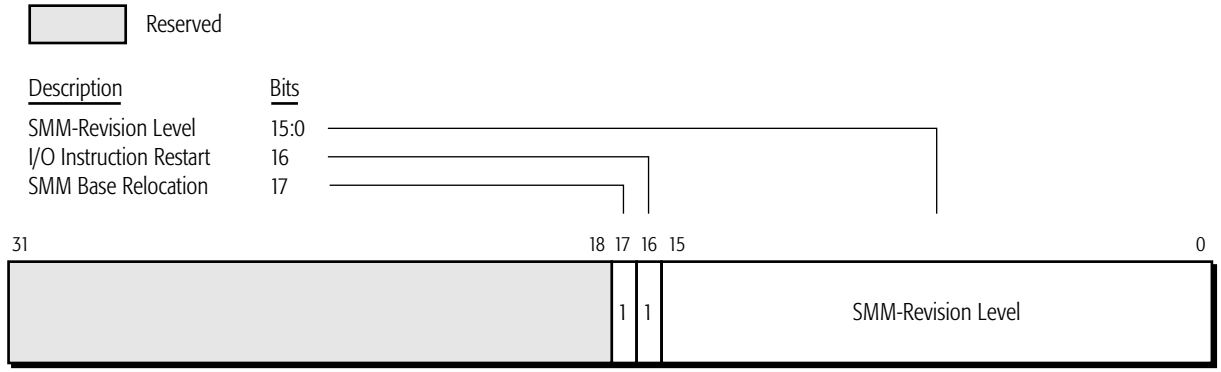


Figure 10-3. SMM-Revision Identifier

The fields within the SMM-revision identifier are:

- *SMM-revision Level*—Bits 15:0. Specifies the version of SMM supported by the processor. The SMM-revision level is of the form 0_xx64h, where *xx* starts with 00 and is incremented for later revisions to the SMM mechanism.
- *I/O Instruction Restart*—Bit 16. When set to 1, the processor supports restarting I/O instructions that are interrupted by an SMI. This bit is always set to 1 by implementations of the AMD64 architecture. See "I/O Instruction Restart" on page 331 for information on using this feature.
- *SMM Base Relocation*—Bit 17. When set to 1, the processor supports relocation of SMRAM. This bit is always set to 1 by implementations of the AMD64 architecture. See "SMBASE Register" on page 319 for information on using this feature.

All remaining bits in the SMM-revision identifier are reserved.

10.2.5 SMRAM Protected Areas

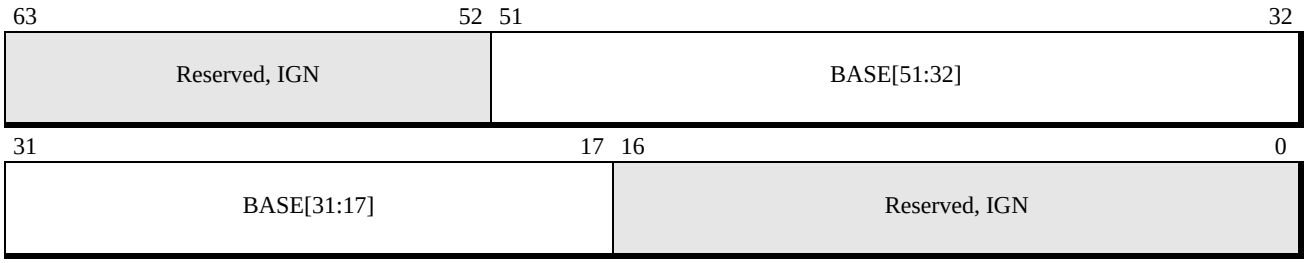
Two areas are provided as safe areas for SMM code and data that are not readily accessible by non-SMM applications. The SMI handler can be located in one of these two ranges, or it can be located outside of these ranges. The handler is placed in the desired range by setting SMBASE accordingly.

The ASeg range is located at a fixed address from A_0000h to B_FFFFh. The TSeg range is located at a variable base specified by the SMM_ADDR MSR with a variable size specified by the SMM_MASK MSR. These ranges must never overlap.

Each CPU memory access is in the TSeg range if the following is true:

Phys Addr[51:17] & SMM_MASK[51:17] = SMM_ADDR[51:17] & SMM_MASK[51:17].

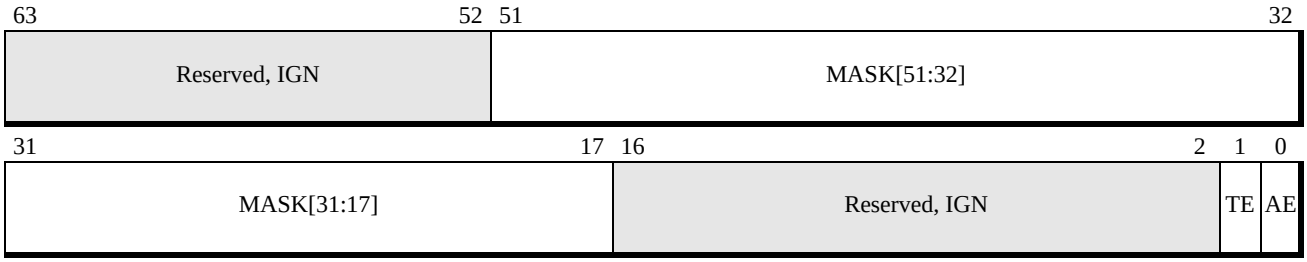
For example, if the TSeg range spans 256 Kbytes starting at address 10_0000h, then SMM_ADDR =0010_0000h and SMM_MASK=FFFC_0000h. This results in a TSeg address range from 0010_0000 to 0013_FFFFh. The TSeg range must be aligned to a 128 Kbyte boundary and the minimum TSeg size is 128 Kbytes.



Bits	Mnemonic	Description	R/W
63:52	Reserved	Reserved, Ignore	
51:17	BASE	SMM TSeg Base Address	R/W
16:0	Reserved	Reserved, Ignore	

Figure 10-4. SMM_ADDR Register Format

- *SMM TSeg Base Address (BASE)*—Bits 51:17. Specifies the base address of the TSeg range of protected addresses.



Bits	Mnemonic	Description	R/W
63:52	Reserved	Reserved, Ignore	
51:17	MASK	TSeg Mask	R/W
16:2	Reserved	Reserved, Ignore	
1	TE	Tseg Address Range Enable	R/W
0	AE	Aseg Address Range Enable	R/W

Figure 10-5. SMM_MASK Register Format

- *ASeg Address Range Enable (AE)*—Bit 0. Specifies whether the ASeg address range is enabled for protection. When the bit is set to 1, the ASeg address range is enabled for protection. When cleared to 0, the ASeg address range is disabled for protection.

- *TSeg Address Range Enable (TE)*—Bit 1. Specifies whether the TSeg address range is enabled for protection. When the bit is set to 1, the TSeg address range is enabled for protection. When cleared to 0, the TSeg address range is disabled for protection.
- *TSeg Mask (MASK)*—Bits 51:17. Specifies the mask used to determine the TSeg range of protected addresses. The physical address is in the TSeg range if the following is true:

  Phys Addr[51:17] & SMM_MASK[51:17] = SMM_ADDR[51:17] & SMM_MASK[51:17].

Note that a processor is not required to implement all 52 bits of the physical address.

10.3 Using SMM

10.3.1 System-Management Interrupt (SMI)

SMM is entered using the system-management interrupt (SMI). SMI is an external non-maskable interrupt that operates differently from and independently of other interrupts. SMI has priority over all other external interrupts, including NMI (see “Priorities” on page 255 for a list of the interrupt priorities). SMIs are disabled when in SMM, which prevents reentrant calls to the SMM handler.

When an SMI is received by the processor, the processor stops fetching instructions and waits for currently-executing instructions to complete and write their results. The SMI also waits for all buffered memory writes to update the caches or system memory. When these activities are complete, the processor uses implementation-dependent external signaling to acknowledge back to the system that it has received the SMI.

10.3.2 SMM Operating-Environment

The SMM operating-environment is similar to real mode, except that the segment limits in SMM are 4 Gbytes rather than 64 Kbytes. This allows an SMM handler to address memory in the range from 0h to 0FFFF_FFFFh. As with real mode, segment-base addresses are restricted to 20 bits in SMM, and the default operand-size and address-size is 16 bits. To address memory locations above 1 Mbyte, the SMM handler must use the 32-bit operand-size-override and address-size-override prefixes.

After saving the processor state in the SMRAM state-save area, a processor running in SMM sets the segment-selector registers and control registers into a state consistent with real mode. Other registers are also initialized upon entering SMM, as shown in Table 10-3.

Table 10-3. SMM Register Initialization

Register		Initial SMM Contents
CS	Selector	SMBASE right-shifted 4 bits
	Base	SMBASE
	Limit	FFFF_FFFFh
	Attr	Read-Write-Execute

Table 10-3. SMM Register Initialization (continued)

Register		Initial SMM Contents
DS, ES, FS, GS, SS	Selector	0000h
	Base	0000_0000_0000_0000h
	Limit	FFFF_FFFFh
	Attr	Read-Write
RIP		0000_0000_0000_8000h
RFLAGS		0000_0000_0000_0002h
CR0		PE, EM, TS, PG bits cleared to 0. All other bits are unmodified.
CR4		0000_0000_0000_0000h
DR7		0000_0000_0000_0400h
EFER		0000_0000_0000_0000h

10.3.3 Exceptions and Interrupts

All hardware interrupts are disabled upon entering SMM, but exceptions and software interrupts are not disabled. If necessary, the SMM handler can re-enable hardware interrupts. Software that handles interrupts in SMM should consider the following:

- *SMI*—If an SMI occurs while the processor is in SMM, it is latched by the processor. The latched SMI occurs when the processor leaves SMM.
- *NMI*—If an NMI occurs while the processor is in SMM, it is latched by the processor, but the NMI handler is not invoked until the processor leaves SMM with the execution of an RSM instruction. A pending NMI causes the handler to be invoked immediately after the RSM completes and before the first instruction in the interrupted program is executed.

  An SMM handler can unmask NMI interrupts by simply executing an IRET. Upon completion of the IRET instruction, the processor recognizes the pending NMI, and transfers control to the NMI handler. Once an NMI is recognized within SMM using this technique, subsequent NMIs are recognized until SMM is exited. Later SMIs cause NMIs to be masked, until the SMM handler unmasks them.
- *Exceptions*—Exceptions (internal processor interrupts) are not disabled and can occur while in SMM. Therefore, the SMM-handler software should be written to avoid generating exceptions.
- *Software Interrupts*—The software-interrupt instructions (BOUND, INT*n*, INT3, and INTO) can be executed while in SMM. However, it is not recommended that the SMM handler use these instructions.
- *Maskable Interrupts*—RFLAGS.IF is cleared to 0 by the processor when SMM is entered. Software can re-enable maskable interrupts while in SMM, but it must follow the guidelines listed below for handling interrupts.
- *Debug Interrupts*—The processor disables the debug interrupts when SMM is entered by clearing DR7 to 0 and clearing RFLAGS.TF to 0. The SMM handler can re-enable the debug facilities while in SMM, but it must follow the guidelines listed below for handling interrupts.

- *INIT*—The processor does not recognize INIT while in SMM.

Because the RFLAGS.IF bit is cleared when entering SMM, the HLT instruction should not be executed in SMM without first setting the RFLAGS.IF bit to 1. Setting this bit to 1 allows the processor to exit the halt state by using an external maskable interrupt.

In the cases where an SMM handler must accept and handle interrupts and exceptions, several guidelines must be followed:

- Interrupt handlers must be loaded and accessible before enabling interrupts.
- A real-mode interrupt vector table located at virtual (linear) address 0 is required.
- Segments accessed by the interrupt handler cannot have a base address greater than 20 bits because of the real-mode addressing used in SMM. In SMM, the 16-bit value stored in the segment-selector register is left-shifted four bits to form the 20-bit segment-base address, like real mode.
- Only the IP (rIP[15:0]) is pushed onto the stack as a result of an interrupt in SMM, because of the real-mode addressing used in SMM. If the SMM handler is interrupted at a code-segment offset above 64 Kbytes, then the return address on the stack must be adjusted by the interrupt-handler, and a RET instruction with a 32-bit operand-size override must be used to return to the SMM handler.
- If the interrupt-handler is located below 1 Mbyte, and the SMM handler is located above 1 Mbyte, a RET instruction cannot be used to return to the SMM handler. In this case, the interrupt handler can adjust the return pointer on the stack, and use a far CALL to transfer control back to the SMM handler.

10.3.4 Invalidating the Caches

The processor can cache SMRAM-memory locations. If the system implements physically separate SMRAM and system memory, it is possible for SMRAM and system memory locations to alias into identical cache locations. In some processor implementations, the cache contents must be written to memory and invalidated when SMM is entered *and* exited. This prevents the processor from using previously-cached main-memory locations as aliases for SMRAM-memory locations when SMM is entered, and vice-versa when SMM is exited.

Implementations of the AMD64 architecture *do not require cache invalidation* when entering and exiting SMM. Internally, the processor keeps track of SMRAM and system-memory accesses separately and properly handles situations where aliasing occurs. Cached system memory and SMRAM locations can persist across SMM mode changes. Removal of the requirement to writeback and invalidate the cache simplifies SMM entry and exit and allows SMM code to execute more rapidly.

10.3.5 Saving Additional Processor State

Several registers are not saved or restored automatically by the SMM mechanism. These are:

- The 128-bit media instruction registers.
- The 64-bit media instruction registers.

- The x87 floating-point registers.
- The page-fault linear-address register (CR2).
- The task-priority register (CR8).
- The debug registers, DR0, DR1, DR2, and DR3.
- The memory-type range registers (MTRRs).
- Model-specific registers (MSRs).

These registers are not saved because SMM handlers do not normally use or modify them. If an SMI results in a processor reset (due to powering down the processor, for example) or the SMM handler modifies the contents of the unsaved registers, the SMM handler should save and restore the original contents of those registers. The unsaved registers, along with those stored in the SMRAM state-save area, need to be saved in a non-volatile storage location if a processor reset occurs. The SMM handler should execute the CPUID instruction to determine the feature set available in the processor, and be able to save and restore the registers required by those features. For more information on using the CPUID instruction, see Section 3.3, “Processor Feature Identification,” on page 71.

The SMM handler can execute any of the 128-bit media, 64-bit media, or x87 instructions. A simple method for saving and restoring those registers is to use the FXSAVE and FXRSTOR instructions, respectively, if it is supported by the processor. See “Saving Media and x87 Execution Unit State” on page 342 for information on saving and restoring those registers.

Floating-point exceptions can occur when the SMM handler uses media or x87 floating-point instructions. If the SMM handler uses floating-point exception handlers, they must follow the usage guidelines established in “Exceptions and Interrupts” on page 328. A simple method for dealing with floating-point exceptions while in SMM is to simply mask all exception conditions using the appropriate floating-point control register. When the exceptions are masked, the processor handles floating-point exceptions internally in a default manner, and allows execution to continue uninterrupted.

10.3.6 Operating in Protected Mode and Long Mode

Software can enable protected mode from SMM and it can also enable and activate long mode. An SMM handler can use this capability to enter 64-bit mode and save additional processor state that cannot be accessed from outside 64-bit mode (for example, the most-significant 32 bits of CR2).

10.3.7 Auto-Halt Restart

The auto-halt restart entry is located at offset FEC9h in the SMM state-save area. The size of this field is one byte, as compared with two bytes in previous versions of SMM.

When entering SMM, the processor loads the auto-halt restart entry to indicate whether SMM was entered from the halt state, as follows:

- Bit 0 indicates the processor state upon entering SMM:
  - When set to 1, the processor entered SMM from the halt state.

- When cleared to 0, the processor did not enter SMM from the halt state.

- Bits 7:1 are cleared to 0.

The SMM handler can write the auto-halt restart entry to specify whether the return from SMM should take the processor back to the halt state or to the instruction-execution state specified by the SMM state-save area. The values written are:

- *Clear to 00h*—The processor returns to the state specified by the SMM state-save area.
- *Set to any non-zero value*—The processor returns to the halt state.

If the return from SMM takes the processor back to the halt state, the HLT instruction is not re-executed. However, the halt special bus-cycle is driven on the processor bus after the RSM instruction executes.

The result of entering SMM from a non-halt state and returning to a halt state is not predictable.

10.3.8 I/O Instruction Restart

The I/O-instruction restart entry is located at offset FEC8h in the SMM state-save area. The size of this field is one byte, as compared with two bytes in previous versions of SMM. The I/O-instruction restart mechanism is supported when the I/O-instruction restart bit (bit 16) in the SMM-revision identifier is set to 1. This bit is always set to 1 in the AMD64 architecture.

When an I/O instruction is interrupted by an SMI, the I/O-instruction restart entry specifies whether the interrupted I/O instruction should be re-executed following an RSM that returns from SMM. Re-executing a trapped I/O instruction is useful, for example, when an I/O write is performed to a powered-down disk drive. When this occurs, the system logic monitoring the access can issue an SMI to have the SMM handler power-up the disk drive and retry the I/O write. The SMM handler does this by querying system logic and detecting the failed I/O write, asking system logic to initiate the disk-drive power-up sequence, enabling the I/O instruction restart mechanism, and returning from SMM. Upon returning from SMM, the I/O write to the disk drive is restarted.

When an SMI occurs, the processor always clears the I/O-instruction restart entry to 0. If the SMI interrupted an I/O instruction, then the SMM handler can modify the I/O-instruction restart entry as follows:

- *Clear to 00h (default value)*—The I/O instruction is not restarted, and the instruction following the interrupted I/O-instruction is executed. When a REP (repeat) prefix is used with an I/O instruction, it is possible that the next instruction to be executed is the next I/O instruction in the repeat loop.
- *Set to any non-zero value*—The I/O instruction is restarted.

While in SMM, the handler must determine the cause of the SMI and examine the processor state at the time the SMI occurred to determine whether or not an I/O instruction was interrupted. Implementations provide state information in the SMM save-state area to assist in this determination:

- I/O Instruction Restart DWORD—indicates whether the SMI interrupted an I/O instruction, and saves extra information describing the I/O instruction.

- I/O Instruction Restart RIP—the RIP of the interrupted I/O instruction.
- I/O Instruction Restart RCX—the RCX of the interrupted I/O instruction.
- I/O Instruction Restart RSI—the RSI of the interrupted I/O instruction.
- I/O Instruction Restart RDI—the RDI of the interrupted I/O instruction.

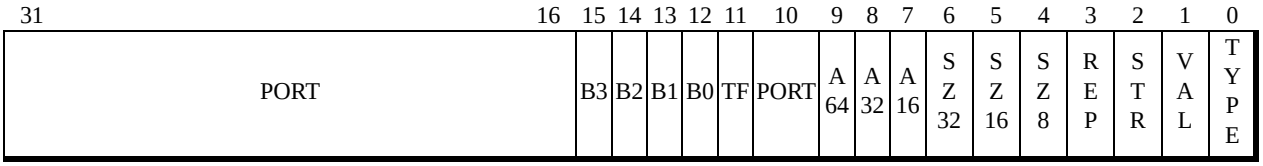


Figure 10-6. I/O Instruction Restart Dword

The fields in the I/O Instruction Restart DWORD are as follows:

- B3—DR6.B3 status
- B2—DR6.B2 status
- B1—DR6.B1 status
- B0 —DR6.B0 status
- TF—The I/O instruction was interrupted with Single Stepping (EFLAGS.TF = 1)
- PORT—Intercepted I/O port
- A64—64-bit address
- A32—32-bit address
- A16—16-bit address
- SZ32—32-bit I/O port size
- SZ16—16-bit I/O port size
- SZ8—8-bit I/O port size
- REP—Repeated port access
- STR—String based port access (INS, OUTS)
- VAL—Valid (SMI was detected during an I/O instruction.)
- TYPE—Access type (0 = OUT instruction, 1 = IN instruction).

10.3.9 SMM Page Configuration Lock

The SMM Page Configuration Lock feature allows the SMM handler to lock the paging configuration. The feature is enabled by setting HWCR.SMM_PGCFG_LOCK=1 (bit 33). Once locked, the paging configuration cannot be modified until SMM is exited using the RSM instruction. The processor clears HWCR.SMM_PGCFG_LOCK when completing the RSM instruction. If page configuration locking is needed when the processor enters SMM again in the future, HWCR.SMM_PGCFG_LOCK must be set again by the SMM handler.

When SMM Page Configuration Lock is enabled, the following will result in a #GP exception:

- Writing Extended Feature Register (EFER) using the WRMSR instruction.
- Writing CR0, CR3, or CR4 using the MOV CRn instruction.

Attempting to set HWCR.SMM_PGCFG_LOCK when not in SMM results in a #GP exception. Before setting HWCR.SMM_PGCFG_LOCK, system software must verify the processor supports the SMM Page Configuration Lock feature by checking that CPUID Fn8000_0021_EAX[SmmPgCfgLock] (bit 3) = 1. For more information on using the CPUID instruction see Section 3.3, “Processor Feature Identification,” on page 71.

10.4 Leaving SMM

Software leaves SMM and returns to the interrupted program by executing the RSM instruction. RSM causes the processor to load the interrupted state from the SMRAM state-save area and then transfer control back to the interrupted program. RSM cannot be executed in any mode other than SMM, otherwise an invalid-opcode exception (#UD) occurs.

An RSM causes a processor shutdown if an invalid-state condition is found in the SMRAM state-save area. Only an external reset, external processor-initialization, or non-maskable external interrupt (NMI) can cause the processor to leave the shutdown state. The invalid SMRAM state-save-area conditions that can cause a processor shutdown during an RSM are:

- CR0.PE=0 and CR0.PG=1.
- CR0.CD=0 and CR0.NW=1.
- Certain reserved bits are set to 1, including:
  - Any CR0 bit in the range 63:32 is set to 1.
  - Any unsupported bit in CR3 is set to 1.
  - Any unsupported bit in CR4 is set to 1.
  - Any DR6 bit or DR7 bit in the range 63:32 is set to 1.
  - Any unsupported bit in EFER is set to 1.
- Invalid returns to long mode, including:
  - EFER.LME=1, CR0.PG=1, and CR4.PAE=0.
  - EFER.LME=1, CR0.PG=1, CR4.PAE=1, CS.L=1, and CS.D=1.
- The SSM revision identifier is modified.

Some SMRAM state-save-area conditions are ignored, and the registers, or bits within the registers, are restored in a default manner by the processor. This avoids a processor shutdown when an invalid condition is stored in SMRAM. The default conditions restored by the processor are:

- The EFER.LMA register bit is set to the value obtained by logically ANDing the SMRAM values of EFER.LME, CR0.PG, and CR4.PAE.

- The RFLAGS.VM register bit is set to the value obtained by logically ANDing the SMRAM values of RFLAGS.VM, CR0.PE, and the inverse of EFER.LMA.
- The base values of FS, GS, GDTR, IDTR, LDTR, and TR are restored in canonical form. Those values are sign-extended to bit 63 using the most-significant implemented bit.
- Unimplemented segment-base bits in the CS, DS, ES, and SS registers are cleared to 0.
- SSP is canonicalized (i.e., sign-extended to bit 63).

10.5 Multiprocessor Considerations

For multiprocessor operation, each logical processor must be given a separate SMBASE value so that the save-state areas do not overlap. For systems with fewer than 64 logical processors it is sufficient to stagger the SMBASE values by 512 bytes. Note that this also offsets theSMI entry point by the same amount for each processor. With 64 or more logical processors, the entry points will start to collide with the save-state areas. Staggering the SMBASE values by 1024 bytes results in 512-byte entry point areas interleaved with the 512-byte state-save areas, and so provides scaling beyond 63 logical processors.

Further details on multiprocessor aspects of SMM may be found in the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* for a given processor family.

11 SSE, MMX, and x87 Programming

This chapter describes the system-software implications of supporting applications that use the Streaming SIMD Extensions (SSE), MMX™, and x87 instructions. Throughout this chapter, these instructions are collectively referred to as *media and x87* (media/x87) instructions. A complete listing of the instructions that fall in this category—and the detailed operation of each instruction—can be found in volumes 4 and 5. Refer to Volume 1 for information on using these instructions in application software.

The SSE instruction set is comprised of the *legacy SSE* instruction set which includes the SSE1, SSE2, SSE3, SSSE3, SSE4A, SSE4.1, and SSE4.2 subsets and the *extended SSE* instruction set which includes the AVX, FMA4, and XOP subsets. Many of the extended SSE instructions support both 128-bit and 256-bit data types.

11.1 Overview of System-Software Considerations

Processor implementations can support different combinations of the SSE, MMX, and x87 instruction sets. Two sets of registers—independent of the general-purpose registers—support these instructions. The SSE instructions operate on the YMM/XMM registers, and the 64-bit media and x87-instructions operate on the aliased MMX/x87 registers. The SSE and x87 floating-point instruction sets have distinct status registers, control registers, exception vectors, and system-software control bits for managing the operating environment. System software that supports use of these instructions must be able to manage these resources properly including:

- Detecting support for the instruction set, and enabling any optional features, as necessary.
- Saving and restoring the processor media or x87 state.
- Execution of floating-point instructions (media or x87) can produce exceptions. System software must supply exception handlers for all unmasked floating-point exceptions.

11.2 Determining Media and x87 Feature Support

Support for the architecturally defined subsets within the media and x87 instructions is implementation dependent. System software executes the CPUID instruction to determine whether a processor implements any of these features (see Section 3.3, “Processor Feature Identification,” on page 71 for more information on using the CPUID instruction). After CPUID is executed feature support can be determined by examining specific bit fields returned in the EAX, ECX, and EDX registers.

The following table summarizes the architecturally defined SSE subsets and state management instructions and gives the feature bits returned by the CPUID function. If the indicated bit is set, the feature is supported by the processor.

Table 11-1. SSE Subsets – CPUID Feature Identifiers

CPUID Fn	Field Name	Field Bit	Instruction Subset
Legacy SSE			
0000_0001h	EDX[SSE]	EDX[25]	Original Streaming SIMD Extensions (SSE1)
0000_0001h	EDX[SSE2]	EDX[26]	SSE2
0000_0001h	ECX[SSE3]	ECX[0]	SSE3
0000_0001h	ECX[SSSE3]	ECX[9]	SSSE3
0000_0001h	ECX[SSE41]	ECX[19]	SSE4.1
0000_0001h	ECX[SSE42]	ECX[20]	SSE4.2
8000_0001h	ECX[SSE4A]	ECX[6]	SSE4A: EXTRQ, INSERTQ, MOVNTSS, and MOVNTSD instructions
Extended SSE			
0000_0001h	ECX[AVX]	ECX[28]	AVX
8000_0001h	ECX[XOP]	ECX[11]	AMD XOP
0000_0001h	ECX[FMA]	ECX[12]	FMA
8000_0001h	ECX[FMA4]	ECX[16]	AMD FMA4
MMX			
0000_0001h or 8000_0001h	EDX[MMX]	EDX[23]	Original MMX™ Instructions
8000_0001h	EDX[MmxExt]	EDX[22]	AMD Extensions to MMX
8000_0001h	EDX[3DNow]	EDX[31]	AMD 3DNow!™
8000_0001h	EDX[3DNowExt]	EDX[30]	AMD Extensions to 3DNow!
x87			
0000_0001h or 8000_0001h	EDX[FPU]	EDX[0]	x87 instruction set and facilities
Context Management Instructions			
0000_0001h or 8000_0001h	EDX[FXSR]	EDX[24]	FXSAVE / FXRSTOR instructions
8000_0001h	EDX[FFXSR]	EDX[25]	Hardware optimizations for FXSAVE / FXRSTOR
0000_0001h	ECX[XSAVE]	ECX[26]	XSAVE / XRSTOR instructions
0000_000Dh ECX=01h	EAX[XSAVEOPT]	EAX[0]	XSAVEOPT

Some instructions may be listed in more than one subset. If software attempts to execute an instruction belonging to an unsupported instruction subset, an invalid-opcode exception (#UD) occurs. Refer to Appendix D, “Instruction Subsets and CPUID Feature Flags” in Volume 3 for specific information.

11.3 Enabling SSE Instructions

Use of the 256-bit and 128-bit media instructions by application software requires system software support. System software must determine which SSE subsets are supported, enable those that are to be used, and supply code to handle the various exceptions that may occur during the execution of these instructions. The legacy SSE instructions and the extended SSE instructions often require unique exception handling.

11.3.1 Enabling Legacy SSE Instruction Execution

When legacy SSE instructions are supported, system software must set CR4.OSFXSR to let the processor know that the software supports the FXSAVE/FXRSTOR instructions. When the processor detects CR4.OSFXSR = 1, it allows execution of the legacy SSE instructions. If system software does not set CR4.OSFXSR, any attempt to execute these instructions causes an invalid-opcode exception (#UD). System software must also *clear* the CR0.EM (emulate coprocessor) bit to 0, otherwise an attempt to execute a legacy SSE instruction causes a #UD exception. An attempt to execute either FXSAVE or FXRSTOR when CR0.EM is set results in a #NM exception.

System software should also *set* the CR0.MP (monitor coprocessor) bit to 1. When CR0.EM=0 and CR0.MP=1, all media instructions, x87 instructions, and the FWAIT/WAIT instructions cause a device-not-available exception (#NM) when the CR0.TS bit is set. System software can use the #NM exception to perform lazy context switching, saving and restoring media and x87 state only when necessary after a task switch. See “CR0 Register” on page 42 for more information.

11.3.2 Enabling Extended SSE Instruction Execution

After the steps specified above are completed to enable legacy SSE instruction execution, additional steps are required to enable the extended SSE instructions and state management. System software must carry out the following process:

- Confirm that the hardware supports the XSAVE, XRSTOR, XSETBV, and XGETBV instructions and the XCR0 register (XFEATURE_ENABLED_MASK) by executing the CPUID instruction function 0000_0001h. If CPUID Fn0000_0001_ECX[XSAVE] is set, hardware support is verified.
- Optionally confirm hardware support of the XSAVEOPT instruction by executing CPUID function 0000_000Dh, sub-function 1 (ECX = 1). If CPUID Fn0000_000D_EAX_x1[XSAVEOPT] is set, the processor supports the XSAVEOPT instruction. XSAVEOPT is a performance optimized version of XSAVE.
- Confirm that hardware supports the extended SSE instructions by verifying XFeatureSupportedMask[2:0] = 111b. XFeatureSupportedMask is accessed via the CPUID instruction function 0000_000Dh, sub-function 0 (ECX = 0). XFeatureSupportedMask[31:0] is returned in the EAX register.

  If CPUID Fn0000_000D_EAX_x0[2:0] = 111b, hardware supports x87, legacy SSE, and extended SSE instructions. Bit 0 of EAX signifies x87 floating-point and MMX support, bit 1 signifies legacy SSE support, and bit 2 signifies extended SSE support. Support for both x87 and legacy SSE instructions are required for processors that support the extended SSE instructions.

- Set CR4[OSXSAVE] (bit 18) to enable the use of the XSETBV and XGETBV instructions. XSETBV is a privileged instruction that writes the XCRn registers. XCR0 is the XFEATURE_ENABLED_MASK used to manage media and x87 processor state using the XSAVE, XSAVEOPT, and XRSTOR instructions.
- Enable the x87/MMX, legacy SSE, and extended SSE instructions and processor state management by setting the x87, SSE, and YMM bits of XCR0 (XFEATURE_ENABLED_MASK). This is done via the privileged instruction XSETBV. Enabling extended SSE capabilities without enabling legacy SSE capabilities is not allowed. The x87 flag (bit 0) of the XFEATURE_ENABLED_MASK must be set when writing XCR0.
- Determine the XSAVE/XRSTOR memory save area size requirement. The field XFeatureEnabledSizeMax specifies the size requirement in bytes based on the currently enabled extended features and is returned in the EBX register after execution of CPUID Function 0000_000Dh, sub-function 0 (ECX = 0).
- Allocate the save/restore area based on the information obtained in the previous step.

For a detailed description of the XSETBV and XGETBV instructions, see individual instruction reference pages in Volume 4. See the section entitled “XFEATURE_ENABLED_MASK” in Volume 4 for details on the field definitions for XFEATURE_ENABLED_MASK.

For more information on using the CPUID instruction to obtain processor feature information, see Section 3.3, “Processor Feature Identification,” on page 71.

11.3.3 SIMD Floating-Point Exception Handling

System software must supply an exception handler if unmasked SSE floating-point exceptions are allowed to occur. When an unmasked exception is detected, the processor transfers control to the SIMD floating-point exception (#XF) handler provided by the operating system. System software must let the processor know that the #XF handler is available by setting CR4.OSXMMEXCPT to 1. If this bit is set to 1, the processor transfers control to the #XF handler when it detects an unmasked exception, otherwise a #UD exception occurs. When the processor detects a masked exception, it handles it in a default manner regardless of the CR4.OSXMMEXCPT value.

11.4 Media and x87 Processor State

The media and x87 processor state includes the contents of the registers used by SSE, MMX, and x87 instructions. System software that supports such applications must be capable of saving and restoring these registers.

11.4.1 SSE Execution Unit State

Figure 11-1 shows the registers whose contents are affected by execution of SSE instructions. These include:

- YMM/XMM0–15—Sixteen 256-bit/128-bit SSE registers. In legacy and compatibility modes, software access is limited to the first eight registers.

- MXCSR—The 32-bit Media eXtensions Control and Status Register.

All of these registers are visible to application software. Refer to “Streaming SIMD Extensions Media and Scientific Programming” in Volume 1 for more information on these registers.

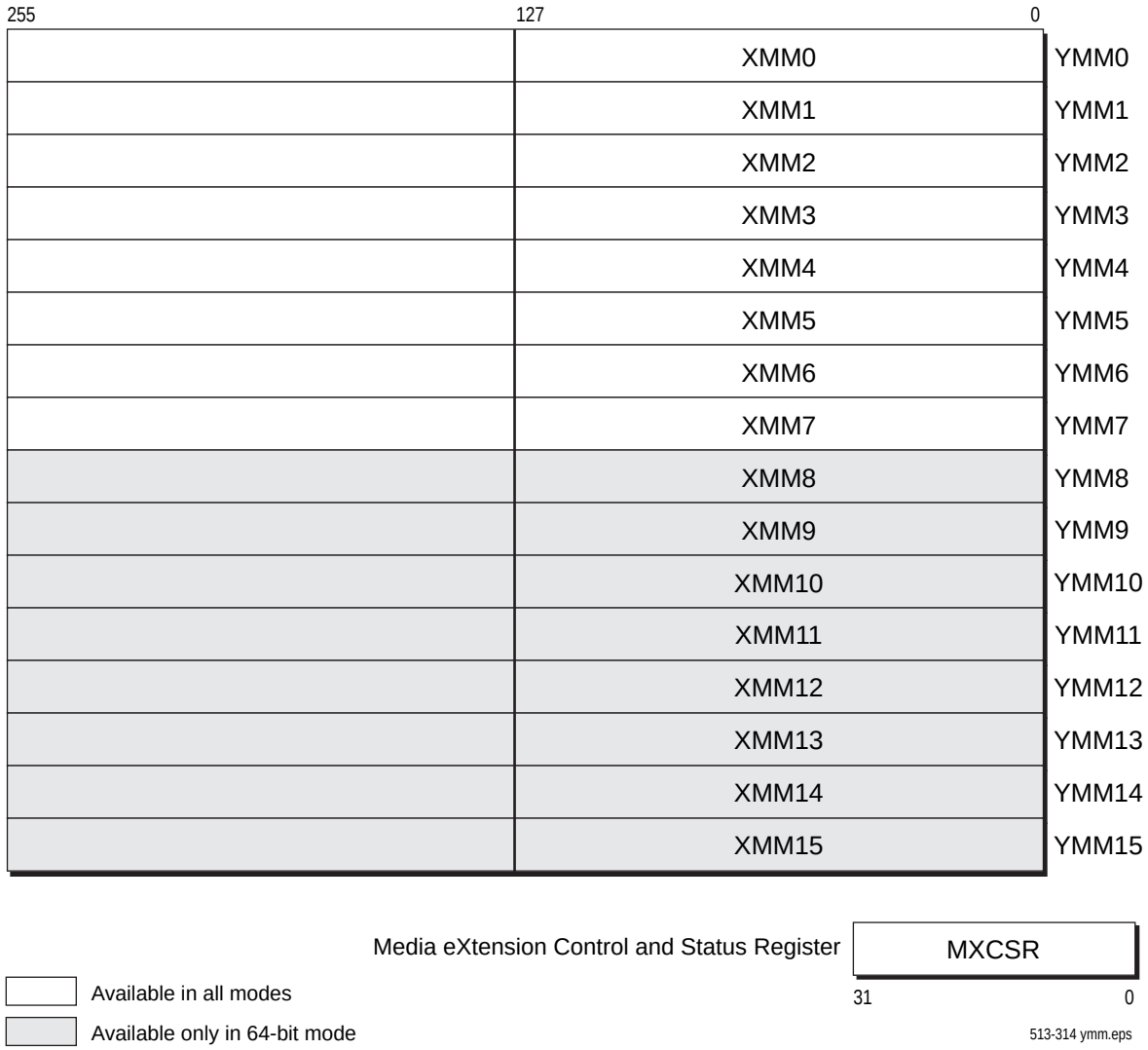


Figure 11-1. SSE Execution Unit State

11.4.2 MMX Execution Unit State

Figure 11-2 on page 340 shows the register contents that are affected by execution of 64-bit media instructions. These registers include:

- *mmx0–mmx7*—Eight 64-bit media registers.
- *FSW*—Two fields (TOP and ES) in the 16-bit x87 status word register.

- *FTW*—The 16-bit x87 tag word.

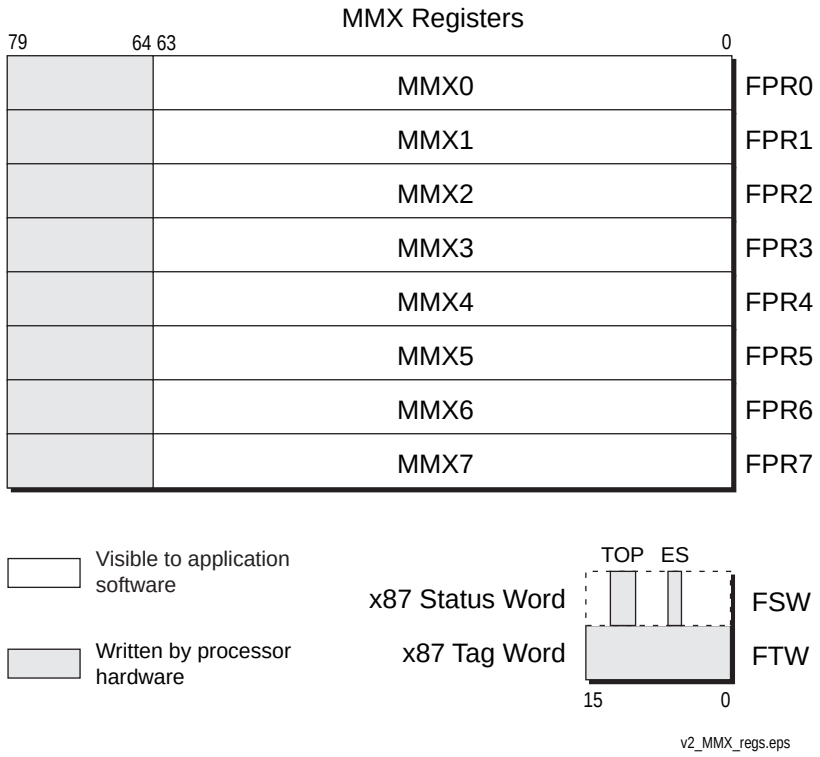


Figure 11-2. MMX Execution Unit State

The 64-bit media instructions and x87 floating-point instructions share the same physical data registers. Figure 11-2 shows how the 64-bit registers (MMX0–MMX7) are aliased onto the low 64 bits of the 80-bit x87 floating-point physical data registers (FPR0–FPR7). Refer to “64-Bit Media Programming” in Volume 1 for more information on these registers.

Of the registers shown in Figure 11-2, only the eight 64-bit MMX registers are visible to 64-bit media application software. The processor maintains the contents of the two fields of the x87 status word—top-of-stack-pointer (TOP) and exception summary (ES)—and the 16-bit x87 tag word during execution of 64-bit media instructions, as described in “Actions Taken on Executing 64-Bit Media Instructions” in Volume 1.

64-bit media instructions do not generate x87 floating-point exceptions, nor do they set any status flags. However, 64-bit media instructions can trigger an unmasked floating-point exception caused by a previously executed x87 instruction. 64-bit media instructions do this by reading the x87 FSW.ES bit to determine whether such an exception is pending.

11.4.3 x87 Execution Unit State

Figure 11-3 on page 342 shows the registers whose contents are affected by execution of x87 floating-point instructions. These registers include:

- *fpr0–fpr7*—Eight 80-bit floating-point physical registers.
- *FCW*—The 16-bit x87 control word register.
- *FSW*—The 16-bit x87 status word register.
- *FTW*—The 16-bit x87 tag word.
- *Last x87 Instruction Pointer*—This value is a pointer (32-bit, 48-bit, or 64-bit, depending on effective operand size and mode) to the last non-control x87 floating-point instruction executed.
- *Last x87 Data Pointer*—The pointer (32-bit, 48-bit, or 64-bit, depending on effective operand size and mode) to the data operand referenced by the last non-control x87 floating-point instruction executed, if that instruction referenced memory; if it did not, then this value is implementation dependent.
- *Last x87 Opcode*—An 11-bit permutation of the instruction opcode from the last non-control x87 floating-point instruction executed.

Of the registers shown in Figure 11-3 on page 342, only FPR0–FPR7, FCW, and FSW are directly updated by x87 application software. The processor maintains the contents of the FTW, instruction and data pointers, and opcode registers during execution of x87 instructions. Refer to “Registers” in Volume 1 for more information on these registers.

The 11-bit instruction opcode register holds a permutation of the two-byte instruction opcode from the last non-control x87 instruction executed by the processor. (For a definition of *non-control x87 instruction*, see “Control” in Chapter 6 of Volume 1.) The opcode field is formed as follows:

- Opcode Register Field[10:8] = First x87 opcode byte[2:0].
- Opcode Register Field[7:0] = Second x87 opcode byte[7:0].

For example, the x87 opcode D9 F8h is stored in the opcode register as 001_1111_1000b. The low-order three bits of the first opcode byte, D9h (1101_1001b), are stored in opcode-register bits 10:8. The second opcode byte, F8h (1111_1000b), is stored in bits 7:0 of the opcode register. The high-order five bits of the first opcode byte (1101_1b) are not needed because they are identical for all x87 instructions.

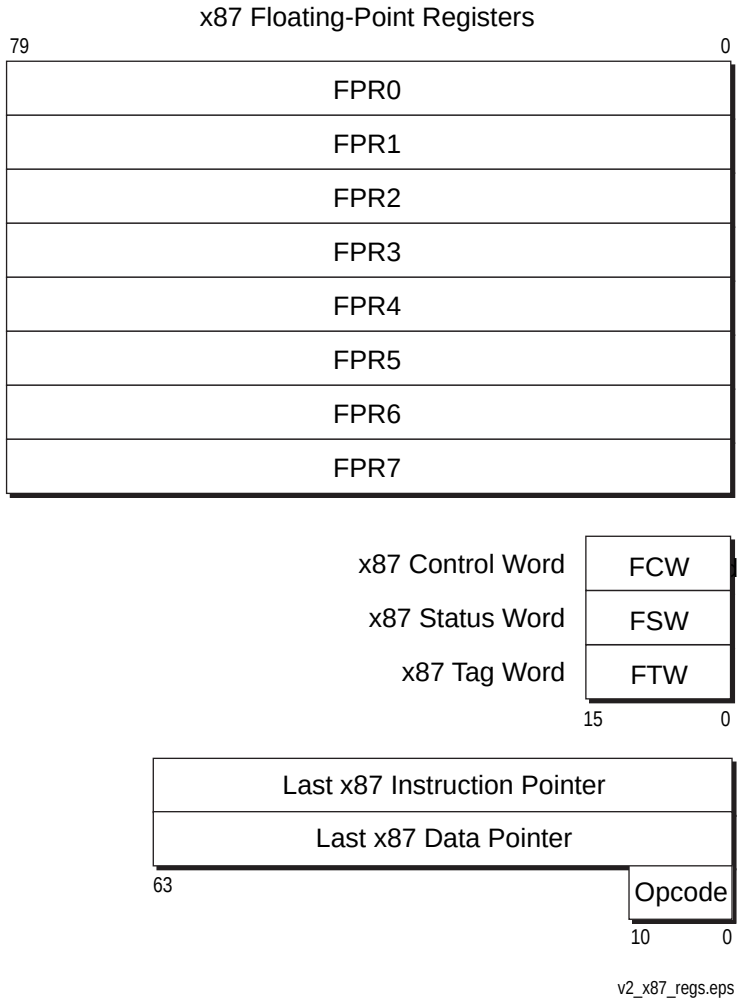


Figure 11-3. x87 Execution Unit State

11.4.4 Saving Media and x87 Execution Unit State

In most cases, operating systems, exception handlers, and device drivers should save and restore the media and/or x87 processor state between task switches or other interventions in the execution of 128-bit, 64-bit, or x87 procedures. Application programs are also free to save and restore state at any time.

In general, system software should use the FXSAVE and FXRSTOR instructions to save and restore the entire media and x87 processor state. The FSAVE/FNSAVE and FRSTOR instructions can be used for saving and restoring the x87 state. Because the 64-bit media registers are physically aliased onto the x87 registers, the FSAVE/FNSAVE and FRSTOR instructions can also be used to save and restore the 64-bit media state. However, FSAVE/FNSAVE and FRSTOR do not save or restore the 128-bit media state.

FSAVE/FNSAVE and FRSTOR Instructions. The FSAVE/FNSAVE and FRSTOR instructions save and restore the entire register state for 64-bit media instructions and x87 floating-point instructions. The FSAVE instruction stores the register state, but only after handling any pending unmasked-x87 floating-point exceptions. The FNSAVE instruction stores the register state but skips the reporting and handling of these exceptions. The state of all MMX/FPR registers is saved, as well as all other x87 state (the control word register, status word register, tag word, instruction pointer, data pointer, and last opcode). After saving this state, the tag state for all MMX/FPR registers is changed to *empty* and is thus available for a new procedure.

Starting on page 344, Figure 11-4 through Figure 11-7 show the memory formats used by the FSAVE/FNSAVE and FRSTOR instructions when storing the x87 state in various processor modes and using various effective-operand sizes. This state includes:

- *x87 Data Registers*
  - FPR0–FPR7 80-bit physical data registers.
- *x87 Environment*
  - FCW: x87 control word register
  - FSW: x87 status word register
  - FTW: x87 tag word
  - Last x87 instruction pointer
  - Last x87 data pointer
  - Last x87 opcode

The eight data registers are stored in the 80 bytes following the environment information. Instead of storing these registers in their physical order (FPR0–FPR7), the processor stores the registers in the their stack order, ST(0)–ST(7), beginning with the top-of-stack, ST(0).

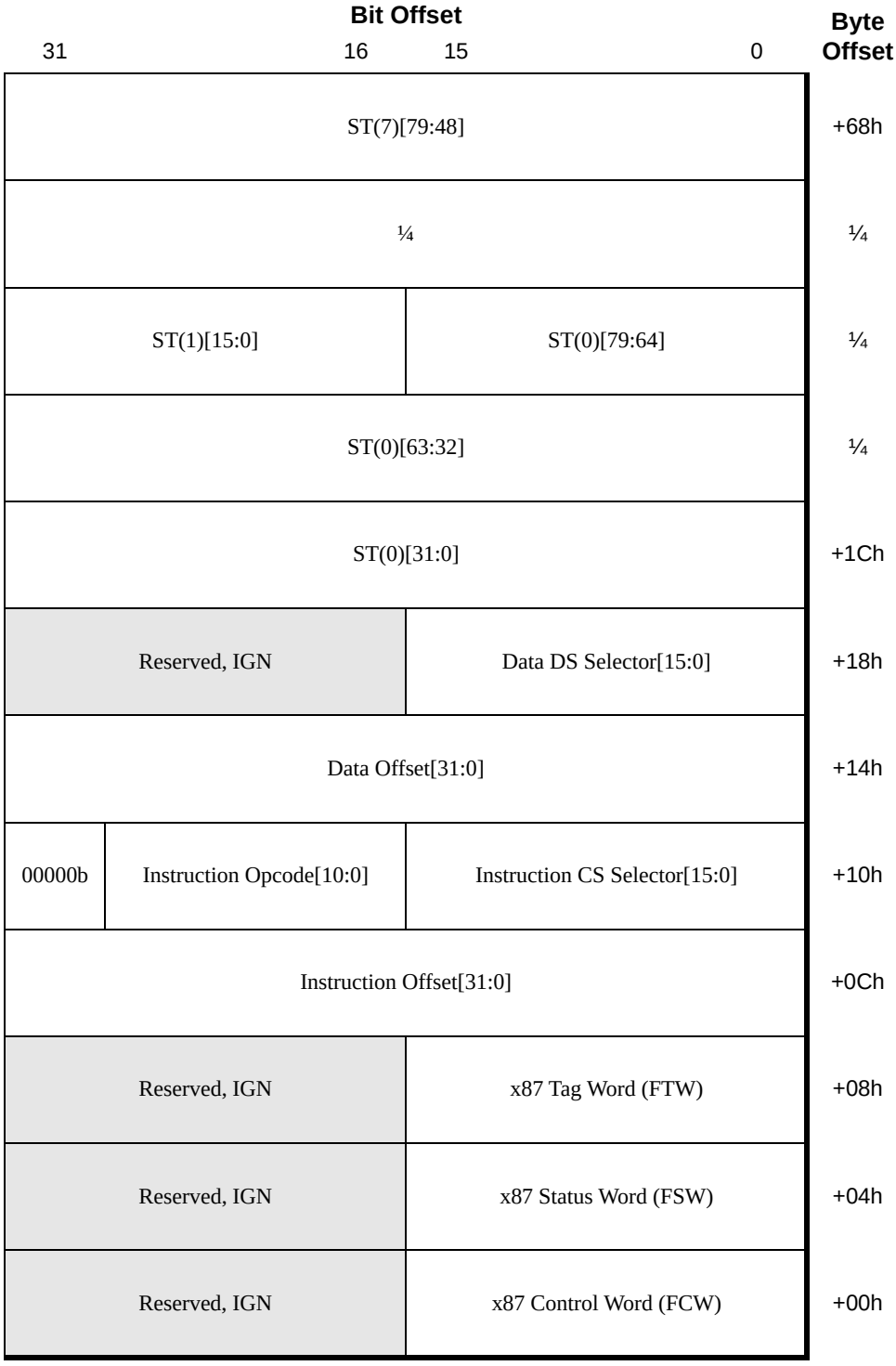


Figure 11-4. FSAVE/FNSAVE Image (32-Bit, Protected Mode)

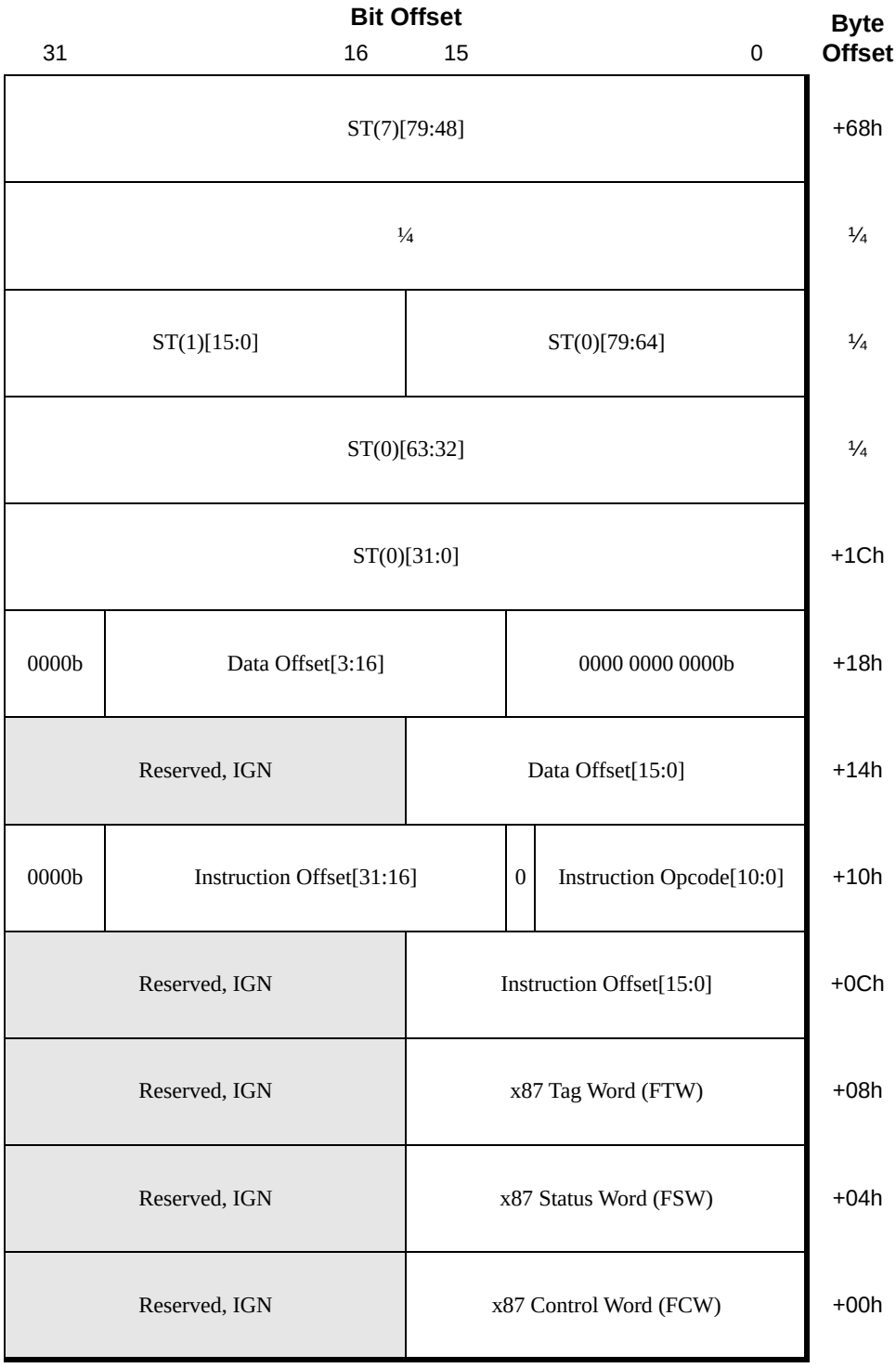


Figure 11-5. FSAVE/FNSAVE Image (32-Bit, Real/Virtual-8086 Modes)

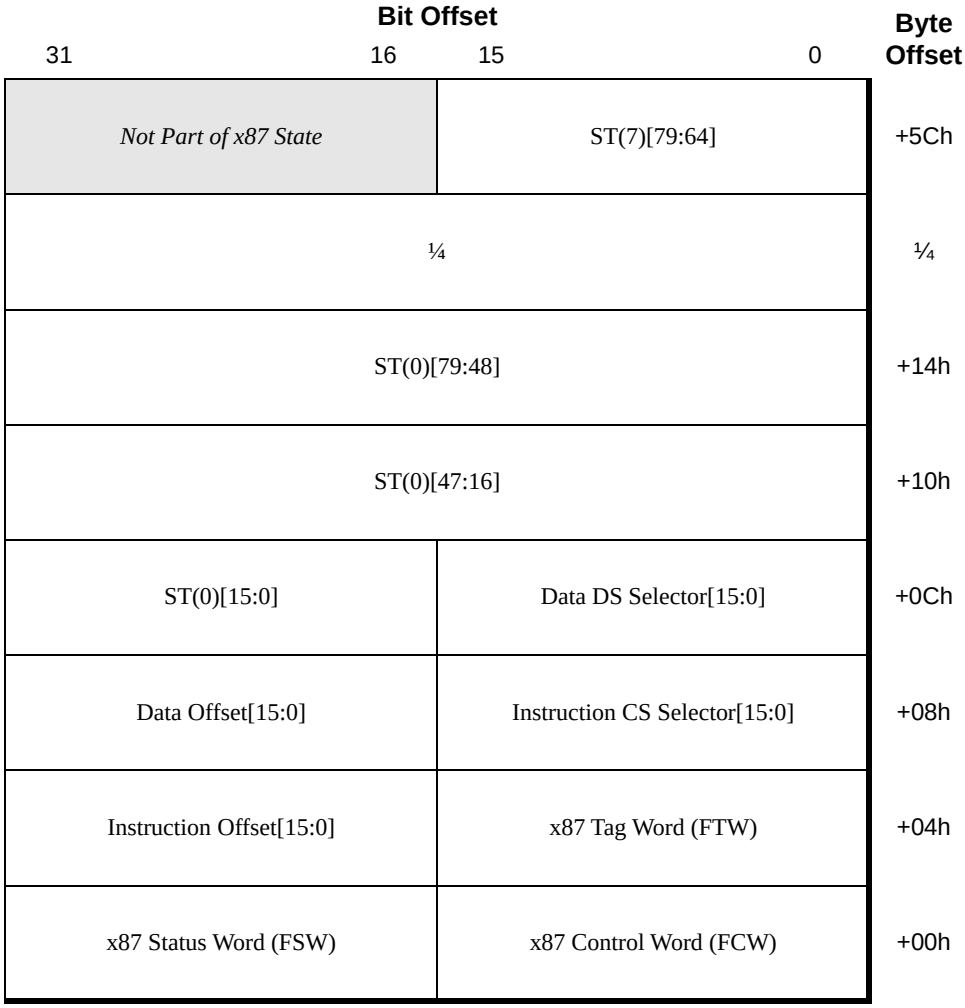


Figure 11-6. FSAVE/FNSAVE Image (16-Bit, Protected Mode)

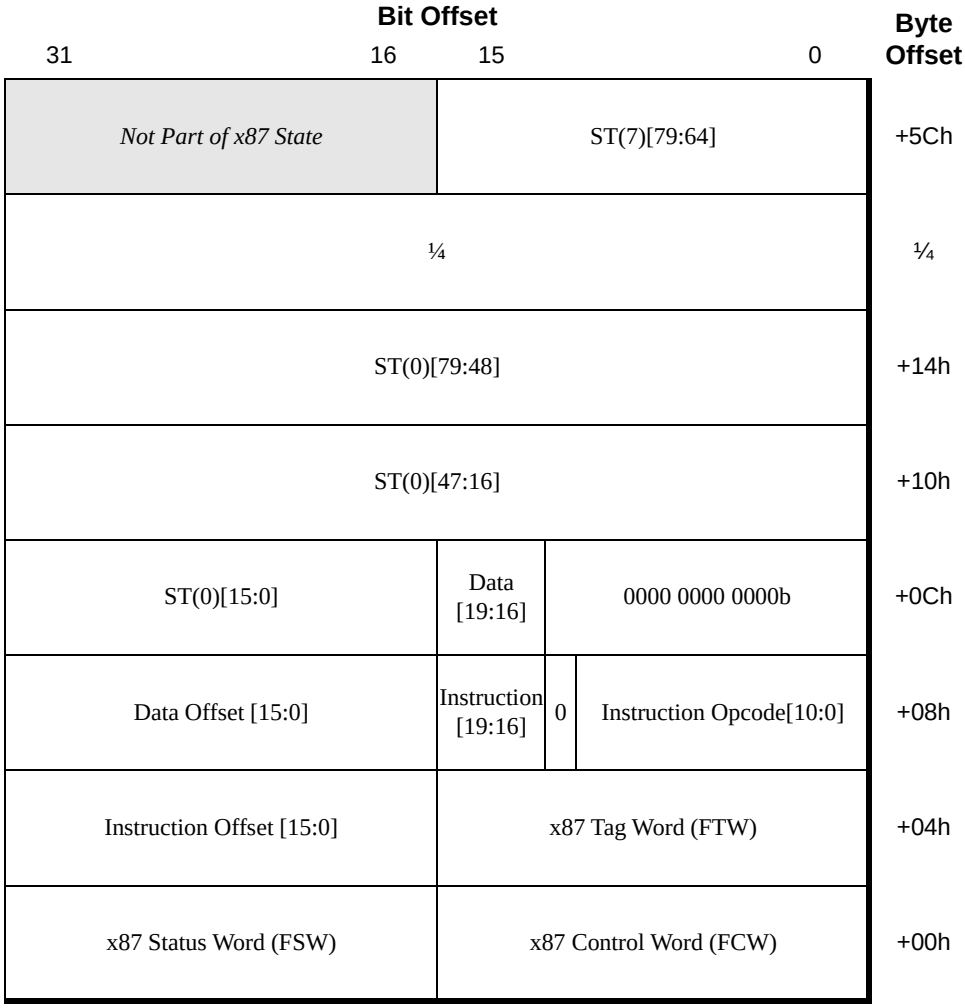


Figure 11-7. FSAVE/FNSAVE Image (16-Bit, Real/Virtual-8086 Modes)

FLDENV/FNLDENV and FSTENV Instructions. The FLDENV/FNLDENV and FSTENV instructions load and store only the x87 floating-point environment. These instructions, unlike the FSAVE/FNSAVE and FRSTOR instructions, do not save or restore the x87 data registers. The FLDENV/FSTENV instructions do not save the full 64-bit data and instruction pointers. 64-bit applications should use FXSAVE/FXRSTOR, rather than FLDENV/FSTENV. The format of the saved x87 environment images for protected mode and real/virtual mode are the same as those of the first 14-bytes of the FSAVE/FNSAVE images for 16-bit operands or 32/64-bit operands, respectively. See Figure 11-4 on page 344, Figure 11-5 on page 345, Figure 11-6 on page 346, and Figure 11-7.

FXSAVE and FXRSTOR Instructions. The FXSAVE and FXRSTOR instructions save and restore the entire 128-bit media, 64-bit media, and x87 state. These instructions usually execute faster than FSAVE/FNSAVE and FRSTOR because they do not normally save and restore the x87 exception pointers (last-instruction pointer, last data-operand pointer, and last opcode). The only case in which they do save the exception pointers is the relatively rare case in which the exception-summary bit in

the x87 status word (FSW.ES) is set to 1, indicating that an unmasked exception has occurred. The FXSAVE and FXRSTOR memory format contains fields for storing these values.

Unlike FSAVE and FNSAVE, the FXSAVE instruction does not alter the x87 tag word. Therefore, the contents of the shared 64-bit MMX and 80-bit FPR registers can remain valid after an FXSAVE instruction (or any other value the tag bits indicated before the save). Also, FXSAVE (like FNSAVE) does not check for pending unmasked-x87 floating-point exceptions.

Figure 11-8 on page 356 shows the memory format of the media x87 state in long mode. If a 32-bit operand size is used in 64-bit mode, the memory format is the same, except that RIP and RDS are stored as *sel:offset* pointers, as shown in Figure 11-9 on page 356.

For more information on the FXSAVE and FXRSTOR instructions, see individual instruction listings in "64-Bit Media Instruction Reference" of Volume 5.

11.5 XSAVE/XRSTOR Instructions

The XSAVE, XSAVEOPT, XRSTOR, XGETBV, and XSETBV instructions and associated data structures extend the FXSAVE/FXRSTOR memory image used to manage processor states and provide additional functionality. These instructions do not obviate the FXSAVE/FXRSTOR instructions. For more information about FXSAVE/FXRSTOR, see *"FXSAVE and FXRSTOR Instructions" in Volume 2*. For detailed descriptions of FXSAVE and FXRSTOR, see individual instruction listings in AMD64 Architecture Programmer's Manual *"Volume 5: 64-Bit Media and x87 Floating-Point Instructions."*

The CPUID instruction is used to identify features supported in processor hardware. Extended control registers are used to enable and disable the handling of processor states associated with supported hardware features and to communicate to an application whether an operating system supports a particular feature that has a processor state specific to it.

11.5.1 CPUID Enhancements

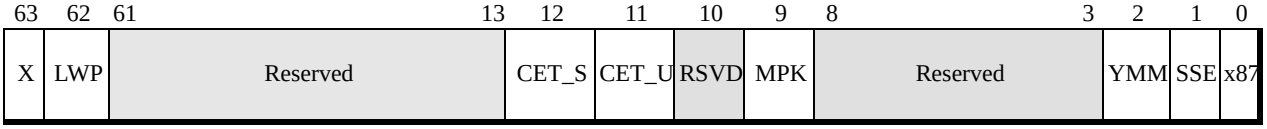
- CPUID Fn0000_00001_ECX[XSAVE] indicates that the processor supports XSAVE/XRSTOR instructions and at least one XCR.
- CPUID Fn0000_00001_ECX[OSXSAVE] indicates whether the operating system has enabled extensible state management and supports processor extended state management.
- CPUID Fn0000_0000D enumerates processor states (including legacy x87 FPU states, SSE states, and processor extended states), the offset, and the size of the save area for each processor extended state. Sub-functions (ECX > 0) provide details concerning features and support of processor states enumerated in the root function.

11.5.2 XFEATURE_ENABLED_MASK

XFEATURE_ENABLED_MASK is set up by privileged software to enable the saving and restoring of extended processor architectural state information supported by a specific processor. Clearing defined bit fields in this mask inhibits the XSAVE instruction from saving (and XRSTOR from restoring) this state information.

XFEATURE_ENABLED_MASK is addressed as XCR0 in the extended control register space and is accessed via the XSETBV and XGETBV instructions.

XFEATURE_ENABLED_MASK is defined as follows:



Bits	Mnemonic	Description
63	X	Reserved specifically for XCR0 bit vector expansion. Reserved, MBZ.
62	LWP	When set, Lightweight Profiling (LWP) extensions are enabled and XSAVE/XRSTOR supports LWP state management.
61:13	—	Reserved, MBZ
12	CET_S	When set, CET supervisor state is supported by XSAVE/XRSTOR
11	CET_U	When set, CET user state is supported by XSAVE/XRSTOR
10	—	Reserved, MBZ
9	MPK	When set, PKRU state management is supported by XSAVE/XRSTOR.
8:3	—	Reserved, MBZ
2	YMM	When set, 256-bit SSE state management is supported by XSAVE/XRSTOR. Must be set to enable AVX extensions.
1	SSE	When set, 128-bit SSE state management is supported by XSAVE/XRSTOR. This bit must be set if YMM is set. Must be set to enable AVX extensions.
0	x87	x87 FPU state management is supported by XSAVE/XRSTOR. Must be set to 1.

Hardware initializes XCR0 to 0000_0000_0000_0001h. On writing this register, software must insure that XCR0[63:3] is clear, XCR0[0] is set, and that XCR0[2:1] is not equal to 10b. An attempt to write data that violates these rules results in a #GP.

11.5.3 Extended Save Area

The XSAVE/XRSTOR save area extends the legacy 512-byte FXSAVE/FXRSTOR memory image to provide a compatible register state management environment as well as an upward migration path. The save area is architecturally defined to be extendable and enumerated by the sub-functions of CPUID Fn 0000_000Dh. Figure 11-2 shows the format of the XSAVE/XRSTOR area.

Table 11-2. Extended Save Area Format

Save Area	Offset (Byte)	Size (Bytes)
FPU/SSE Save Area	0	512
Header	512	64
Reserved, (Ext_Save_Area_2)	CPUID Fn 0000_000D_EBX_x02	CPUID Fn 0000_000D_EAX_x02
Reserved, (Ext_Save_Area_3)	CPUID Fn 0000_000D_EBX_x03	CPUID Fn 0000_000D_EAX_x03
Reserved, (Ext_Save_Area_4)	CPUID Fn 0000_000D_EBX_x04	CPUID Fn 0000_000D_EAX_x04
Reserved, (…)	…	…
Note: *Bytes 464–511 are available for software use. XRSTOR ignores bytes 464–511 of an XSAVE image.*		

The register fields of the first 512 bytes of the XSAVE/XRSTOR area are the same as those of the FXSAVE/FXRSTOR area, but the 512-byte area is organized as x87 FPU states, MXCSR (including MXCSR_MASK), and XMM registers. The layout of the save area is fixed and may contain non-contiguous individual save areas because a processor does not support certain extended states or

because system software does not support certain processor extended states. The save area is not compacted when features are not saved or are not supported by the processor or by system software.

For more information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 71.

11.5.4 Instruction Functions

CR4.OSXSAVE and XCR0 can be read at all privilege levels but written only at ring 0.

- XGETBV reads XCR0.
- XSETBV writes XCR0, ring 0 only.
- XRSTOR restores states specified by bitwise AND of a mask operand in EDX:EAX with XCR0.
- XSAVE (and XSAVEOPT) saves states specified by bitwise AND of a mask operand in EDX:EAX with XCR0.

11.5.5 YMM States and Supported Operating Modes

Extended instructions operate on YMM states by means of extended (XOP/VEX) prefix encoding. When a processor supports YMM states, the states exist in all operating modes, but interfaces to access the YMM states may vary by mode. Processor support for extended prefix encoding is independent of processor support of YMM states.

Instructions that use extended prefix encoding are generally supported in long and protected modes, but are not supported in real or virtual 8086 modes, or when entering SMM mode. Bits 255:128 of the YMM register state are maintained across transitions into and out of these modes. The XSAVE/XRSTOR instructions function in all operating modes; XRSTOR can modify YMM register state in any operating mode, using state information from the XSAVE/XRSTOR area.

11.5.6 Extended SSE Execution State Management

Operating system software must use the XSAVE/XRSTOR instructions for extended SSE execution state management. XSAVEOPT, a performance optimized version of XSAVE, may be used instead of XSAVE once the XSAVE/XRSTOR save area is initialized. In the following discussion XSAVEOPT may be substituted for the instruction XSAVE. The instructions also provide an interface to manage XMM/MXCSR states and x87 FPU states in conjunction with processor extended states. An operating system must enable extended SSE execution state management prior to the execution of extended SSE instructions. Attempting to execute an extended SSE instruction without enabling execution state management causes a #UD exception.

11.5.6.1 Enabling Extended SSE Instruction Execution

To enable extended SSE instruction execution and state management, system software must carry out the following process:

- Confirm that the hardware supports the XSAVE, XRSTOR, XSETBV, and XGETBV instructions and the XCR0 register (XFEATURE_ENABLED_MASK) by executing the CPUID instruction function 0000_0001h. If CPUID Fn0000_0001_ECX[XSAVE] is set, hardware support is verified.
- Optionally confirm hardware support of the XSAVEOPT instruction by executing CPUID function 0000_000Dh, sub-function 1 (ECX = 1). If CPUID Fn0000_000D_EAX_x1[XSAVEOPT] is set, the processor supports the XSAVEOPT instruction. XSAVEOPT is a performance optimized version of XSAVE. (SDCR-3580)
- Confirm that hardware supports the extended SSE instructions by verifying XFeatureSupportedMask[2:0] = 111b. XFeatureSupportedMask is accessed via the CPUID instruction function 0000_000Dh, sub-function 0 (ECX = 0).

  If CPUID Fn0000_000D_EAX_x0[2:0] = 111b, hardware supports x87, legacy SSE, and extended SSE instructions. Bit 0 of EAX signifies x87 floating-point and MMX support, bit 1 signifies legacy SSE support, and bit 2 signifies extended SSE support. Support for both x87 and legacy SSE instructions are required for processors that support the extended SSE instructions.
- Set CR4[OSXSAVE] (bit 18) to enable the use of the XSETBV and XGETBV instructions. XSETBV is a privileged instruction that writes the XCRn registers. XCR0 is the XFEATURE_ENABLED_MASK used to manage media and x87 processor state using the XSAVE, XSAVEOPT, and XRSTOR instructions.
- Enable the x87/MMX, legacy SSE, and extended SSE instructions and processor state management by setting the x87, SSE, and YMM bits of XCR0 (XFEATURE_ENABLED_MASK). Enabling extended SSE capabilities without enabling legacy SSE capabilities is not allowed. The x87 flag (bit 0) of the XFEATURE_ENABLED_MASK must be set when writing XCR0.
- Determine the XSAVE/XRSTOR memory save area size requirement. The field XFeatureEnabledSizeMax specifies the size requirement in bytes based on the currently enabled extended features and is returned in the EAX register after execution of CPUID Function 0000_000Dh, sub-function 0 (ECX = 0).
- Allocate the save/restore area based on the information obtained in the previous step.

For more information on the XSETBV and XGETBV instructions, see individual instruction descriptions in Volume 4. XFEATURE_ENABLED_MASK fields are defined in Section 11.5.2 above.

For more information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 71.

11.5.7 Saving Processor State

The XSTATE header starts at byte offset 512 in the save area. XSTATE_BV is the first 64-bit field in the header. The order of bit vectors in XSTATE_BV matches the order of bit vectors in XCR0. The XSAVE instruction sets bits in the XSTATE_BV vector field when it writes the corresponding processor extended state to a save area in memory. XSAVE modifies only bits for processor states specified by bitwise AND of the XSAVE bit mask operand in EDX:EAX with XCR0. If software modifies the save area image of a particular processor state component directly, it must also set the corresponding bit of XSTATE_BV. If the bit is not set, directly modified state information in a save area image may be ignored by XRSTOR.

XSAVEOPT, a performance optimized version of the XSAVE instruction, may be used (if supported) in lieu of the XSAVE instruction once the XSAVE/XRSTOR save area has been initialized via the execution of the XSAVE instruction.

11.5.8 Restoring Processor State

When XRSTOR is executed, processor state components are updated only if the corresponding bits in the mask operand (EDX:EAX) and XCR0 are both set. For each updated component, when the corresponding bit in the XSTATE_BV field in the save area header is set, the component is loaded from the save area in memory. When the XSTATE_BV bit is cleared, the state is set to the hardware-specified initial values shown in Table 11-3.

Table 11-3. XRSTOR Hardware-Specified Initial Values

Component	Initial Value
x87	FCW = 037Fh FSW = 0000h Empty/Full = 00h (FTW = FFFFh) x87 Error Pointers = 0 ST0 - ST7 = 0
XMM	XMM0 - XMM15 = 0, if 64-bit mode XMM0 - XMM7 = 0, if !64-bit mode
YMM_HI	YMM_HI0 -Y MM_HI15 = 0, if 64-bit mode YMM_HI0-YMM_HI7 = 0, if !64-bit mode
LWP	LWP disabled
MPK	PKRU = 0
CET_S	PL0_SSP = PL1_SSP = PL2_SSP = 0
CET_U	U_CET = 0, PL3_SSP = 0

11.5.9 MXCSR State Management

The MXCSR has no hardware-specified initial state; it is read from the save area in memory whenever either XMM or YMM_HI are updated.

11.5.10 Mode-Specific XSAVE/XRSTOR State Management

Some state is conditionally saved or updated, depending on processor state:

- On processors where CPUID Fn8000_0008_EBX[2] is 0, the x87 error pointers are not saved or restored if the state saved or loaded from memory doesn't have a pending #MF. On processors where CPUID Fn8000_0008_EBX[2] is 1, the error pointers are always restored from the save area (and if in 64-bit mode the CS and DS portions of the error pointer registers are zeroed), and the error pointer fields in the save area are zeroed if there is no pending #MF, else the error pointer offset registers are written to the save area.
- XMM8–XMM15 are not saved or restored in non 64-bit mode.
- YMM_HI8–YMM_HI15 are not saved or restored in non 64-bit mode.

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
Reserved, IGN ¼ Reserved, IGN																+1F0h … +1A0h
XMM15																+190h
XMM14																+180h
XMM13																+170h
XMM12																+160h
XMM11																+150h
XMM10																+140h
XMM9																+130h
XMM8																+120h
XMM7																+110h
XMM6																+100h
XMM5																+F0h
XMM4																+E0h
XMM3																+D0h
XMM2																+C0h
XMM1																+B0h
XMM0																+A0h
Reserved, IGN																+90h
Reserved, IGN																+80h
Reserved, IGN																+70h
Reserved, IGN																+60h
Reserved, IGN																+50h
Reserved, IGN																+40h
Reserved, IGN																+30h
Reserved, IGN																+20h
MXCSR_MASK								MXCSR								+10h
RIP[1]										FOP		0	FTW	FSW	FCW	+00h

1. Stored as sel:offset if operand size is 32 bits. 32bit sel:offset format of the pointers is shown in figure 11-9.

Figure 11-8. FXSAVE and FXRSTOR Image (64-bit Mode)

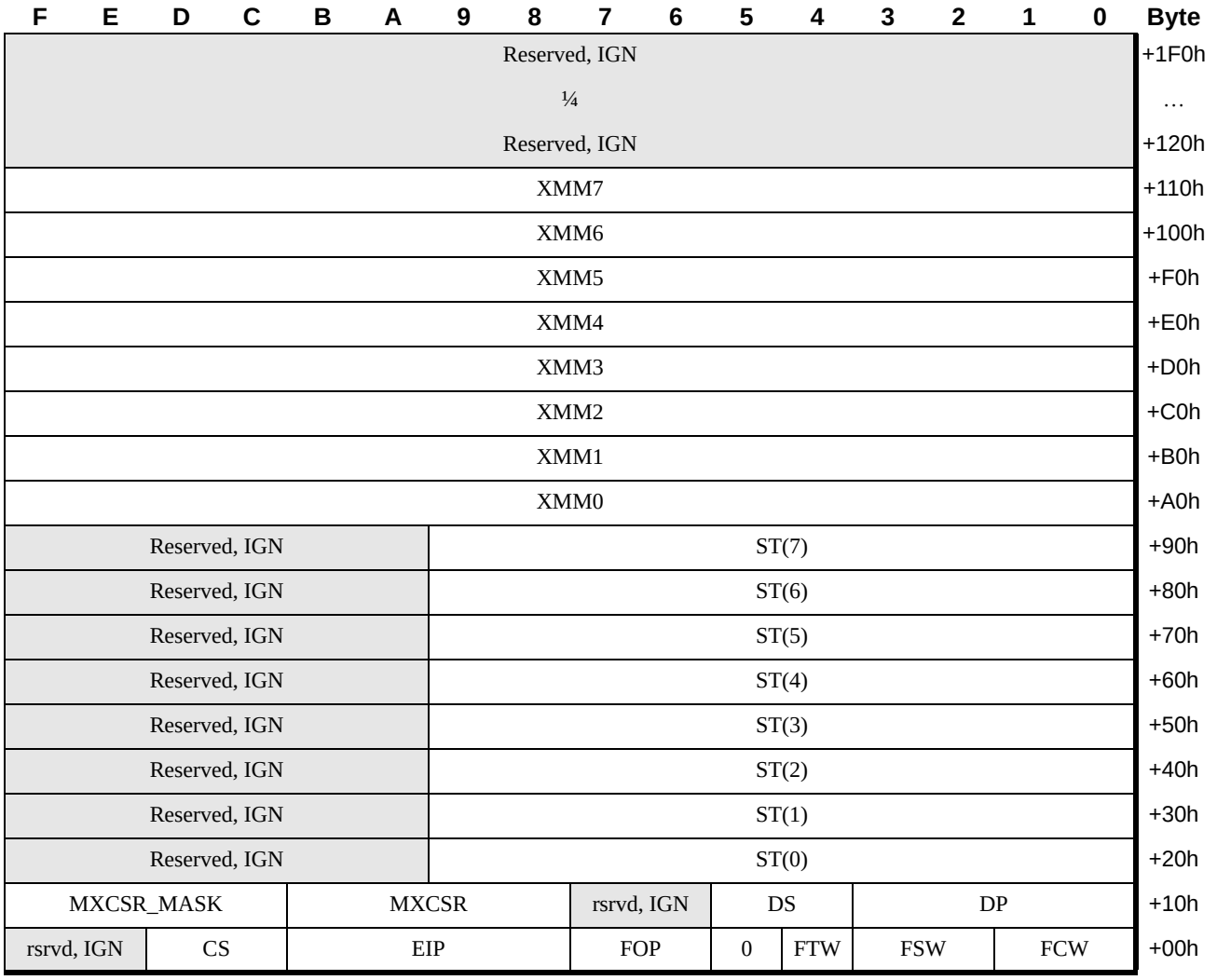


Figure 11-9. FXSAVE and FXRSTOR Image (Non-64-bit Mode)

Software can read and write all fields within the FXSAVE and FXRSTOR memory image. These fields include:

- *FCW*—Bytes 01h–00h. x87 control word.
- *FSW*—Bytes 03h–02h. x87 status word.
- *FTW*—Byte 04h. x87 tag word. See “FXSAVE Format for x87 Tag Word” on page 357 for additional information on the FTW format saved by the FXSAVE instruction.
- (Byte 05h contains the value 00h.)
- *FOP*—Bytes 07h–06h. last x87 opcode.

- *Last x87 Instruction Pointer*—A pointer to the last non-control x87 floating-point instruction executed by the processor:
  - *RIP (64-bit format)*—Bytes 0Fh–08h. 64-bit offset into the code segment (used without a CS selector).
  - *EIP (32-bit format)*—Bytes 0Bh–08h. 32-bit offset into the code segment.
  - *CS (32-bit format)*—Bytes 0Dh–0Ch. Segment selector portion of the pointer.
- *Last x87 Data Pointer*—If the last non-control x87 floating point instruction referenced memory, this value is a pointer to the data operand referenced by the last non-control x87 floating-point instruction executed by the processor:
  - *RDP (64-bit format)*—Bytes 17h–10h. 64-bit offset into the data segment (used without a DS selector).
  - *DP (32-bit format)*—Bytes 13h–10h. 32-bit offset into the data segment.
  - *DS (32-bit format)*—Bytes 15h–14h. Segment selector portion of the pointer.

  If the last non-control x87 instruction did not reference memory, then the value in the pointer is implementation dependent.
- *MXCSR*—Bytes 1Bh–18h. 128-bit media-instruction control and status register. This register is saved only if CR4.OSFXSR is set to 1.
- *MXCSR_MASK*—Bytes 1Fh–1Ch. Set bits in MXCSR_MASK indicate supported feature bits in MXCSR. For example, if bit 6 (the DAZ bit) in the returned MXCSR_MASK field is set to 1, the DAZ mode and the DAZ flag in MXCSR *are* supported. Cleared bits in MXCSR_MASK indicate reserved bits in MXCSR. If software attempts to set a reserved bit in the MXCSR register, a #GP exception will occur. To avoid this exception, after software clears the FXSAVE memory image and executes the FXSAVE instruction, software should use the value returned by the processor in the MXCSR_MASK field when writing a value to the MXCSR register, as follows:
  - *MXCSR_MASK = 0:* If the processor writes a zero value into the MXCSR_MASK field, the denormals-are-zeros (DAZ) mode and the DAZ flag in MXCSR are *not* supported. Software should use the default mask value, 0000_FFBFh (bit 6, the DAZ bit, and bits 31:16 cleared to 0), to mask any value it writes to the MXCSR register to ensure that all reserved bits in MXCSR are written with 0, thus avoiding a #GP exception.
  - *MXCSR_MASK … 0:* If the processor writes a non-zero value into the MXCSR_MASK field, software should AND this value with any value it writes to the MXCSR register.
- *MMXn/FPRn*—Bytes 9Fh–20h. Shared 64-bit media and x87 floating-point registers. As in the case of the x87 FSAVE instruction, these registers are stored in stack order ST(0)–ST(7). The upper six bytes in the memory image for each register are reserved.
- *XMMn*—Bytes 11Fh–A0h. 128-bit media registers. These registers are saved only if CR4.OSFXSR is set to 1.

FXSAVE Format for x87 Tag Word. Rather than saving the entire x87 tag word, FXSAVE saves a single-byte encoded version. FXSAVE encodes each of the eight two-bit fields in the x87 tag word as follows:

- Two-bit values of 00, 01, and 10 are encoded as a 1, indicating the corresponding x87 FPR*n* register holds a value.
- A two-bit value of 11 is encoded as a 0, indicating the corresponding x87 FPR*n* is empty.

For example, assume an FSAVE instruction saves an x87 tag word with the value 83F1h. This tag-word value describes the x87 FPRn contents as follows:

x87 Register	FPR7	FPR6	FPR5	FPR4	FPR3	FPR2	FPR1	FPR0
Tag Word Value (hex)	8		3		F		1	
Tag Value (binary)	10	00	00	11	11	11	00	01
Meaning	Special	Valid	Valid	Empty	Empty	Empty	Valid	Zero

When an FXSAVE is used to write the x87 tag word to memory, it encodes the value as E3h. This encoded version describes the x87 FPRn contents as follows:

x87 Register	FPR7	FPR6	FPR5	FPR4	FPR3	FPR2	FPR1	FPR0
Encoded Tag Byte (hex)	E				3			
Tag Value (binary)	1	1	1	0	0	0	1	1
Meaning	Valid	Valid	Valid	Empty	Empty	Empty	Valid	Valid

If necessary, software can decode the single-bit FXSAVE tag-word fields into the two-bit field FSAVE uses by examining the contents of the corresponding FPR registers saved by FXSAVE. Table 11-4 on page 359 shows how the FPR contents are used to find the equivalent FSAVE tag-field value. The *fraction* column refers to fraction portion of the extended-precision significand (bits 62:0). The *integer bit* column refers to the integer-portion of the significand (bit 63). See Chapter 11, “SSE, MMX, and x87 Programming,” on page 335 for more information on floating-point numbering formats.

Table 11-4. Deriving FSAVE Tag Field from FXSAVE Tag Field

Encoded FXSAVE Tag Field	Exponent	Integer Bit[2]	Fraction[1]	Type of Value	Equivalent FSAVE Tag Field
1 (Valid)	All 0s	0	All 0s	Zero	01 (Zero)
		0	Not all 0s	Denormal	10 (Special)
		1	All 0s	Pseudo Denormal	
		1	Not all 0s		
	Neither all 0s nor all 1s	0	don't care	Unnormal	
		1		Normal	00 (Valid)
	All 1s	0		Pseudo Infinity or Pseudo NaN	10 (Special)
		1	All 0s	Infinity	
			Not all 0s	NaN	
0 (Empty)	don't care			Empty	11 (Empty)

Note:
1. *Bits 62:0 of the significand. Bit 62, the most-significant bit of the fraction, is also called the M bit.*
2. *Bit 63 of the significand, also called the J bit.*

Performance Considerations. When system software supports multi-tasking, it must be able to save the processor state for one task and load the state for another. For performance reasons, the media and/or x87 processor state is usually saved and loaded only when necessary. System software can save and load this state at the time a task switch occurs. However, if the new task does not use the state, loading the state is unnecessary and reduces performance.

The task-switch bit (CR0.TS) is provided as a *lazy* context-switch mechanism that allows system software to save and load the processor state only when necessary. When CR0.TS=1, a device-not-available exception (#NM) occurs when an attempt is made to execute a 128-bit media, 64-bit media, or x87 instruction. System software can use the #NM exception handler to save the state of the previous task, and restore the state of the current task. Before returning from the exception handler to the media or x87 instruction, system software must clear CR0.TS to 0 to allow the instruction to be executed. Using this approach, the processor state is saved only when the registers are used.

In legacy mode, the hardware task-switch mechanism sets CR0.TS=1 during a task switch (see "Task Switched (TS) Bit" on page 44 for more information). In long mode, the hardware task-switching is not supported, and the CR0.TS bit is not set by the processor. Instead, the architecture assumes that system software handles all task-switching and state-saving functions. If CR0.TS is to be used in long mode for controlling the save and restore of media or x87 state, system software must set and clear it explicitly.

12 Task Management

This chapter describes the hardware task-management features. All of the legacy x86 task-management features are supported by the AMD64 architecture in legacy mode, but most features are not available in long mode. Long mode, however, requires system software to initialize and maintain certain task-management *resources*. The details of these resource-initialization requirements for long mode are discussed in “Task-Management Resources” on page 362.

12.1 Hardware Multitasking Overview

A task (also called a *process*) is a program that the processor can execute, suspend, and later resume executing at the point of suspension. During the time a task is suspended, other tasks are allowed to execute. Each task has its own execution space, consisting of:

- Code segment and instruction pointer.
- Data segments.
- Stack segments for each privilege level.
- General-purpose registers.
- rFLAGS register.
- Local-descriptor table.
- Task register, and a link to the previously-executed task.
- I/O-permission and interrupt-permission bitmaps.
- Pointer to the page-translation tables (CR3).

The state information defining this execution space is stored in the task-state segment (TSS) maintained for each task.

Support for hardware multitasking is provided in legacy mode. Hardware multitasking provides automated mechanisms for switching tasks, saving the execution state of the suspended task, and restoring the execution state of the resumed task. When hardware multitasking is used to switch tasks, the processor takes the following actions:

- Suspends execution of the task, allowing any executing instructions to complete and save their results.
- Saves the task execution state in the task TSS.
- Loads the execution state for the new task from its TSS.
- Begins executing the new task at the location specified in the new task TSS.

Software can switch tasks by branching to a new task using the CALL or JMP instructions. Exceptions and interrupts can also switch tasks if the exception or interrupt handlers are themselves separate tasks. IRET can be used to return to an earlier task.

12.2 Task-Management Resources

The hardware-multitasking features are available when protected mode is enabled (CR0.PE=1). Protected-mode software execution, by definition, occurs as part of a task. While system software is not required to use the hardware-multitasking features, it is required to initialize certain task-management resources for at least one task (the current task) when running in protected mode. This single task is needed to establish the protected-mode execution environment. The resources that must be initialized are:

- *Task-State Segment (TSS)*—A segment that holds the processor state associated with a task.
- *TSS Descriptor*—A segment descriptor that defines the task-state segment.
- *TSS Selector*—A segment selector that references the TSS descriptor located in the GDT.
- *Task Register*—A register that holds the TSS selector and TSS descriptor for the current task.

Figure 12-1 on page 363 shows the relationship of these resources to each other in both 64-bit and 32-bit operating environments.

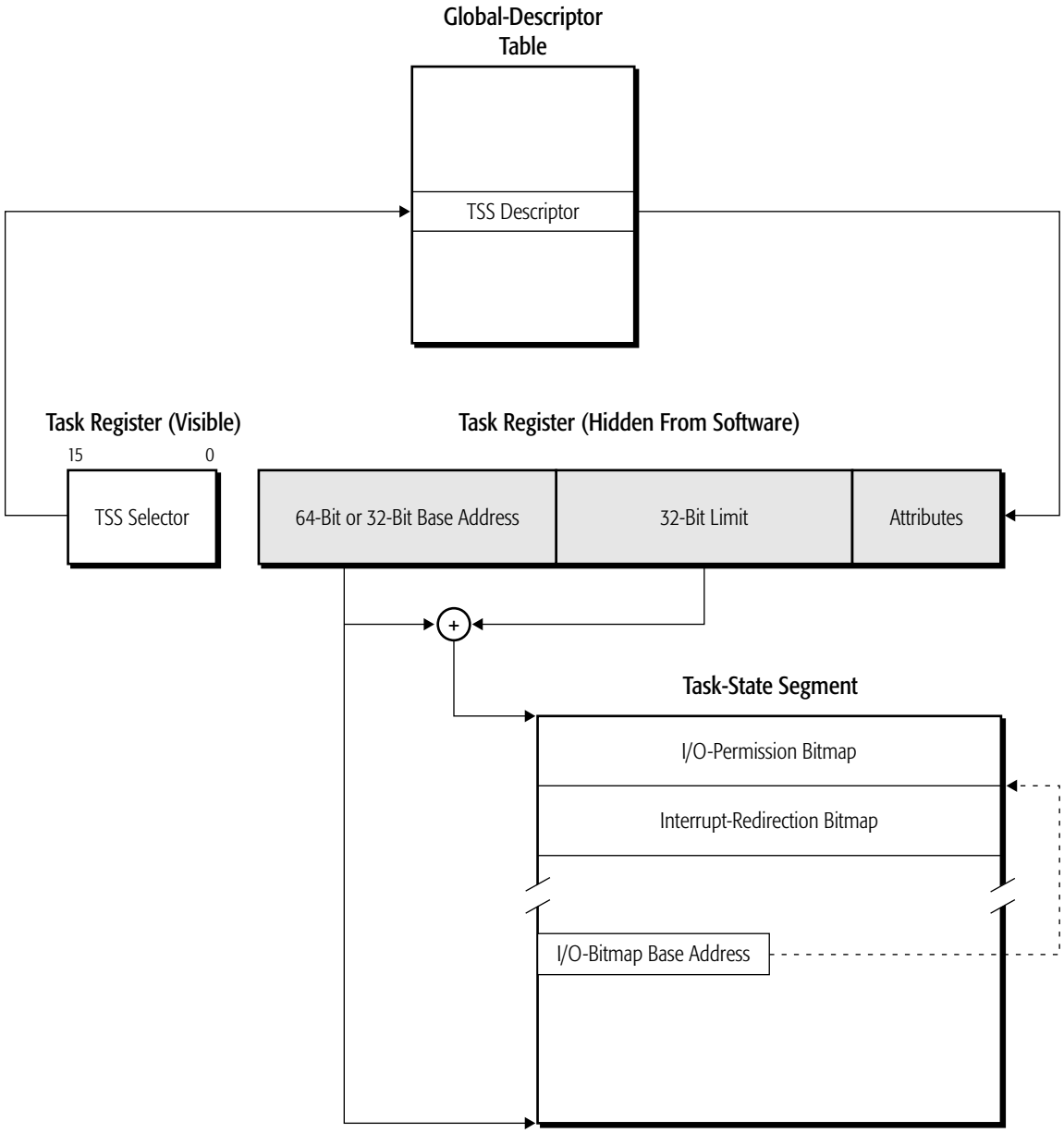


Figure 12-1. Task-Management Resources

A fifth resource is available in legacy mode for use by system software that uses the hardware-multitasking mechanism to manage more than one task:

- *Task-Gate Descriptor*—This form of gate descriptor holds a reference to a TSS descriptor and is used to control access between tasks.

The task-management resources are described in the following sections.

12.2.1 TSS Selector

TSS selectors are selectors that point to task-state segment descriptors in the GDT. Their format is identical to all other segment selectors, as shown in Figure 12-2.

Bits	Mnemonic	Description
15:3		Selector Index
2	TI	Table Indicator
1:0	RPL	Requestor Privilege Level

Figure 12-2. Task-Segment Selector

The selector format consists of the following fields:

Selector Index. Bits 15:3. The selector-index field locates the TSS descriptor in the global-descriptor table.

Table Indicator (TI) Bit. Bit 2. The TI bit must be cleared to 0, which indicates that the GDT is used. TSS descriptors cannot be located in the LDT. If a reference is made to a TSS descriptor in the LDT, a general-protection exception (#GP) occurs.

Requestor Privilege-Level (RPL) Field. Bits 1:0. RPL represents the privilege level (CPL) the processor is operating under at the time the TSS selector is loaded into the task register.

12.2.2 TSS Descriptor

The TSS descriptor is a system-segment descriptor, and it can be located only in the GDT. The format for an 8-byte, legacy-mode and compatibility-mode TSS descriptor can be found in “System Descriptors” on page 94. The format for a 16-byte, 64-bit mode TSS descriptor can be found in “System Descriptors” on page 99.

The fields within a TSS descriptor (all modes) are described in “Descriptor Format” on page 88. The following additional information applies to TSS descriptors:

- *Segment Limit*—When shadow stacks are not enabled (CR4.CET=0), a TSS descriptor must have a segment limit value of at least 67h, which defines a minimum TSS size of 68h (104 decimal) bytes. If shadow stacks are enabled (CR4.CET=1), the TSS segment limit must be at least 06Bh (for a minimum size of 108 decimal bytes), in order to accommodate the 32-bit shadow stack pointer (SSP). If the limit is less than the specified values, an invalid-TSS exception (#TS) occurs during the task switch. When an I/O-permission bitmap, interrupt-redirection bitmap, or additional state

information is included in the TSS, the limit must be set to a value large enough to enclose that information. In this case, if the TSS limit is not large enough to hold the additional information, a #GP exception occurs when an attempt is made to access beyond the TSS limit. No check for the larger limit is performed during the task switch.

- *Type*—Four system-descriptor types are defined as TSS types, as shown in Table 4-5 on page 94. Bit 9 is used as the descriptor busy bit (B). This bit indicates that the task is busy when set to 1, and available when cleared to 0. Busy tasks are the currently running task and any previous (outer) tasks in a nested-task hierarchy. Task recursion is not supported, and a #GP exception occurs if an attempt is made to transfer control to a busy task. See "Nesting Tasks" on page 382 for additional information.

  In long mode, the 32-bit TSS types (available and busy) are redefined as 64-bit TSS types, and only 64-bit TSS descriptors can be used. Loading the task register with an available 64-bit TSS causes the processor to change the TSS descriptor type to indicate a busy 64-bit TSS. Because long mode does not support task switching, the TSS-descriptor busy bit is never cleared by the processor to indicate an available 64-bit TSS.

  Sixteen-bit TSS types are illegal in long mode. A general-protection exception (#GP) occurs if a reference is made to a 16-bit TSS.

12.2.3 Task Register

The *task register* (TR) points to the TSS location in memory, defines its size, and specifies its attributes. As with the other descriptor-table registers, the TR has two portions. A *visible* portion holds the TSS selector, and a *hidden* portion holds the TSS descriptor. When the TSS selector is loaded into the TR, the processor automatically loads the TSS descriptor from the GDT into the hidden portion of the TR.

The TR is loaded with a new selector using the LTR instruction. The TR is also loaded during a task switch, as described in "Switching Tasks" on page 375.

Figure 12-3 shows the format of the TR in legacy mode.

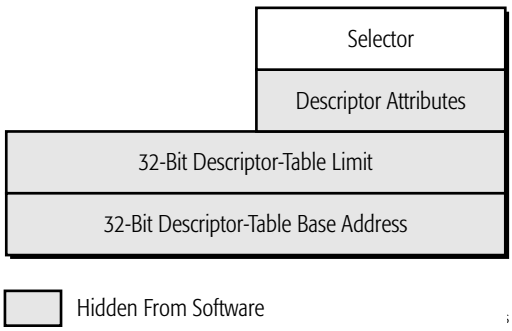


Figure 12-3. TR Format, Legacy Mode

Figure 12-4 shows the format of the TR in long mode (both compatibility mode and 64-bit mode).

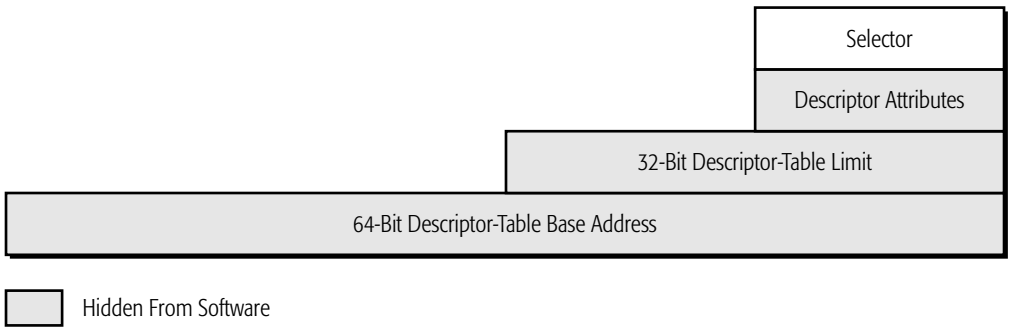


Figure 12-4. TR Format, Long Mode

The AMD64 architecture expands the TSS-descriptor base-address field to 64 bits so that system software running in long mode can access a TSS located anywhere in the 64-bit virtual-address space. The processor ignores the 32 high-order base-address bits when running in legacy mode. Because the TR is loaded from the GDT, the system-segment descriptor format has been expanded to 16 bytes by the AMD64 architecture in support of 64-bit mode. See “System Descriptors” on page 99 for more information on this expanded format. The high-order base-address bits are only loaded from 64-bit mode using the LTR instruction. Figure 12-5 shows the relationship between the TSS and GDT.

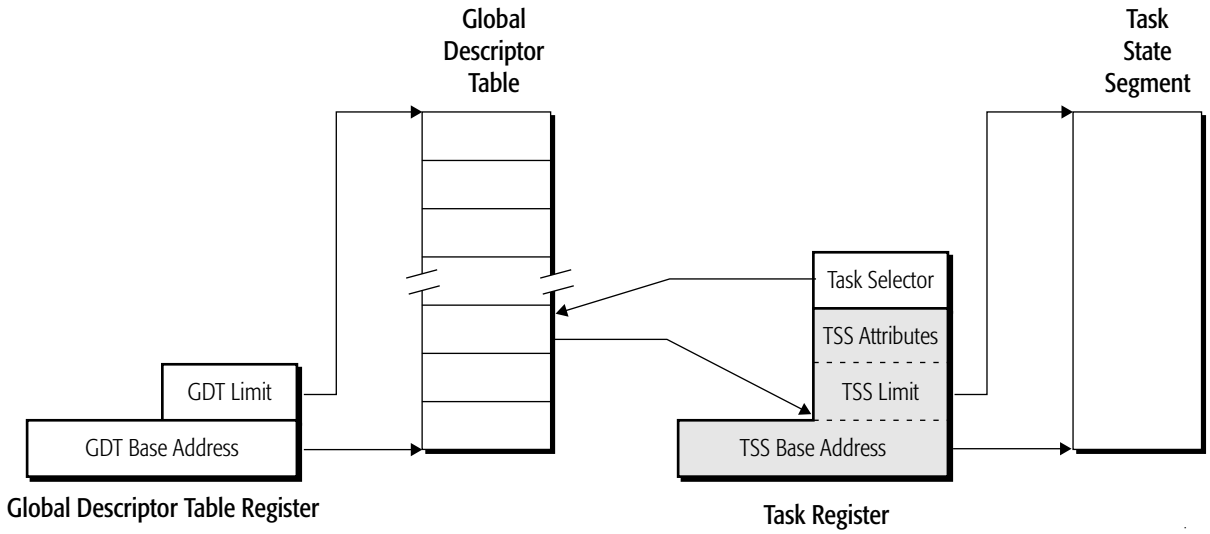


Figure 12-5. Relationship between the TSS and GDT

Long mode requires the use of a 64-bit TSS type, and this type must be loaded into the TR by executing the LTR instruction in *64-bit mode*. Executing the LTR instruction in 64-bit mode loads the TR with the full 64-bit TSS base address from the 16-byte TSS descriptor format (compatibility mode can only load 8-byte system descriptors). A processor running in either compatibility mode or 64-bit mode uses the full 64-bit TR.base address.

12.2.4 Legacy Task-State Segment

The task-state segment (TSS) is a data structure in memory that the processor uses to save and restore the execution state for a task when a task switch occurs. Figure 12-6 on page 368 shows the format of a legacy 32-bit TSS.

Bit Offset 31	16	15	0	Byte Offset
I/O-Permission Bitmap (IOPB) (Up to 8 Kbytes)				IOPB Base
Interrupt-Redirection Bitmap (IRB) (Eight 32-Bit Locations)				
–	Operating-System Data Structure		–	
SSP				+68h
I/O-Permission Bitmap Base Address		Reserved, IGN	T	+64h
Reserved, IGN		LDT Selector		+60h
Reserved, IGN		GS		+5Ch
Reserved, IGN		FS		+58h
Reserved, IGN		DS		+54h
Reserved, IGN		SS		+50h
Reserved, IGN		CS		+4Ch
Reserved, IGN		ES		+48h
EDI				+44h
ESI				+40h
EBP				+3Ch
ESP				+38h
EBX				+34h
EDX				+30h
ECX				+2Ch
EAX				+28h
EFLAGS				+24h
EIP				+20h
CR3				+1Ch
Reserved, IGN		SS2		+18h
ESP2				+14h
Reserved, IGN		SS1		+10h
ESP1				+0Ch
Reserved, IGN		SS0		+08h
ESP0				+04h
Reserved, IGN		Link (Prior TSS Selector)		+00h

Figure 12-6. Legacy 32-bit TSS

The 32-bit TSS contains three types of fields:

- *Static fields* are read by the processor during a task switch when a new task is loaded, but are not written by the processor when a task is suspended.
- *Dynamic fields* are read by the processor during a task switch when a new task is loaded, and are written by the processor when a task is suspended.
- *Software-defined fields* are read and written by software, but are not read or written by the processor. All but the first 104 bytes of a TSS can be defined for software purposes, minus any additional space required for the optional I/O-permission bitmap and interrupt-redirection bitmap.

TSS fields are not read or written by the processor when the LTR instruction is executed. The LTR instruction loads the TSS descriptor into the TR and marks the task as busy, but it does not cause a task switch.

The TSS fields used by the processor in legacy mode are:

- *Link*—Bytes 01h–00h, dynamic field. Contains a copy of the task selector from the previously-executed task. See “Nesting Tasks” on page 382 for additional information.
- *Stack Pointers*—Bytes 1Bh–04h, static field. Contains the privilege 0, 1, and 2 stack pointers for the task. These consist of the stack-segment selector (SS*n)*, and the stack-segment offset (ESP*n*).
- *CR3*—Bytes 1Fh–1Ch, static field. Contains the page-translation-table base-address (CR3) register for the task.
- *EIP*—Bytes 23h–20h, dynamic field. Contains the instruction pointer (EIP) for the next instruction to be executed when the task is restored.
- *EFLAGS*—Bytes 27h–24h, dynamic field. Contains a copy of the EFLAGS image at the point the task is suspended.
- *General-Purpose Registers*—Bytes 47h–28h, dynamic field. Contains a copy of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI values at the point the task is suspended.
- *Segment-Selector Registers*—Bytes 59h–48h, dynamic field. Contains a copy of the ES, CS, SS, DS, FS, and GS, values at the point the task is suspended.
- *LDT Segment-Selector Register*—Bytes 63h–60h, static field. Contains the local-descriptor-table segment selector for the task.
- *T (Trap) Bit*—Bit 0 of byte 64h, static field. This bit, when set to 1, causes a debug exception (#DB) to occur on a task switch. See “Breakpoint Instruction (INT3)” on page 398 for additional information.
- *Shadow Stack Pointer (SSP)*—Bytes 6Bh-68h, static field. Contains the 32-bit SSP for the incoming task. Note that the SSP of the outgoing task is not saved in this field.
- *I/O-Permission Bitmap Base Address*—Bytes 67h–66h, static field. This field represents a 16-bit offset into the TSS. This offset points to the beginning of the I/O-permission bitmap, and the end of the interrupt-redirection bitmap.
- *I/O-Permission Bitmap*—Static field. This field specifies protection for I/O-port addresses (up to the 64K ports supported by the processor), as follows:

- Whether the port can be accessed at any privilege level.
- Whether the port can be accessed outside the privilege level established by EFLAGS.IOPL.
- Whether the port can be accessed when the processor is running in virtual-8086 mode.

Because one bit is used per 8-byte I/O-port, this bitmap can take up to 8 Kbytes of TSS space. The bitmap can be located anywhere within the first 64 Kbytes of the TSS, as long as it is above byte 103. The last byte of the bitmap must contain all ones (0FFh). See “I/O-Permission Bitmap” on page 370 for more information.

- *Interrupt-Redirection Bitmap*—Static field. This field defines how each of the 256-possible software interrupts is directed in a virtual-8086 environment. One bit is used for each interrupt, for a total bitmap size of 32 bytes. The bitmap can be located anywhere above byte 103 within the first 64 Kbytes of the TSS. See “Interrupt Redirection of Software Interrupts” on page 281 for information on using this field.

The TSS can be paged by system software. System software that uses the hardware task-switch mechanism must guarantee that a page fault does not occur during a task switch. Because the processor only reads and writes the first 104 TSS bytes during a task switch, this restriction only applies to those bytes. The simplest approach is to align the TSS on a page boundary so that all critical bytes are either present or not present. Then, if a page fault occurs when the TSS is accessed, it occurs before the first byte is read. If the page fault occurs after a portion of the TSS is read, the fault is unrecoverable.

I/O-Permission Bitmap. The I/O-permission bitmap (IOPB) allows system software to grant less-privileged programs access to individual I/O ports, overriding the effect of RFLAGS.IOPL for those devices. When an I/O instruction is executed, the processor checks the IOPB only if the processor is in virtual x86 mode or the CPL is greater than the RFLAGS.IOPL field. Each bit in the IOPB corresponds to a byte I/O port. A word I/O port corresponds to two consecutive IOPB bits, and a doubleword I/O port corresponds to four consecutive IOPB bits. Access is granted to an I/O port of a given size when *all* IOPB bits corresponding to that port are clear. If any bits are set, a #GP occurs.

The IOPB is located in the TSS, as shown by the example in Figure 12-7 on page 371. Each TSS can have a different copy of the IOPB, so access to individual I/O devices can be granted on a task-by-task basis. The I/O-permission bitmap base-address field located at byte 66h in the TSS is an offset into the TSS locating the start of the IOPB. If all 64K IO ports are supported, the IOPB base address must not be greater than 0DFFFh, otherwise accesses to the bitmap cause a #GP to occur. An extra byte must be present after the last IOPB byte. This byte must have all bits set to 1 (0FFh). This allows the processor to read two IOPB bytes each time an I/O port is accessed. By reading two IOPB bytes, the processor can check all bits when unaligned, multi-byte I/O ports are accessed.

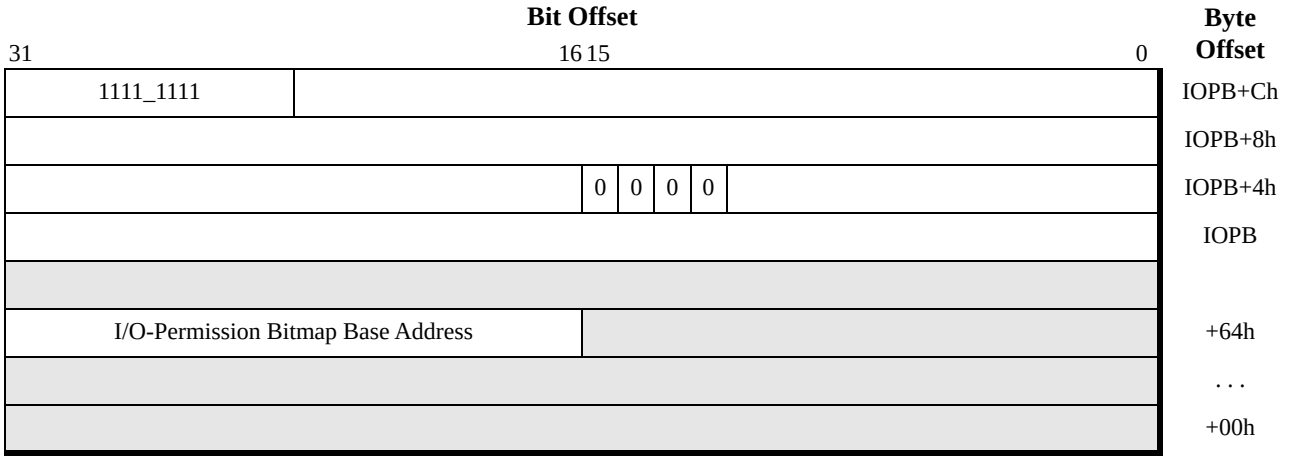


Figure 12-7. I/O-Permission Bitmap Example

Bits in the IOPB sequentially correspond to I/O port addresses. The example in Figure 12-7 shows bits 12 through 15 in the second doubleword of the IOPB cleared to 0. Those bit positions correspond to byte I/O ports 44h through 47h, or alternatively, doubleword I/O port 44h. Because the bits are cleared to zero, software running at any privilege level can access those I/O ports.

By adjusting the TSS limit, it may happen that some ports in the I/O-address space have no corresponding IOPB entry. Ports not represented by the IOPB will cause a #GP exception. Referring again to Figure 12-7, the last IOPB entry is at bit 23 in the fourth IOPB doubleword, which corresponds to I/O port 77h. In this example, all ports from 78h and above will cause a #GP exception, as if their permission bit was set to 1.

12.2.5 64-Bit Task State Segment

Although the hardware task-switching mechanism is not supported in long mode, a 64-bit task state segment (TSS) must still exist. System software must create at least one 64-bit TSS for use after activating long mode, and it must execute the LTR instruction, *in 64-bit mode*, to load the TR register with a pointer to the 64-bit TSS that serves both 64-bit-mode programs and compatibility-mode programs.

The legacy TSS contains several fields used for saving and restoring processor-state information. The legacy fields include general-purpose register, EFLAGS, CR3 and segment-selector register state, among others. Those legacy fields are not supported by the 64-bit TSS. System software must save and restore the necessary processor-state information required by the software-multitasking implementation (if multitasking is supported). Figure 12-8 on page 373 shows the format of a 64-bit TSS.

The 64-bit TSS holds several pieces of information important to long mode that are not directly related to the task-switch mechanism:

- *RSPn*—Bytes 1Bh–04h**.** The full 64-bit canonical forms of the stack pointers (RSP) for privilege levels 0 through 2.

- *ISTn*—Bytes 5Bh–24h**.** The full 64-bit canonical forms of the interrupt-stack-table (IST) pointers. See “Interrupt-Stack Table” on page 276 for a description of the IST mechanism.
- *I/O Map Base Address*—Bytes 67h–66h**.** The 16-bit offset to the I/O-permission bit map from the 64-bit TSS base. The function of this field is identical to that in a legacy 32-bit TSS. See “I/O-Permission Bitmap” on page 370 for more information.

Bit Offset 31	16 15	0	Byte Offset
I/O-Permission Bitmap (IOPB) (Up to 8 Kbytes)			IOPB Base
-		-	
I/O Map Base Address	Reserved, IGN		+64h
Reserved, IGN			+60h +5Ch
IST7[63:32]			+58h
IST7[31:0]			+54h
IST6[63:32]			+50h
IST6[31:0]			+4Ch
IST5[63:32]			+48h
IST5[31:0]			+44h
IST4[63:32]			+40h
IST4[31:0]			+3Ch
IST3[63:32]			+38h
IST3[31:0]			+34h
IST2[63:32]			+30h
IST2[31:0]			+2Ch
IST1[63:32]			+28h
IST1[31:0]			+24h
Reserved, IGN			+20h +1Ch
RSP2[63:32]			+18h
RSP2[31:0]			+14h
RSP1[63:32]			+10h
RSP1[31:0]			+0Ch
RSP0[63:32]			+08h
RSP0[31:0]			+04h
Reserved, IGN			+00h

Figure 12-8. Long Mode TSS Format

12.2.6 Task Gate Descriptor (Legacy Mode Only)

Task-gate descriptors hold a selector reference to a TSS and are used to control access between tasks. Unlike a TSS descriptor or other gate descriptors, a task gate can be located in any of the three descriptor tables (GDT, LDT, and IDT). Figure 12-9 shows the format of a task-gate descriptor.

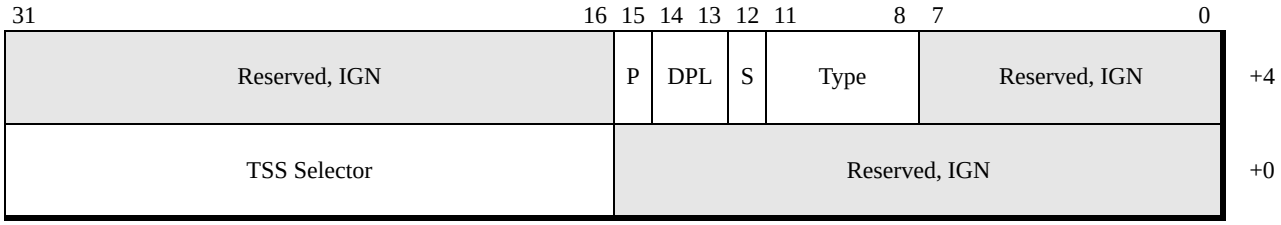


Figure 12-9. Task-Gate Descriptor, Legacy Mode Only

The task-gate descriptor fields are:

- *System (S) and Type*—Bits 12 and 11:8 (respectively) of byte +4. These bits are encoded by software as 00101b to indicate a task-gate descriptor type.
- *Present (P)*—Bit 15 of byte +4. The segment-present bit indicates the segment referenced by the gate descriptor is loaded in memory. If a reference is made to a segment when P=0, a segment-not-present exception (#NP) occurs. This bit is set and cleared by system software and is never altered by the processor.
- *Descriptor Privilege-Level (DPL)*—Bits 14:13 of byte +4. The DPL field indicates the gate-descriptor privilege level. DPL can be set to any value from 0 to 3, with 0 specifying the most privilege and 3 the least privilege.

12.3 Hardware Task-Management in Legacy Mode

This section describes the operation of the task-switch mechanism when the processor is running in legacy mode. None of these features are supported in long mode (either compatibility mode or 64-bit mode).

12.3.1 Task Memory-Mapping

The hardware task-switch mechanism gives system software a great deal of flexibility in managing the sharing and isolation of memory—both virtual (linear) and physical—between tasks.

Segmented Memory. The segmented memory for a task consists of the segments that are loaded during a task switch and any segments that are later accessed by the task code. The hardware task-switch mechanism allows tasks to either share segments with other tasks, or to access segments in isolation from one another. Tasks that share segments actually share a virtual-address (linear-address) space, but they do not necessarily share a physical-address space. When paging is enabled, the virtual-to-physical mapping for each task can differ, as is described in the following section. Shared segments

do share physical memory when paging is disabled, because virtual addresses are used as physical addresses.

A number of options are available to system software that shares segments between tasks:

- Sharing segment descriptors using the GDT. All tasks have access to the GDT, so it is possible for segments loaded in the GDT to be shared among tasks.
- Sharing segment descriptors using a single LDT. Each task has its own LDT, and that LDT selector is automatically saved and restored in the TSS by the processor during task switches. Tasks, however, can share LDTs simply by storing the same LDT selector in multiple TSSs. Using the LDT to manage segment sharing and segment isolation provides more flexibility to system software than using the GDT for the same purpose.
- Copying shared segment descriptors into multiple LDTs. Segment descriptors can be copied by system software into multiple LDTs that are otherwise not shared between tasks. Allowing segment sharing at the segment-descriptor level, rather than the LDT level or GDT level, provides the greatest flexibility to system software.

In all three cases listed above, the actual data and instructions are shared between tasks only when the tasks' virtual-to-physical address mappings are identical.

Paged Memory. Each task has its own page-translation table base-address (CR3) register, and that register is automatically saved and restored in the TSS by the processor during task switches. This allows each task to point to its own set of page-translation tables, so that each task can translate virtual addresses to physical addresses independently. Page translation must be enabled for changes in CR3 values to have an effect on virtual-to-physical address mapping. When page translation is disabled, the tables referenced by CR3 are ignored, and virtual addresses are equivalent to physical addresses.

12.3.2 Switching Tasks

The hardware task-switch mechanism transfers program control to a new task when any of the following occur:

- A CALL or JMP instruction with a selector operand that references a task gate is executed. The task gate can be located in either the LDT or GDT.
- A CALL or JMP instruction with a selector operand that references a TSS descriptor is executed. The TSS descriptor must be located in the GDT.
- A software-interrupt instruction (INT*n*) is executed that references a task gate located in the IDT.
- An exception or external interrupt occurs, and the vector references a task gate located in the IDT.
- An IRET is executed while the EFLAGS.NT bit is set to 1, indicating that a return is being performed from an inner-level task to an outer-level task. The new task is referenced using the selector stored in the current-task link field. See "Nesting Tasks" on page 382 for additional information. The RET instruction *cannot* be used to switch tasks.

When a task switch occurs, the following operations are performed automatically by the processor:

- The processor performs privilege-checking to determine whether the currently-executing program is allowed to access the target task. If this check fails, the task switch is aborted without modifying the processor state, and a general-protection exception (#GP) occurs. The privilege checks performed depend on the cause of the task switch:
  - If the task switch is initiated by a CALL or JMP instruction through a TSS descriptor, the processor checks that both the currently-executing program CPL and the TSS-selector RPL are numerically less-than or equal-to the TSS-descriptor DPL.
  - If the task switch takes place through a task gate, the CPL and task-gate RPL are compared with the task-gate DPL, and no comparison is made using the TSS-descriptor DPL. See “Task Switches Using Task Gates” on page 380.
  - Software interrupts, hardware interrupts, and exceptions all transfer control without checking the task-gate DPL.
  - The IRET instruction transfers control without checking the TSS-descriptor DPL.
- The processor performs limit-checking on the target TSS descriptor to verify that the TSS limit is greater than or equal to:
  - 67h (at least 104 bytes), when shadow stacks are not enabled (CR4.CET=0).
  - 6Bh (at least 108 bytes), when shadow stacks are enabled (CR4.CET=1).

  If this check fails, the task switch is aborted without modifying the processor state, and an invalid-TSS exception (#TS) occurs.
- If shadow stacks are enabled at the current CPL and the task switch was initiated by an IRET instruction, the current SSP must be aligned to 8 bytes. If this check fails, the task switch is aborted without modifying the processor state, and an #TS(current task) exception is generated.
- The current-task state is saved in the TSS. This includes the next-instruction pointer (EIP), EFLAGS, the general-purpose registers, and the segment-selector registers.

  Up to this point, any exception that occurs aborts the task switch without changing the processor state. From this point forward, any exception that occurs does so in the context of the new task. If an exception occurs in the context of the new task during a task switch, the processor finishes loading the new-task state without performing additional checks. The processor transfers control to the #TS handler after this state is loaded, but before the first instruction is executed in the new task. When a #TS occurs, it is possible that some of the state loaded by the processor did not participate in segment access checks. The #TS handler must verify that all segments are accessible before returning to the interrupted task.
- The task register (TR) is loaded with the new-task TSS selector, and the hidden portion of the TR is loaded with the new-task descriptor. The TSS now referenced by the processor is that of the new task.
- The current task is marked as busy. The previous task is marked as available or remains busy, based on the type of linkage. See “Nesting Tasks” on page 382 for more information.
- CR0.TS is set to 1. This bit can be used to save other processor state only when it becomes necessary. For more information, see the next section, “Saving Other Processor State.”

- If shadow stacks are enabled (CR4.CET=1), the following shadow stack actions are performed:

```
saveCsLipSsp = FALSE
checkCsLip   = FALSE
Read CS and EFLAGS from the incoming TSS
newCPL = (EFLAGS.VM == 1) ? 3 : CS.RPL

IF (task switch was initiated by CALL, interrupt or exception)
   {
   IF (ShadowStacksEnabled at current CPL)
     {
     IF ((current CPL == 3) && (newCPL < 3))
        PL3_SSP = SSP     // switching from user to supv
      ELSE
        {
        saveCsLipSsp = TRUE  // all other priv changes
        tempSSP = SSP
        tempLIP = CS.base + EIP
        tempCS  = CS.sel
        }
     }
   } //end task switch initated by CALL,int, or exception

ELSEIF (task switch was initiated by IRET)
   {
   IF (ShadowStacksEnabled at current CPL)
      {
      // pop CS, LIP and SSP from shadow stack
      IF ((newCPL == current CPL) || (newCPL < 3))
        {
        // no priv change, or supv to user/supv
        tempCS  = SSTK_READ_MEM.d [SSP+16]
        tempLIP = SSTK_READ_MEM.d [SSP+8]
        tempSSP = SSTK_READ_MEM.d [SSP]
```

```
        SSP = SSP + 24
        checkCsLip = TRUE
        }
      // check shadow stack token and clear busy bit
       temp_Token = SSTK_READ_MEM.q [SSP]   // read supervisor sstk token
       expected_Token = SSP OR 0x01         // busy bit must be set
       IF (temp_token == expected token)
            SSTK_WRITE_MEM.q [SSP] = SSP    // token OK, clear busy bit
       SSP = 0
      }     // end shadow stacks enabled at current CPL
   } //end task switch initiated by IRET
```

- The new-task state is loaded from the TSS. This includes the next-instruction pointer (EIP), EFLAGS, the general-purpose registers, and the segment-selector registers. The processor clears the segment-descriptor present (P) bits (in the hidden portion of the segment registers) to prevent access into the new segments, until the task switch completes successfully.
- The LDTR and CR3 registers are loaded from the TSS, changing the virtual-to-physical mapping from that of the old task to the new task. Because this is done in the middle of accessing the new TSS, system software must ensure that TSS addresses are translated identically in all tasks.
- The descriptors for all previously-loaded segment selectors are loaded into the hidden portion of the segment registers. This sets or clears the P bits for the segments as specified by the new descriptor values.
- If shadow stacks are enabled (CR4.CET=1), the following shadow stack actions are performed:

```
IF (ShadowStacksEnabled at current CPL)
   {
   IF (EFLAGS.VM == 1)
        EXCEPTION [#TSS(new task selector)]

   IF (task switch was initiated by a CALL, JMP, interrupt or exception)
      {
      newSSP = SSTK_READ_MEM.d [TSS offset 0x68]  // read new SSP
      IF (newSSP[2:0] != 0)                       // must be 8-byte aligned
           EXCEPTION [#TSS(new task selector)]
```

```
      // check token and set busy
      temp_Token = SSTK_READ_MEM.q [newSSP]      // read sstk token
      expected_Token = SSP                       // busy bit must be clear
      IF (temp_token != expected_token)          // token must be valid
            EXCEPTION [#TSS(new task selector)]
      SSTK_WRITE_MEM.q [SSP] = SSP OR 0x01      // valid token, set busy bit
      SSP = newSSP

      IF (saveCsLipSsP == TRUE) // push old CS,LIP,SSP onto new sstk
         {
         SSTK_WRITE_MEM.q [SSP-24] = tempCS
         SSTK_WRITE_MEM.q [SSP-16] = tempLIP
         SSTK_WRITE_MEM.q [SSP-8]  = tempSSP
         SSP = SSP - 24
         }
      } // end task switch initiated by CALL, JMP, interrupt or exception
   } // end shadow stacks enabled at current CPL

ELSEIF (task switch was initiated by an IRET)
   {
   IF (checkCsLip == TRUE)
      {                              // check CS, LIP against shadow stack
      IF (tempCS != CS)
            EXCEPTION [#CP(RETF/IRET)]  // CS must match
      IF (tempLIP != (CS.base + EIP))
            EXCEPTION [#CP(RETF/IRET)]  // LIP must match
      }

   IF ShadowStackEnabled at newCPL
      {
```

```
      IF (!checkCsLip)
         tempSSP = PL3_SSP;
      IF (tempSSP[1:0] != 0)          // SSP must be 4-byte aligned
               EXCEPTION [#CP(RETF/IRET)]
      IF (tempSSP[63:32] != 0)         // and SSP must be <4 GB
               EXCEPTION [#CP(RETF/IRET)]
      SSP = tempSSP
      }  // end shadow stacks enabled at new CPL
   } // end task swtich initiated by IRET
```

If the above steps complete successfully, the processor begins executing instructions in the new task beginning with the instruction referenced by the CS:EIP far pointer loaded from the new TSS. The privilege level of the new task is taken from the new CS segment selector's RPL.

Saving Other Processor State. The processor does not automatically save the registers used by the media or x87 instructions. Instead, the processor sets CR0.TS to 1 during a task switch. Later, when an attempt is made to execute any of the media or x87 instructions while TS=1, a device-not-available exception (#NM) occurs. System software can then save the previous state of the media and x87 registers and clear the CR0.TS bit to 0 before executing the next media/x87 instruction. As a result, the media and x87 registers are saved only when necessary after a task switch.

12.3.3 Task Switches Using Task Gates

When a control transfer to a new task occurs through a task gate, the processor reads the task-gate DPL (DPL_G) from the task-gate descriptor. Two privilege checks, both of which must pass, are performed on DPL_G before the task switch can occur successfully:

- The processor compares the CPL with DPL_G. The CPL must be numerically *less than or equal to* DPL_G for this check to pass. In other words, the following expression must be true: $CPL \leq DPL_G$.
- The processor compares the RPL in the task-gate selector with DPL_G. The RPL must be numerically *less than or equal to* DPL_G for this check to pass. In other words, the following expression must be true: $RPL \leq DPL_G$.

Unlike call-gate control transfers, the processor does not read the DPL from the target TSS descriptor (DPL_S) and compare it with the CPL when a task gate is used.

Figure 12-10 on page 382 shows two examples of task-gate privilege checks. In Example 1, the privilege checks pass:

- The task-gate DPL (DPL_G) is at the lowest privilege (3), specifying that software running at any privilege level (CPL) can access the gate.

- The selector referencing the task gate passes its privilege check because the RPL is numerically less than or equal to DPL_G.

In Example 2, both privilege checks fail:

- The task-gate DPL (DPL_G) specifies that only software at privilege-level 0 can access the gate. The current program does not have enough privilege to access the task gate, because its CPL is 2.
- The selector referencing the task-gate descriptor does not have a high enough privilege to complete the reference. Its RPL is numerically greater than DPL_G.

Although both privilege checks failed in the example, if only one check fails, access into the target task is denied.

Because the legacy task-switch mechanism is not supported in long mode, *software cannot use task gates in long mode*. Any attempt to transfer control to another task using a task gate in long mode causes a general-protection exception (#GP) to occur.

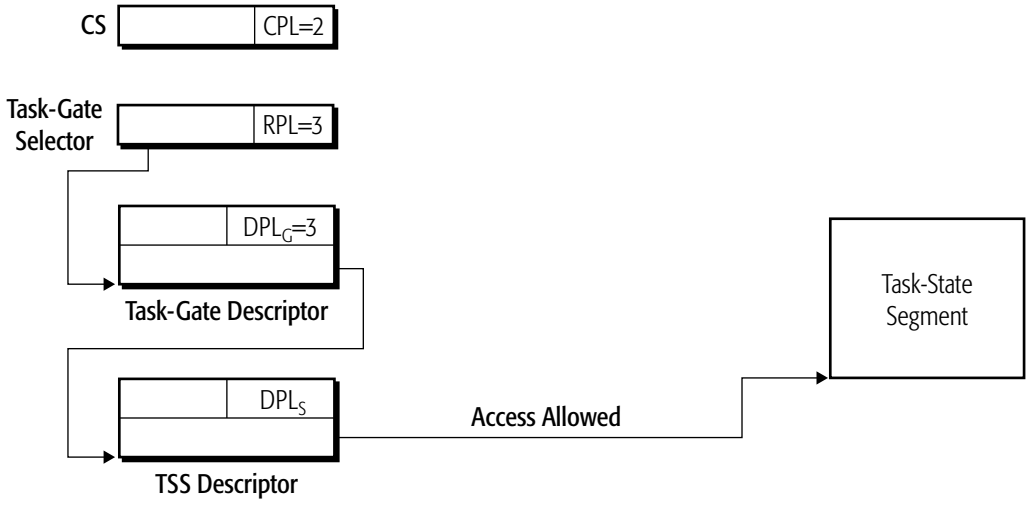


Figure 12-10. Privilege-Check Examples for Task Gates

12.3.4 Nesting Tasks

The hardware task-switch mechanism supports task nesting through the use of EFLAGS *nested-task* (NT) bit and the TSS link-field. The manner in which these fields are updated and used during a task switch depends on how the task switch is initiated:

- The JMP instruction does not update EFLAGS.NT or the TSS link-field. Task nesting is not supported by the JMP instruction.
- The CALL instruction, INT*n* instructions, interrupts, and exceptions can only be performed from outer-level tasks to inner-level tasks. All of these operations set the EFLAGS.NT bit for the new task to 1 during a task switch, and copy the selector for the previous task into the new-task link field.
- An IRET instruction which returns to another task only occurs when the EFLAGS.NT bit for the current task is set to 1, and only can be performed from an inner-level task to an outer-level task. When an IRET results in a task switch, the new task is referenced using the selector stored in the current-TSS link field. The EFLAGS.NT bit for the current task is cleared to 0 during the task switch.

Table 12-1 summarizes the effect various task-switch initiators have on EFLAGS.NT, the TSS link-field, and the TSS-busy bit. (For more information on the busy bit, see the next section, “Preventing Recursion.”)

Table 12-1. Effects of Task Nesting

Task-Switch Initiator	Old Task			New Task		
	EFLAGS.NT	Link (Selector)	Busy	EFLAGS.NT	Link (Selector)	Busy
JMP	—	—	Clear to 0 (was 1)	—	—	Set to 1
CALL INT*n* Interrupt Exception	—		— (Was 1)	Set to 1	Old Task	Set to 1
IRET	Clear to 0 (was 1)	—	Clear to 0 (was 1)	—		

Note:
“—” indicates no change is made.

Programs running at any privilege level can set EFLAGS.NT to 1 and execute the IRET instruction to transfer control to another task. System software can keep control over improperly nested-task switches by initializing the link field of all TSSs that it creates. That way, improperly nested-task switches always transfer control to a known task.

Preventing Recursion. Task recursion is not allowed by the hardware task-switch mechanism. If recursive-task switches were allowed, they would replace a previous task-state image with a newer image, discarding the previous information. To prevent recursion from occurring, the processor uses the busy bit located in the TSS-descriptor type field (bit 9 of byte +4). Use of this bit depends on how the task switch is initiated:

- The JMP instruction clears the busy bit in the old task to 0 and sets the busy bit in the new task to 1. A general-protection exception (#GP) occurs if an attempt is made to JMP to a task with a set busy bit.
- The CALL instruction, INT*n* instructions, interrupts, and exceptions set the busy bit in the new task to 1. The busy bit in the old task remains set to 1, preventing recursion through task-nesting levels. A general-protection exception (#GP) occurs if an attempt is made to switch to a task with a set busy bit.
- An IRET to another task (EFLAGS.NT must be 1) clears the busy bit in the old task to 0. The busy bit in the new task is not altered, because it was already set to 1.

Table 12-1 on page 383 summarizes the effect various task-switch initiators have on the TSS-busy bit.

13 Software Debug and Performance Resources

Testing, debug, and performance optimization consume a significant portion of the time needed to develop a new computer or software product and move it successfully into production. To stay competitive, product developers need tools that allow them to rapidly detect, isolate, and correct problems before a product is shipped. The goal of the debug and performance features incorporated into processor implementations of the AMD64 architecture is to support the tool chain solutions used in software and hardware product development.

The debug and performance resources that can be supported by AMD64 architecture implementations include:

- *Software Debug*—Software-debug facilities include the debug registers (DR0–DR7), debug exception, and breakpoint exception. Additional features are provided using model-specific registers (MSRs). These registers are used to set breakpoints on branches, interrupts, and exceptions and to single step from one branch to the next. The software-debug capability is described in the following section.
- *Performance Monitoring Counters*—Performance monitoring counters (PMCs) are provided to count specific processor hardware events. A set of control registers allow the selection of events to be monitored and a corresponding set of counter registers track the frequency of monitored events. These counters are described in Section 13.2 “Performance Monitoring Counters” on page 400.
- *Instruction-Based Sampling*— Instruction-based sampling is a hardware-based facility that enables system software to capture specific data concerning instruction fetch and instruction execution operation based on random sampling. This facility is described in Section 13.3 “Instruction-Based Sampling” on page 409.
- *Lightweight Profiling*—AMD64 architecture provides instructions that allow user-level programs to manage the gathering of instruction statistics using very little overhead. This facility is described in Section 13.4 “Lightweight Profiling” on page 422.

Although a subset of the facilities listed are available in all processor implementations, the remainder are optional. Support for optional facilities is indicated via CPUID feature bits. The means of determining support for each architected facility is described along with the facility in the sections that follow.

A given processor product may include additional debug and performance monitoring capabilities beyond those which are architecturally-defined. For details see the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

13.1 Software-Debug Resources

Software can program breakpoints into the debug registers, causing a debug exception (#DB) when matches occur on instruction-memory addresses, data-memory addresses, or I/O addresses. The breakpoint exception (#BP) is also supported to allow software to set breakpoints by placing INT3 instructions in the instruction memory for a program. Program control is transferred to the breakpoint exception (#BP) handler when an INT3 instruction is executed.

In addition to the debug features supported by the debug registers (DR0–DR7), the processor also supports features supported by model-specific registers (MSRs). Together, these capabilities provide a rich set of breakpoint conditions, including:

- *Breakpoint On Address Match*—Breakpoints occur when the address stored in a address-breakpoint register matches the address of an instruction or data reference. Up to four address-match breakpoint conditions can be set by software.
- *Single Step All Instructions*—Breakpoints can be set to occur on every instruction, allowing a debugger to examine the contents of registers as a program executes.
- *Single Step Control Transfers*—Breakpoints can be set to occur on control transfers, such as calls, jumps, interrupts, and exceptions. This can allow a debugger to narrow a problem search to a specific section of code before enabling single stepping of all instructions.
- *Breakpoint On Any Instruction*—Breakpoints can be set on any specific instruction using either the address-match breakpoint condition or using the INT3 instruction to force a breakpoint when the instruction is executed.
- *Breakpoint On Task Switch*—Software forces a #DB exception to occur when a task switch is performed to a task with the T bit in the TSS set to 1. Debuggers can use this capability to enable or disable debug conditions for a specific task.

Problem areas can be identified rapidly using the information supplied by the debug registers when breakpoint conditions occur:

- Special conditions that cause a #DB exception are recorded in the DR6 debug-status register, including breakpoints due to task switches and single stepping. The DR6 register also identifies which address-breakpoint register (DR0–DR3) caused a #DB exception due to an address match. When combined with the DR7 debug-control register settings, the cause of a #DB exception can be identified.
- To assist in analyzing the instruction sequence a processor follows in reaching its current state, the source and destination addresses of control-transfer events are saved by the processor. These include branches (calls and jumps), interrupts, and exceptions. Debuggers can use this information to narrow a problem search to a specific section of code before single stepping all instructions.

13.1.1 Debug Registers

The AMD64 architecture supports the legacy debug registers, DR0–DR7. These registers are expanded to 64 bits by the AMD64 architecture. In legacy mode and in compatibility mode, only the lower 32 bits are used. In these modes, writes to a debug register fill the upper 32 bits with zeros, and

reads from a debug register return only the lower 32 bits. In 64-bit mode, all 64 bits of the debug registers are read and written. Operand-size prefixes are ignored.

The debug registers can be read and written only when the current-protection level (CPL) is 0 (most privileged). Attempts to read or write the registers at a lower-privilege level (CPL>0) cause a general-protection exception (#GP).

Several debug registers described below are model-specific registers (MSRs). See “Software-Debug MSRs” on page 672 for a listing of the debug-MSR numbers and their reset values. Some processor implementations include additional MSRs used to support implementation-specific software debug features. For more information on these registers and their capabilities, see the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

13.1.1.1 Address-Breakpoint Registers (DR0-DR3)

Figure 13-1 shows the format of the four address-breakpoint registers, DR0-DR3. Software can load a virtual (linear) address into any of the four registers, and enable breakpoints to occur when the address matches an instruction or data reference. The MOV DR*n* instructions *do not* check that the virtual addresses loaded into DR0–DR3 are in canonical form. Breakpoint conditions are enabled using the debug-control register, DR7 (see “Debug-Control Register (DR7)” on page 389).

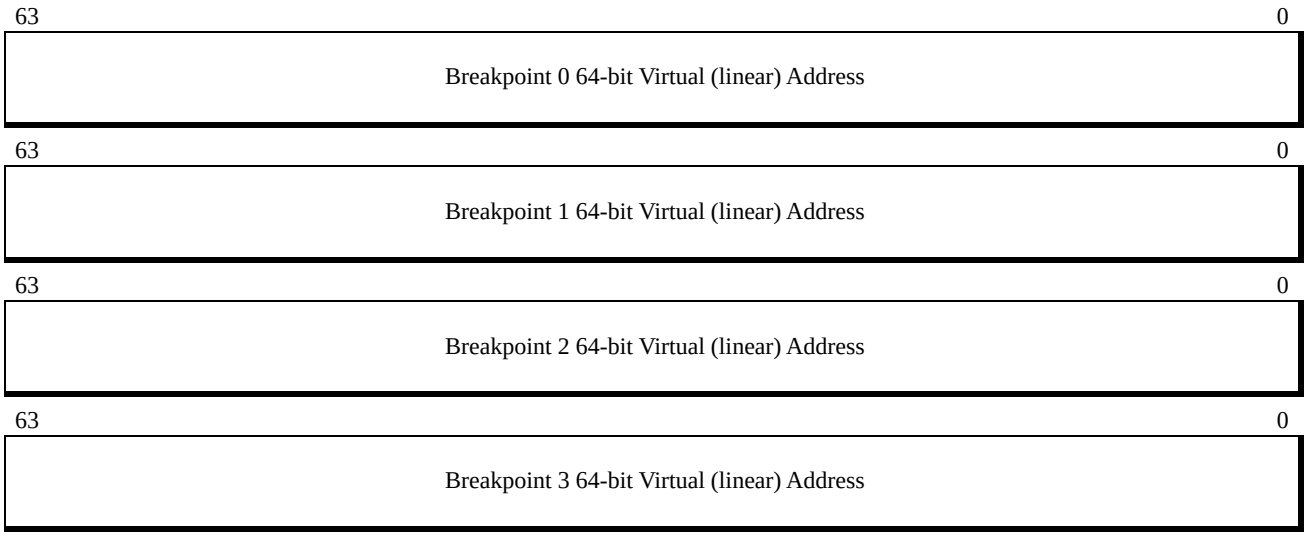


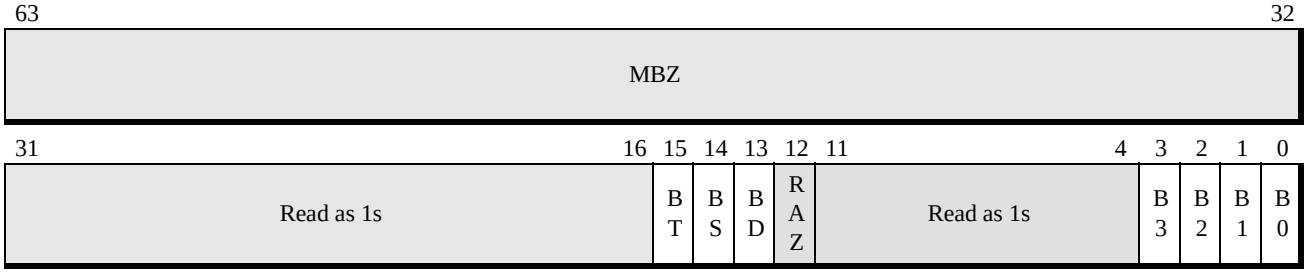
Figure 13-1. Address-Breakpoint Registers (DR0–DR3)

13.1.1.2 Reserved Debug Registers (DR4, DR5)

The DR4 and DR5 registers are reserved and should not be used by software. These registers are aliased to the DR6 and DR7 registers, respectively. When the debug extensions are enabled (CR4[DE] = 1) attempts to access these registers cause an invalid-opcode exception (#UD).

13.1.1.3 Debug-Status Register (DR6)

Figure 13-2 on page 388 shows the format of the debug-status register, DR6. Debug status is loaded into DR6 when an enabled debug condition is encountered that causes a #DB exception.



Bits	Mnemonic	Description	Access type
63:16	—	Reserved	MBZ
15	BT	Breakpoint Task Switch	R/W
14	BS	Breakpoint Single Step	R/W
13	BD	Breakpoint Debug Access Detected	R/W
12	—	Reserved	RAZ
11:4	—	Reserved	RAI
3	B3	Breakpoint #3 Condition Detected	R/W
2	B2	Breakpoint #2 Condition Detected	R/W
1	B1	Breakpoint #1 Condition Detected	R/W
0	B0	Breakpoint #0 Condition Detected	R/W

Figure 13-2. Debug-Status Register (DR6)

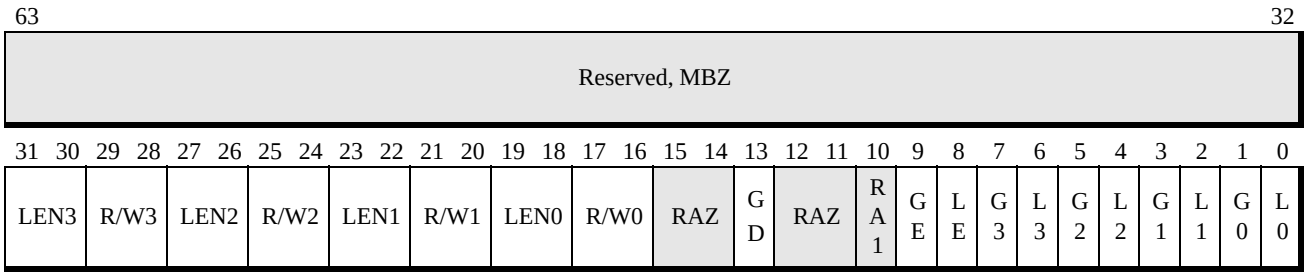
Bits 15:13 of the DR6 register are not cleared by the processor and must be cleared by software after the contents have been read. Register fields are:

- *Breakpoint-Condition Detected (B3–B0)*—Bits 3:0. The processor updates these four bits on every debug breakpoint or general-detect condition. A bit is set to 1 if the corresponding address-breakpoint register detects an enabled breakpoint condition, as specified by the DR7 L*n*, G*n*, R/W*n* and LEN*n* controls, and is cleared to 0 otherwise. For example, B1 (bit 1) is set to 1 if an address-breakpoint condition is detected by DR1.
- *Debug-Register-Access Detected (BD)*—Bit 13. The processor sets this bit to 1 if software accesses any debug register (DR0–DR7) while the general-detect condition is enabled (DR7[GD] = 1).
- *Single Step (BS)*—Bit 14. The processor sets this bit to 1 if the #DB exception occurs as a result of single-step mode (rFLAGS[TF] = 1). Single-step mode has the highest-priority among debug exceptions. Other status bits within the DR6 register can be set by the processor along with the BS bit.
- *Task-Switch (BT)*—Bit 15. The processor sets this bit to 1 if the #DB exception occurred as a result of task switch to a task with a TSS T-bit set to 1.

All remaining bits in the DR6 register are reserved. Reserved bits 31:16 and 11:4 must all be set to 1, while reserved bit 12 must be cleared to 0. In 64-bit mode, the upper 32 bits of DR6 are reserved and must be written with zeros. Writing a 1 to any of the upper 32 bits results in a general-protection exception, #GP(0).

13.1.1.4 Debug-Control Register (DR7)

Figure 13-3 shows the format of the debug-control register, DR7. DR7 is used to establish the breakpoint conditions for the address-breakpoint registers (DR0–DR3) and to enable debug exceptions for each address-breakpoint register individually. DR7 is also used to enable the general-detect breakpoint condition.



Bits	Mnemonic	Description	Access type
63:32	—	Reserved	MBZ
31:30	LEN3	Length of Breakpoint #3	R/W
29:28	R/W3	Type of Transaction(s) to Trap	R/W
27:26	LEN2	Length of Breakpoint #2	R/W
25:24	R/W2	Type of Transaction(s) to Trap	R/W
23:22	LEN1	Length of Breakpoint #1	R/W
21:20	R/W1	Type of Transaction(s) to Trap	R/W
19:18	LEN0	Length of Breakpoint #0	R/W
17:16	R/W0	Type of Transaction(s) to Trap	R/W
15:14	—	Reserved	RAZ
13	GD	General Detect Enabled	R/W
12:11	—	Reserved	RAZ
10	—	Reserved	RA1
9	GE	Global Exact Breakpoint Enabled	R/W
8	LE	Local Exact Breakpoint Enabled	R/W
7	G3	Global Exact Breakpoint #3 Enabled	R/W
6	L3	Local Exact Breakpoint #3 Enabled	R/W
5	G2	Global Exact Breakpoint #2 Enabled	R/W
4	L2	Local Exact Breakpoint #2 Enabled	R/W
3	G1	Global Exact Breakpoint #1 Enabled	R/W
2	L1	Local Exact Breakpoint #1 Enabled	R/W
1	G0	Global Exact Breakpoint #0 Enabled	R/W
0	L0	Local Exact Breakpoint #0 Enabled	R/W

Figure 13-3. Debug-Control Register (DR7)

The fields within the DR7 register are all read/write. These fields are:

- *Local-Breakpoint Enable (L3–L0)*—Bits 6, 4, 2, and 0 (respectively). Software individually sets these bits to 1 to enable debug exceptions to occur when the corresponding address-breakpoint register (DR*n*) detects a breakpoint condition while executing the *current* task. For example, if L1 (bit 2) is set to 1 and an address-breakpoint condition is detected by DR1, a #DB exception occurs. These bits are cleared to 0 by the processor when a hardware task-switch occurs.
- *Global-Breakpoint Enable (G3–G0)*—Bits 7, 5, 3, and 1 (respectively). Software sets these bits to 1 to enable debug exceptions to occur when the corresponding address-breakpoint register (DR*n*) detects a breakpoint condition while executing *any* task. For example, if G1 (bit 3) is set to 1 and an address-breakpoint condition is detected by DR1, a #DB exception occurs. These bits are never cleared to 0 by the processor.
- *Local-Enable (LE)*—Bit 8. Software sets this bit to 1 in legacy implementations to enable exact breakpoints while executing the *current* task. This bit is ignored by implementations of the AMD64 architecture. All breakpoint conditions, except certain string operations preceded by a repeat prefix, are exact.
- *Global-Enable (GE)*—Bit 9. Software sets this bit to 1 in legacy implementations to enable exact breakpoints while executing *any* task. This bit is ignored by implementations of the AMD64 architecture. All breakpoint conditions, except certain string operations preceded by a repeat prefix, are exact.
- *General-Detect Enable (GD)*—Bit 13. Software sets this bit to 1 to cause a debug exception to occur when an attempt is made to execute a MOV DR*n* instruction to any debug register (DR0–DR7). This bit is cleared to 0 by the processor when the #DB handler is entered, allowing the handler to read and write the DR*n* registers. The #DB exception occurs before executing the instruction, and DR6[BD] is set by the processor. Software debuggers can use this bit to prevent the currently-executing program from interfering with the debug operation.
- *Read/Write (R/W3–R/W0)*—Bits 29:28, 25:24, 21:20, and 17:16 (respectively). Software sets these fields to control the breakpoint conditions used by the corresponding address-breakpoint registers (DR*n*). For example, control-field R/W1 (bits 21:20) controls the breakpoint conditions for the DR1 register. The R/W*n* control-field encodings specify the following conditions for an address-breakpoint to occur:
  - 00—Only on instruction execution.
  - 01—Only on data write.
  - 10—This encoding is further qualified by CR4[DE] as follows:
    . CR4[DE] = 0—Condition is undefined.
    . CR4[DE] = 1—Only on I/O read or I/O write.
  - 11—Only on data read or data write.
- *Length (LEN3–LEN0)*—Bits 31:30, 27:26, 23:22, and 19:18 (respectively). Software sets these fields to control the range used in comparing a memory address with the corresponding address-breakpoint register (DR*n*). For example, control-field LEN1 (bits 23:22) controls the breakpoint-comparison range for the DR1 register.

The value in DR*n* defines the low-end of the address range used in the comparison. LEN*n* is used to mask the low-order address bits in the corresponding DR*n* register so that they are not used in the address comparison. To work properly, breakpoint boundaries must be aligned on an address corresponding to the range size specified by LEN*n*. The LEN*n* control-field encodings specify the following address-breakpoint-comparison ranges:

- 00—1 byte.
- 01—2 byte, must be aligned on a word boundary.
- 10—8 byte, must be aligned on a quadword boundary. (Long mode only; otherwise undefined.)
- 11—4 byte, must be aligned on a doubleword boundary.

If the R/W*n* field is used to specify instruction breakpoints (R/W*n*=00), the corresponding LEN*n* field must be set to 00. Setting LEN*n* to any other value produces undefined results.

All remaining bits in the DR7 register are reserved. Reserved bits 15:14 and 12:11 must all be cleared to 0, while reserved bit 10 must be set to 1. In 64-bit mode, the upper 32 bits of DR7 are reserved and must be written with zeros. Writing a 1 to any of the upper 32 bits results in a general-protection #GP(0) exception.

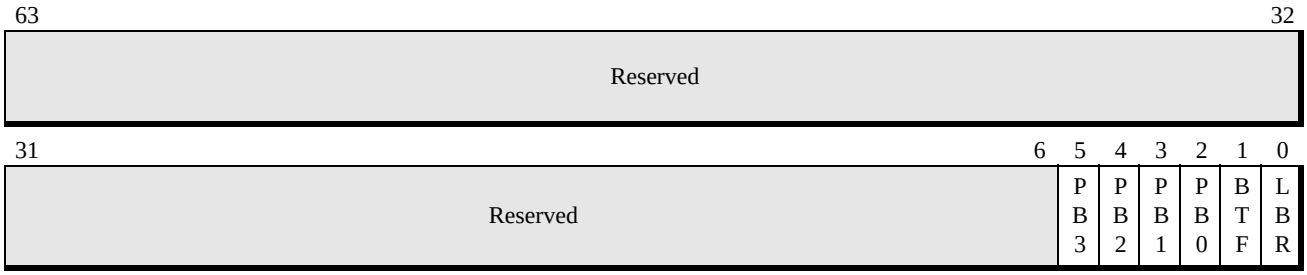
13.1.1.5 64-Bit-Mode Extended Debug Registers

In 64-bit mode, additional encodings for debug registers are available. The R bit of the REX prefix is used to modify the ModRM *reg* field when that field encodes a control register. These additional encodings enable the processor to address DR8–DR15.

Access to the DR8–DR15 registers is implementation-dependent. The architecture does not require any of these extended debug registers to be implemented. Any attempt to access an unimplemented register results in an invalid-opcode exception (#UD).

13.1.1.6 Debug-Control MSR (DebugCtl)

Figure 13-4 on page 392 shows the format of the debug-control MSR (DebugCtl). DebugCtl provides additional debug controls over control-transfer recording and single stepping, and external-breakpoint reporting and trace messages. DebugCtl is read and written using the RDMSR and WRMSR instructions.



Bits	Mnemonic	Description	R/W
63:6	Reserved		
5	PB3	Performance Monitoring Pin Control	R/W
4	PB2	Performance Monitoring Pin Control	R/W
3	PB1	Performance Monitoring Pin Control	R/W
2	PB0	Performance Monitoring Pin Control	R/W
1	BTF	Branch Single Step	R/W
0	LBR	Last-Branch Record	R/W

Figure 13-4. Debug-Control MSR (DebugCtl)

The fields within the DebugCtl register are:

- *Last-Branch Record (LBR)*—Bit 0, read/write. Software sets this bit to 1 to cause the processor to record the source and target addresses of the last control transfer taken before a debug exception occurs. The recorded control transfers include branch instructions, interrupts, and exceptions. See “Control-Transfer Breakpoint Features” on page 398 for more details on the registers. See Figure 13-5 on page 393 for the format of the control-transfer recording MSRs.
- *Branch Single Step (BTF)*—Bit 1, read/write. Software uses this bit to change the behavior of the rFLAGS[TF] bit. When this bit is cleared to 0, the rFLAGS[TF] bit controls instruction single stepping, (normal behavior). When this bit is set to 1, the rFLAGS[TF] bit controls single stepping on control transfers. The single-stepped control transfers include branch instructions, interrupts, and exceptions. Control-transfer single stepping requires both BTF = 1 and rFLAGS[TF] = 1. See “Control-Transfer Breakpoint Features” on page 398 for more details on control-transfer single stepping.
- *Performance-Monitoring/Breakpoint Pin-Control (PBi)*—Bits 5:2, read/write. Software uses these bits to control the type of information reported by the four external performance-monitoring/breakpoint pins on the processor. When a PB*i* bit is cleared to 0, the corresponding external pin (BP*i*) reports performance-monitor information. When a PB*i* bit is set to 1, the corresponding external pin (BP*i*) reports breakpoint information.

All remaining bits in the DebugCtl register are reserved.

13.1.1.7 Control-Transfer Recording MSRs

Figure 13-5 on page 393 shows the format of the 64-bit control-transfer recording MSRs: LastBranchToIP, LastBranchFromIP, LastIntToIP, and LastIntFromIP. These registers are loaded automatically by the processor when the DebugCtl[LBR] bit is set to 1. These MSRs are read-only.

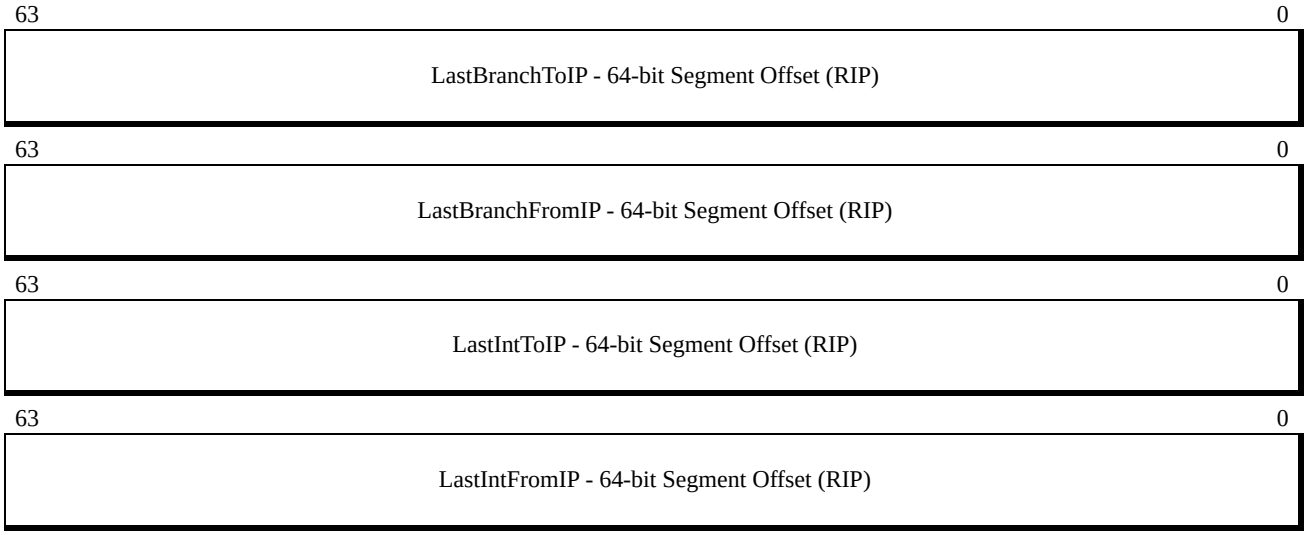


Figure 13-5. Control-Transfer Recording MSRs

13.1.2 Setting Breakpoints

Breakpoints can be set to occur on either instruction addresses or data addresses using the breakpoint-address registers, DR0–DR3 (DR*n*). The values loaded into these registers represent the breakpoint-location virtual address. The debug-control register, DR7, is used to enable the breakpoint registers and to specify the type of access and the range of addresses that can trigger a breakpoint.

Software enables the DR*n* registers using the corresponding local-breakpoint enable (L*n*) or global-breakpoint enable (G*n*) found in the DR7 register. L*n* is used to enable breakpoints only while the current task is active, and it is cleared by the processor when a task switch occurs. G*n* is used to enable breakpoints for all tasks, and it is never cleared by the processor.

The R/W*n* fields in DR7, along with the CR4[DE] bit, specify the type of access required to trigger a breakpoint when an address match occurs on the corresponding DR*n* register. Breakpoints can be set to occur on instruction execution, data reads and writes, and I/O reads and writes. The R/W*n* and CR4[DE] encodings used to specify the access type are described on page 390 of “Debug-Control Register (DR7).”

The LEN*n* fields in DR7 specify the size of the address range used in comparison with data or instruction addresses. LEN*n* is used to mask the low-order address bits in the corresponding DR*n* register so that they are not used in the address comparison. Breakpoint boundaries must be aligned on an address corresponding to the range size specified by LEN*n*. Assuming the access type matches the

type specified by R/W*n*, a breakpoint occurs if any accessed byte falls within the range specified by LEN*n*. For instruction breakpoints, LEN*n* must specify a single-byte range. The LEN*n* encodings used to specify the address range are described on page 390 of "Debug-Control Register (DR7)."

Table 13-1 shows several examples of data accesses, and whether or not they cause a #DB exception to occur based on the breakpoint address in DR*n* and the breakpoint-address range specified by LEN*n*. In this table, R/W*n* always specifies read/write access.

Table 13-1. Breakpoint-Setting Examples

Data-Access Address (hexadecimal)	Access Size (bytes)	Byte-Addresses in Data-Access (hexadecimal)	Breakpoint-Address Range (hexadecimal)	Result
DR*n*=F000, LEN*n*=00 (1 Byte)				
EFFB	8	EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001	F000	#DB
EFFE	2	EFFE, EFFF		—
	4	EFFE, EFFF, F000, F001		#DB
F000	1	F000		
F001	2	F001, F002		—
F005	4	F005, F006, F007, F008		
DR*n*=F004, LEN*n*=11 (4 Bytes)				
EFFB	8	EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001	F004–F007	—
EFFE	2	EFFE, EFFF		
	4	EFFE, EFFF, F000, F001		
F000	1	F000		
F001	2	F001, F002		
F005	4	F005, F006, F007, F008		#DB
DR*n*=F005, LEN*n*=10 (8 Bytes)				
EFFB	8	EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001	F000–F007	#DB
EFFE	2	EFFE, EFFF		—
	4	EFFE, EFFF, F000, F001		#DB
F000	1	F000		
F001	2	F001, F002		
F005	4	F005, F006, F007, F008		

Note:
"—" indicates no #DB occurs.

13.1.3 Using Breakpoints

A debug exception (#DB) occurs when an enabled-breakpoint condition is encountered during program execution. The debug-handler must check the debug-status register (DR6), the conditions enabled by the debug-control register (DR7), and the debug-control MSR (DebugCtl), to determine the #DB cause. The #DB exception corresponds to interrupt vector 1. See "#DB—Debug Exception (Vector 1)" on page 239.

Instruction breakpoints and general-detect conditions cause the #DB exception to occur *before* the instruction is executed, while all other breakpoint and single-stepping conditions cause the #DB exception to occur *after* the instruction is executed. Table 13-2 summarizes where the #DB exception occurs based on the breakpoint condition.

Table 13-2. Breakpoint Location by Condition

Breakpoint Condition	Breakpoint Location
Instruction	Before Instruction is Executed
General Detect	
Data Write Only	After Instruction is Executed[1]
Data Read or Data Write	
I/O Read or I/O Write	
Single Step[1]	After Instruction is Executed
Task Switch	

Note:
1. *Repeated operations (REP prefix) can breakpoint between iterations.*

Instruction breakpoints and general-detect conditions have a lower interrupt-priority than the other breakpoint and single-stepping conditions (see "Priorities" on page 255). Data-breakpoint conditions on the *previous* instruction occur before an instruction-breakpoint condition on the *next* instruction. However, if instruction and data breakpoints can occur as a result of executing a *single* instruction, the instruction breakpoint occurs first (before the instruction is executed), followed by the data breakpoint (after the instruction is executed).

13.1.3.1 Instruction Breakpoints

Instruction breakpoints are set by loading a breakpoint-address register (DR*n*) with the desired instruction virtual-address, and then setting the corresponding DR7 fields as follows:

- L*n* or G*n* is set to 1 to enable the breakpoint for either the local task or all tasks, respectively.
- R/W*n* is set to 00b to specify that the contents of DR*n* are to be compared only with the virtual address of the next instruction to be executed.
- LEN*n* must be set to 00b.

When a #DB exception occurs due to an instruction breakpoint-address in DR*n*, the corresponding B*n* field in DR6 is set to 1 to indicate that a breakpoint condition occurred. The breakpoint occurs before

the instruction is executed, and the breakpoint-instruction address is pushed onto the debug-handler stack. If multiple instruction breakpoints are set, the debug handler can use the B*n* field to identify which register caused the breakpoint.

Returning from the debug handler causes the breakpoint instruction to be executed. Before returning from the debug handler, the rFLAGS[RF] bit should be set to 1 to prevent a reoccurrence of the #DB exception due to the instruction-breakpoint condition. The processor ignores instruction-breakpoint conditions when rFLAGS[RF] = 1, until after the next instruction (in this case, the breakpoint instruction) is executed. After the next instruction is executed, the processor clears rFLAGS[RF].

13.1.3.2 Data Breakpoints

Data breakpoints are set by loading a breakpoint-address register (DR*n*) with the desired data virtual-address, and then setting the corresponding DR7 fields as follows:

- L*n* or G*n* is set to 1 to enable the breakpoint for either the local task or all tasks, respectively.
- R/W*n* is set to 01b to specify that the data virtual-address is compared with the contents of DR*n* only during a memory-write. Setting this field to 11b specifies that the comparison takes place during both memory reads and memory writes.
- LEN*n* is set to 00b, 01b, 11b, or 10b to specify an address-match range of one, two, four, or eight bytes, respectively. Long mode must be active to set LEN*n* to 10b.

When a #DB exception occurs due to a data breakpoint address in DR*n*, the corresponding B*n* field in DR6 is set to 1 to indicate that a breakpoint condition occurred. The breakpoint occurs after the data-access instruction is executed, which means that the original data is overwritten by the data-access instruction. If the debug handler needs to report the previous data value, it must save that value before setting the breakpoint.

Because the breakpoint occurs after the data-access instruction is executed, the address of the instruction following the data-access instruction is pushed onto the debug-handler stack. Repeated string instructions, however, can trigger a breakpoint before all iterations of the repeat loop have completed. When this happens, the address of the string instruction is pushed onto the stack during a #DB exception if the repeat loop is not complete. A subsequent IRET from the #DB handler returns to the string instruction, causing the remaining iterations to be executed. Most implementations cannot report breakpoints exactly for repeated string instructions, but instead report the breakpoint on an iteration later than the iteration where the breakpoint occurred.

13.1.3.3 I/O Breakpoints

I/O breakpoints are set by loading a breakpoint-address register (DR*n*) with the I/O-port address to be trapped, and then setting the corresponding DR7 fields as follows:

- L*n* or G*n* is set to 1 to enable the breakpoint for either the local task or all tasks, respectively.
- R/W*n* is set to 10b to specify that the I/O-port address is compared with the contents of DR*n* only during execution of an I/O instruction. This encoding of R/W*n* is valid only when debug extensions are enabled (CR4[DE] = 1).

- LEN*n* is set to 00b, 01b, or 11b to specify the breakpoint occurs on a byte, word, or doubleword I/O operation, respectively.

The I/O-port address specified by the I/O instruction is zero extended by the processor to 64 bits before comparing it with the DR*n* registers.

When a #DB exception occurs due to an I/O breakpoint in DR*n*, the corresponding B*n* field in DR6 is set to 1 to indicate that a breakpoint condition occurred. The breakpoint occurs after the instruction is executed, which means that the original data is overwritten by the breakpoint instruction. If the debug handler needs to report the previous data value, it must save that value before setting the breakpoint.

Because the breakpoint occurs after the instruction is executed, the address of the instruction following the I/O instruction is pushed onto the debug-handler stack, in most cases. In the case of INS and OUTS instructions that use the repeat prefix, however, the breakpoint occurs after the first iteration of the repeat loop. When this happens, the I/O-instruction address can be pushed onto the stack during a #DB exception if the repeat loop is not complete. A subsequent return from the debug handler causes the next I/O iteration to be executed. If the breakpoint condition is still set, the #DB exception reoccurs after that iteration is complete.

13.1.3.4 Task-Switch Breakpoints

Breakpoints can be set in a task TSS to raise a #DB exception after a task switch. Software enables a task breakpoint by setting the T bit in the TSS to 1. When a task switch occurs into a task with the T bit set, the processor completes loading the new task state. Before the first instruction is executed, the #DB exception occurs, and the processor sets DR6[BT] to 1, indicating that the #DB exception occurred as a result of task breakpoint.

The processor does not clear the T bit in the TSS to 0 when the #DB exception occurs. Software must explicitly clear this bit to disable the task breakpoint. Software should never set the T-bit in the debug-handler TSS if a separate task is used for #DB exception handling, otherwise the processor loops on the debug handler.

13.1.3.5 General-Detect Condition

General-detect is a special debug-exception condition that occurs when software running at any privilege level attempts to access any of the DR*n* registers while DR7[GD] is set to 1. When a #DB exception occurs due to the general-detect condition, the processor clears DR7[GD] and sets DR6[BD] to 1. Clearing DR7[GD] allows the debug handler to access the DR*n* registers without causing infinite #DB exceptions.

A debugger enables general detection to prevent other software from accessing and interfering with the debug registers while they are in use by the debugger. The exception is taken before executing the MOV DR*n* instruction so that the DR*n* contents are not altered.

13.1.4 Single Stepping

Single-step breakpoints are enabled by setting the rFLAGS[TF] bit to 1. TF may be set by the IRET, POPF or SYSRET instructions, with an IRET executed by a debugger being the typical use case. When IRET sets TF, it causes a #DB exception to be taken immediately *after* the *target* of the IRET is executed, returning control to the debugger and thereby single-stepping the target instruction. Setting TF with a POPF instruction also causes a one-instruction delayed #DB exception. When TF is set by SYSRET however, a #DB exception is taken *before* the target instruction executes, hence SYSRET does not provide the single-stepping behavior of IRET.

When a #DB exception occurs due to single stepping, the processor clears rFLAGS[TF] before entering the debug handler, so that the debug handler itself is not single stepped. The processor also sets DR6[BS] to 1, which indicates that the #DB exception occurred as a result of single stepping. The rFLAGS image pushed onto the debug-handler stack has the TF bit set, and single stepping resumes when a subsequent IRET pops the stack image into the rFLAGS register.

Single-step breakpoints have a higher priority than external interrupts. If an external interrupt occurs during single stepping, control is transferred to the #DB handler first, causing the rFLAGS[TF] bit to be cleared. Next, before the first instruction in the debug handler is executed, the processor transfers control to the pending-interrupt handler. This allows external interrupts to be handled outside of single-step mode.

The INT*n*, INT3, and INTO instructions clear the rFLAGS[TF] bit when they are executed. If a debugger is used to single-step software that contains these instructions, it must emulate them instead of executing them.

The single-step mechanism can also be set to single step only control transfers, rather than single step every instruction. See “Single Stepping Control Transfers” on page 399 for additional information.

13.1.5 Breakpoint Instruction (INT3)

The INT3 instruction, or the INT*n* instruction with an operand of 3, can be used to set breakpoints that transfer control to the breakpoint-exception (#BP) handler rather than the debug-exception handler. When a debugger uses the breakpoint instructions to set breakpoints, it does so by replacing the first bytes of an instruction with the breakpoint instruction. The debugger replaces the breakpoint instructions with the original-instruction bytes to clear the breakpoint.

INT3 is a single-byte instruction while INT*n* with an operand of 3 is a two-byte instruction. The instructions have slightly different effects on the breakpoint exception-handler stack. See “#BP—Breakpoint Exception (Vector 3)” on page 240 for additional information on this exception.

13.1.6 Control-Transfer Breakpoint Features

A control transfers is accomplished by using one of following instructions:

- JMP, CALL, RET
- J*cc*, J*r*CXZ, LOOP*cc*

- JMPF, CALLF, RETF
- INT*n*, INT 3, INTO, ICEBP
- Exceptions, IRET
- SYSCALL, SYSRET, SYSENTER, SYSEXIT
- INTR, NMI, SMI, RSM

13.1.6.1 Recording Control Transfers

Software enables control-transfer recording by setting DebugCtl[LBR] to 1. When this bit is set, the processor updates the recording MSRs automatically when control transfers occur:

- *LastBranchFromIP and LastBranchToIP Registers*—On branch instructions, the LastBranchFromIP register is loaded with the segment offset of the branch instruction, and the LastBranchToIP register is loaded with the first instruction to be executed after the branch. On interrupts and exceptions, the LastBranchFromIP register is loaded with the segment offset of the interrupted instruction, and the LastBranchToIP register is loaded with the offset of the interrupt or exception handler.
- *LastIntFromIP and LastIntToIP Registers*—The processor loads these from the LastBranchFromIP register and the LastBranchToIP register, respectively, when most interrupts and exceptions are taken. These two registers are not updated, however, when #DB or #MC exceptions are taken, or the ICEBP instruction is executed.

The processor automatically disables control-transfer recording when a debug exception (#DB) occurs by clearing DebugCtl[LBR] to 0. The contents of the control-transfer recording MSRs are not altered by the processor when the #DB occurs. Before exiting the debug-exception handler, software can set DebugCtl[LBR] to 1 to re-enable the recording mechanism.

Debuggers can trace a control transfer backward from a bug to its source using the recording MSRs and the breakpoint-address registers. The debug handler does this by updating the breakpoint registers from the recording MSRs after a #DB exception occurs, and restarting the program. The program takes a #DB exception on the previous control transfer, and this process can be repeated. The debug handler cannot simply copy the contents of the recording MSR into the breakpoint-address register. The recording MSRs hold segment offsets, while the debug registers hold virtual (linear) addresses. The debug handler must calculate the virtual address by reading the code-segment selector (CS) from the interrupt-handler stack, then reading the segment-base address from the CS descriptor, and adding that base address to the offset in the recording MSR. The calculated virtual-address can then be used as a breakpoint address.

13.1.6.2 Single Stepping Control Transfers

Software can enable control-transfer single stepping by setting DebugCtl[BTF] to 1 and rFLAGS[TF] to 1. The processor automatically disables control-transfer single stepping when a debug exception (#DB) occurs by clearing DebugCtl[BTF]. rFLAGS[TF] is also cleared when a #DB exception occurs. Before exiting the debug-exception handler, software must set both DebugCtl[BTF] and rFLAGS[TF] to 1 to restart single stepping.

When enabled, this single-step mechanism causes a #DB exception to occur on every branch instruction, interrupt, or exception. Debuggers can use this capability to perform a “coarse” single step across blocks of code (bound by control transfers), and then, as the problem search is narrowed, switch into a “fine” single-step mode on every instruction (DebugCtl[BTF] = 0 and rFLAGS[TF] = 1).

Debuggers can use both the single-step mechanism and recording mechanism to support full backward and forward tracing of control transfers.

13.1.7 Debug Breakpoint Address Masking

The Breakpoint Address Extension feature extends the DR[0-3] breakpoint capabilities. Processors with this extension support address mask registers corresponding to each of the DR[0-3] breakpoint registers, in the form of DR[0-3]_ADDR_MASK MSRs. These masks may be used to increase the range of breakpoints by excluding address bits from the breakpoint match. Each bit set to one excludes the corresponding address bit from the breakpoint comparison. Mask bits 11:0 may be used to expand instruction fetch breakpoint ranges up to a 4KB page, while mask bits 31:12 have no effect on instruction breakpoints. For DR0 only, the full mask field (31:0) may be used to qualify data breakpoint matches. This extension is signified by CPUID Fn8000_0001_ECX[26]=1.

An additional extension expands the data breakpoint masking capability of DR0 to the other breakpoint registers, and extends instruction breakpoint masking to bits 31:0 for all registers. This is signified by CPUID Fn8000_0001_ECX[30]=1.

13.2 Performance Monitoring Counters

The AMD64 architecture supports a set of hardware-based performance-monitoring counters (PMCs) that can be utilized to measure the frequency or duration of certain hardware events. MSRs allow the selection of events to be monitored and include a set of corresponding counter registers that accumulate a count of monitored events.

Software tools can use these counters to identify performance bottlenecks, such as sections of code that have high cache-miss rates or frequently mispredicted branches. This information can then be used as a guide for improving overall performance or eliminating performance problems through software optimizations or hardware-design improvements.

Software performance analysis tools often require a means to time-stamp an event or measure elapsed time between two events. The time-stamp counter provides this capability. See Section 13.2.4 “Time-Stamp Counter” on page 407.

The registers used in support of performance monitoring are model-specific registers (MSRs). See “Model-Specific Registers (MSRs)” on page 59 for a general discussion of MSRs and “Performance-Monitoring MSRs” on page 673 for a listing of the performance-monitoring MSR numbers and their reset values.

13.2.1 Performance Counter MSRs

The legacy architecture defines four performance counters (PerfCtr*n*) and corresponding event-select registers (PerfEvtSel*n*). Extensions add northbridge and L2 cache performance monitoring counters. Each *PerfCtr register counts events selected by the corresponding *PerfEvtSel register.

An architectural extension augments the number of performance and event-select registers by adding two more processor counter / event-select pairs. Further extensions add four counter / event-select pairs dedicated to counting northbridge (NB) events and four counter / event-select pairs dedicated to counting L2 cache (L2I) events.

Core logic includes instruction execution pipelines, execution units, and caches closest to the execution hardware. The NB includes logic that routes data traffic between caches, external I/O devices, and a system memory controller which reads and writes system memory (usually implemented as external DRAM). The L2 cache is a cache that is further away from the processor core than the L1 cache or caches. This cache is normally larger than the L1 cache(s) and requires more processor cycles to access. An L2 cache may be shared between physical processor cores.

All implementations support the base set of four performance counter / event-select pairs. Support for the extended performance monitoring registers and the performance-related events selectable via the *PerfEvtSel registers vary by implementation and are described in the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* for that processor.

Core performance counters are used to count processor core events, such as data-cache misses, or the duration of events, such as the number of clocks it takes to return data from memory after a cache miss. During event counting, hardware increments a counter each time it detects an occurrence of a specified event. During duration measurement, hardware counts the number of processor clock cycles required to complete a specific hardware function.

NB performance counters are used to count events that occur within the northbridge. The L2I performance counters are used to count events associated with accessing the L2 cache.

Performance counters and event-select registers are implemented as machine-specific registers (MSRs). The base set of four PerfCtr and PerfEvtSel registers are accessed via a legacy set of MSRs and the extended set of six core PerfCtr / PerfEvtSel registers are accessed via a different set. Extended core PerfCtr / PerfEvtSel registers 0–3 alias the legacy set.

Support for the extended set of core PerfCtr registers and associated PerfEvtSel registers, as well as the sets of northbridge and L2 cache counter / event-select pairs are indicated by CPUID feature bits. See "Detecting Hardware Support for Performance Counters" on page 407. The MSR address assignments for the legacy and extended performance / event-select pairs are listed in Appendix A, Section A.6, "Performance-Monitoring MSRs" on page 673.

The length, in bits, of the performance counters is implementation-dependent, but the maximum length supported is 64 bits. Figure 13-6 shows the format of the performance counter registers.

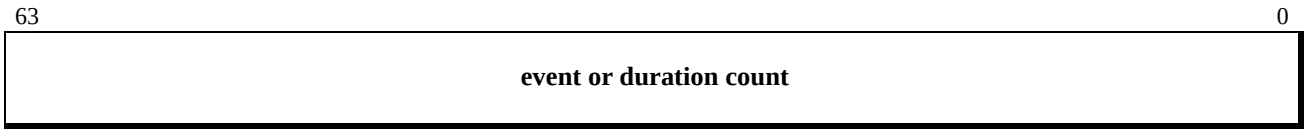


Figure 13-6. Performance Counter Format

For a given processor, all implemented performance counter registers can be read and written by system software running at CPL = 0 using the RDMSR and WRMSR instructions, respectively. The architecture also provides an instruction, RDPMC, which may be employed by user-mode software to read the architected core, northbridge, and L2 performance counters.

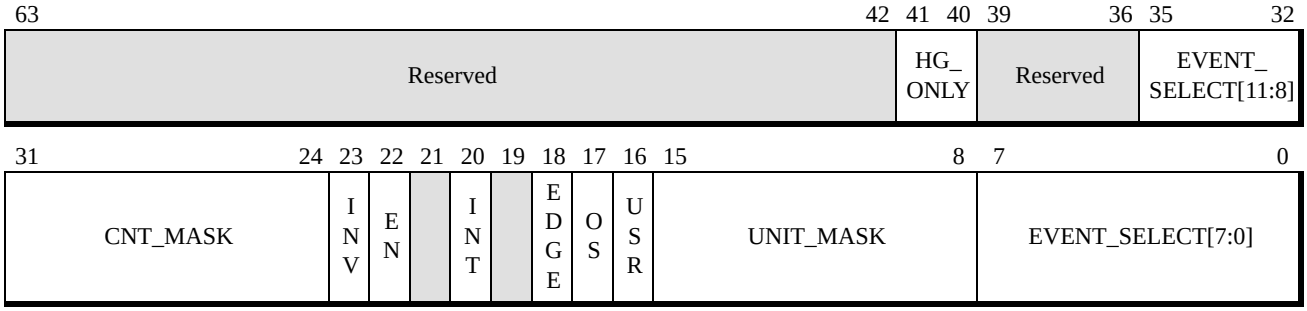
The RDPMC instruction loads the contents of the architected performance counter register specified by the index value contained in the ECX register, into the EDX register and the EAX register. The high 32 bits are returned in EDX, and the low 32 bits are returned in EAX. RDPMC can be executed only at CPL = 0, unless system software enables use of the instruction at all privilege levels. RDPMC can be enabled for use at all privilege levels by setting CR4[PCE] (the *performance-monitor counter-enable* bit) to 1. When CR4[PCE] = 0 and CPL > 0, attempts to execute RDPMC result in a general-protection exception (#GP). For more information on the RDPMC instruction, see the instruction reference page in Volume 3 of this manual.

Writing the performance counters can be useful if software wants to count a specific number of events, and then trigger an interrupt when that count is reached. An interrupt can be triggered when a performance counter overflows (see “Counter Overflow” on page 407 for additional information). Software should use the WRMSR instruction to load the count as a two’s-complement negative number into the performance counter. This causes the counter to overflow after counting the appropriate number of times.

The performance counters are not guaranteed to produce identical measurements each time they are used to measure a particular instruction sequence, and they should not be used to take measurements of very small instruction sequences. The RDPMC instruction is not serializing, and it can be executed out-of-order with respect to other instructions around it. Even when bound by serializing instructions, the system environment at the time the instruction is executed can cause events to be counted before the counter value is loaded into EDX:EAX. The following sections describe the core performance event-select and the northbridge performance event-select registers.

Core Performance Event-Select Registers

The core performance event-select registers (PerfEvtSel*n*) are 64-bit registers used to specify the events counted by the core performance counters, and to control other aspects of their operation. Each performance counter supported by the implementation has a corresponding event-select register that controls its operation. Figure 13-7 below shows the format of the core PerfEvtSel register.



Bits	Mnemonic	Description	R/W
63:42	—	Reserved	
41:40	HG_ONLY	Host/Guest Only	R/W
39:36	—	Reserved	
35:32	EVENT_SELECT[11:8]	Event select bits 11:8	R/W
31:24	CNT_MASK	Counter Mask	R/W
23	INV	Invert Comparison	R/W
22	EN	Counter Enable	R/W
21	—	Reserved	
20	INT	Interrupt Enable	R/W
19	—	Reserved	
18	EDGE	Edge Detect	R/W
17	OS	Operating-System Mode	R/W
16	USR	User Mode	R/W
15:8	UNIT_MASK	Unit Mask	R/W
7:0	EVENT_SELECT[7:0]	Event select bits 7:0	R/W

Figure 13-7. Core Performance Event-Select Register (PerfEvtSel*n*)

The fields shown in Figure 13-7 above are further described below:

- HG_ONLY (Host/Guest Only): read/write. This field qualifies events to be counted based on virtualization operating mode (guest or host). The following table defines how HG_ONLY qualifies the counting of events:

Table 13-3. Host/Guest Only Bits

Host Mode (Bit 41)	Guest Mode (Bit 40)	Events Counted
0	0	All events, irrespective of guest or host mode
0	1	Guest events, if EFER[SVME] = 1
1	0	Host events, if EFER[SVME] = 1
1	1	Guest and host events, if EFER[SVME] = 1

- EVENT_SELECT[11:8] (Event Select): read/write. This field extends the EVENT_SELECT field from 8 bits to 12 bits. See EVENT_SELECT[7:0] below.

- CNT_MASK (Counter Mask): read/write. Used with INV bit to control the counting of multiple events that occur within one clock cycle. The table below describes this:

Table 13-4. Count Control Using CNT_MASK and INV

CNT_MASK	INV	Increment Value
00h	–	Corresponding PerfCtr[*n*] register is incremented by the number of events occurring in a clock cycle. If the number of events is equal to or greater than 32, the count register is incremented by 32.
FFh:01h[1]	0	Corresponding PerfCtr[*n*] register is incremented by 1, if the number of events occurring in a clock cycle is greater than or equal to the CNT_MASK value.
	1	Corresponding PerfCtr[*n*]register is incremented by 1, if the number of events occurring in a clock cycle is less than the CNT_MASK value.
Note 1: Maximum CNT_MASK value (in the range FFh:01h is implementation dependent. Consult applicable BIOS and Kernel Developer's Guide (BKDG) or Processor Programming Reference Manual (PPR) .		

- INV (Invert Comparison): read/write. Used with CNT_MASK field to control the counting of multiple events within one clock cycle. See table above.
- EN (Counter Enable): read/write. Software sets this bit to 1 to enable the PerfEvtSel*n* register, and counting in the corresponding PerfCtr*n* register. Clearing this bit to 0 disables the register pair.
- INT (Interrupt Enable): read/write. Software sets this bit to 1 to enable an interrupt to occur when the performance counter overflows (see "Counter Overflow" on page 407 for additional information). Clearing this bit to 0 disables the triggering of the interrupt.
- EDGE (Edge Detect): read/write. Software sets this bit to 1 to count the number of edge transitions from the negated to asserted state. This feature is useful when coupled with event-duration monitoring, as it can be used to calculate the average time spent in an event. Clearing this bit to 0 disables edge detection.
- OS (Operating-System Mode) and USR (User Mode): read/write. Software uses these bits to control the privilege level at which event counting is performed according to Table 13-5.

Table 13-5. Operating-System Mode and User Mode Bits

OS (Bit 17)	USR (Bit 16)	Event Counting
0	0	No counting.
0	1	Only at CPL > 0.
1	0	Only at CPL = 0.
1	1	At all privilege levels.

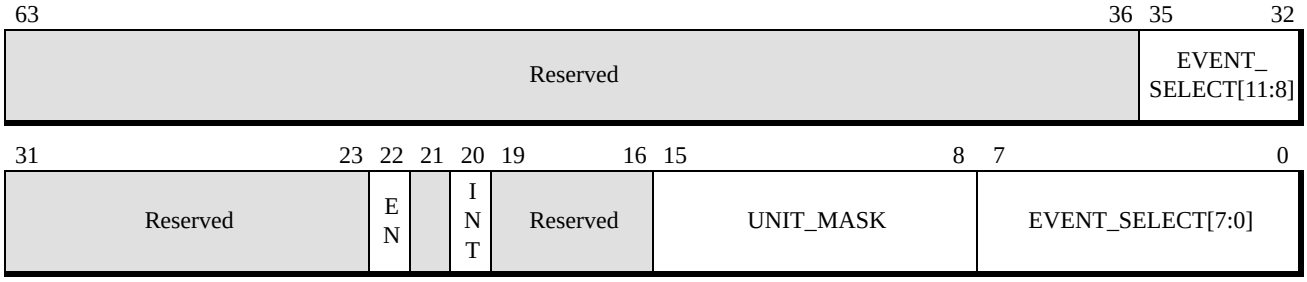
- UNIT_MASK (Unit Mask): read/write. This field further specifies or qualifies the event specified by the EVENT_SELECT field. Depending on implementation, it may be used to specify a sub-event within the class specified by the EVENT_SELECT field or it may act as bit mask and be used to specify a number of events within the class to be monitored simultaneously.

- EVENT_SELECT[7:0] (Event Select [7:0]): read/write. This field concatenated with EVENT_SELECT[11:8] specifies the event or event duration to be counted by the corresponding PerfCtr[*n*] register. The events that can be monitored are implementation dependent. In some implementations, support for a specific EVENT_SELECT value may restricted to a subset of the available performance counters. For more information, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

The core performance event-select registers can be read and written only by system software running at CPL = 0 using the RDMSR and WRMSR instructions, respectively. Any attempt to read or write these registers at CPL > 0 causes a general-protection exception to occur.

Northbridge (NB) Performance Event-Select Registers

The NB performance event-select registers (NB_PerfEvtSel*n*) are 64-bit registers used to specify the events counted by the northbridge performance counters, and to control other aspects of their operation. Each performance counter supported by the implementation has a corresponding event-select register that controls its operation. Figure 13-8 below shows the format of the NB_PerfEvtSel*n* register.



Bits	Mnemonic	Description	R/W
63:36	—	Reserved	
35:32	EVENT_SELECT[11:8]	Event select bits [11:8]	R/W
31:23	—	Reserved	
22	EN	Counter Enable	R/W
21	—	Reserved	
20	INT	Interrupt Enable	R/W
19:16	—	Reserved	
15:8	UNIT_MASK	Unit Mask	R/W
7:0	EVENT_SELECT[7:0]	Event select bits [7:0]	R/W

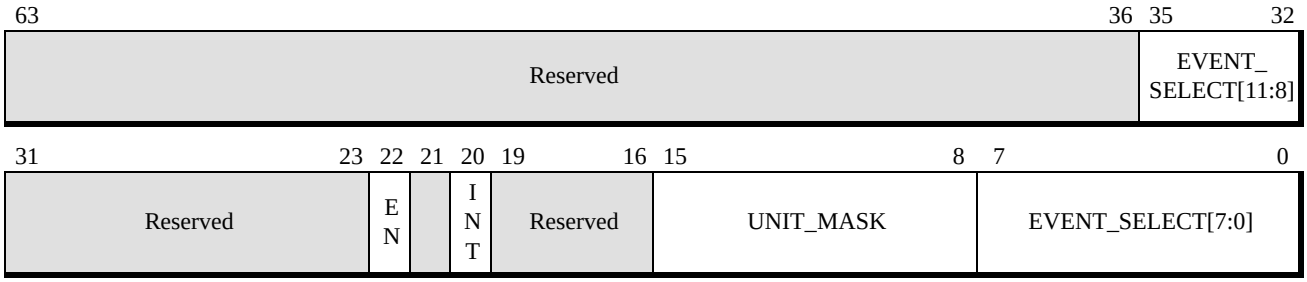
Figure 13-8. Northbridge Performance Event-Select Register (NB_PerfEvtSel*n*)

The northbridge performance event-select registers can be read and written only by system software running at CPL = 0 using the RDMSR and WRMSR instructions, respectively. Any attempt to read or write these registers at CPL > 0 causes a general-protection exception to occur.

For more information on the defined fields within the NB_PerfEvtSel*n* registers, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

L2 Cache (L2I) Performance Event-Select Registers

The L2 cache performance event-select registers (L2I_PerfEvtSel*n*) are 64-bit registers used to specify the events counted by the L2 cache performance counters, and to control other aspects of their operation. Each performance counter supported by the implementation has a corresponding event-select register that controls its operation. Figure 13-9 below shows the format of the L2I_PerfEvtSel*n* register.



Bits	Mnemonic	Description	R/W
63:36	—	Reserved	
35:32	EVENT_SELECT[11:8]	Event select bits [11:8]	R/W
31:23	—	Reserved	
22	EN	Counter Enable	R/W
21	—	Reserved	
20	INT	Interrupt Enable	R/W
19:16	—	Reserved	
15:8	UNIT_MASK	Unit Mask	R/W
7:0	EVENT_SELECT[7:0]	Event select bits [7:0]	R/W

Figure 13-9. L2 Cache Performance Event-Select Register (L2I_PerfEvtSel*n*)

The L2 cache performance event-select registers can be read and written only by system software running at CPL = 0 using the RDMSR and WRMSR instructions, respectively. Any attempt to read or write these registers at CPL > 0 causes a general-protection exception to occur.

For more information on the defined fields within the L2I_PerfEvtSel*n* registers, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product.

Instructions Retired Performance counter

This is a dedicated counter that is always counting instructions retired. It exists at MSR address C000_00E9. It is enabled by setting a 1 to HWCR[30] and its support is indicated by CPUID Fn8000_0008_EBX[1].

13.2.2 Detecting Hardware Support for Performance Counters

Support for extended core, northbridge, and L2 cache performance counters is implementation-dependent. Support on a given processor implementation can be verified using the CPUID instruction.

CPUID Fn8000_0001_ECX[PerfCtrExtCore] = 1 indicates support for the six architecturally defined extended core performance counters and their associated event-select registers. CPUID Fn8000_0001_ECX[PerfCtrExtNB] = 1 indicates support for the four architecturally defined northbridge performance counter / event-select pairs and CPUID Fn8000_0001_ECX[PerfCtrExtL2I] = 1 indicates support for the four architecturally defined L2 cache performance counter / event-select pairs.

See Section 3.3, “Processor Feature Identification,” on page 71 for more information on using the CPUID instruction.

A given processor may implement other performance measurement MSRs with similar capabilities even if one of the optional architected facilities are not.

13.2.3 Using Performance Counters

13.2.3.1 Starting and Stopping

Performance measurement using the PerfCtr*n*, NB_PerfCtr*n*, and L2I_PerfCtr*n* registers is initiated by setting the corresponding *PerfEvtSel*n*[EN] bit to 1. Counting is stopped by clearing the *PerfEvtSel*n*[EN] bit. Software must initialize the remaining *PerfEvtSel*n* fields with the appropriate setup information before or at the same time EN is set. Counting begins when the WRMSR instruction that sets *PerfEvtSel*n*[EN] to 1 completes execution. Counting stops when the WRMSR instruction that clears the EN bit completes execution.

13.2.3.2 Counter Overflow

Some processor implementations support an interrupt-on-overflow capability that allows an interrupt to occur when one of the *PerfCtr*n* registers overflows. The source and type of interrupt is implementation dependent. Some implementations cause a debug interrupt to occur, while others make use of the local APIC to specify the interrupt vector and trigger the interrupt when an overflow occurs. Software enables or disables the triggering of an interrupt on counter overflow by setting or clearing the *PerfEvtSel*n*[INT] bit.

If system software makes use of the interrupt-on-overflow capability, an interrupt handler must be provided that can record information relevant to the counter overflow. Before returning from the interrupt handler, the performance counter can be re-initialized to its previous state so that another interrupt occurs when the appropriate number of events are counted.

13.2.4 Time-Stamp Counter

The time-stamp counter (TSC) is used to count processor-clock cycles. The TSC is cleared to 0 after a processor reset. After a reset, the TSC is incremented at a rate corresponding to the baseline frequency

of the processor (which may differ from actual processor frequency in low power modes of operation). Each time the TSC is read, it returns a monotonically-larger value than the previous value read from the TSC. When the TSC contains all ones, it wraps to zero. The TSC in a 1-GHz processor counts for almost 600 years before it wraps. Figure 13-10 shows the format of the 64-bit time-stamp counter (TSC).

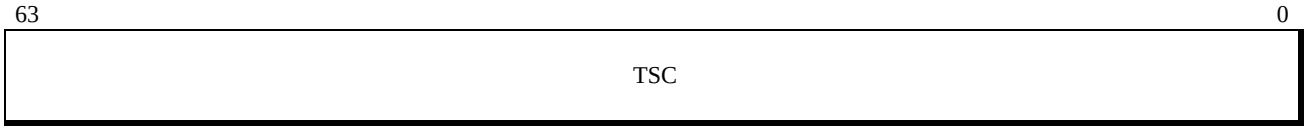


Figure 13-10. Time-Stamp Counter (TSC)

The TSC is a model-specific register that can also be read using one of the special *read time-stamp counter* instructions, RDTSC (Read Time-Stamp Counter) or RDTSCP (Read Time-Stamp Counter and Processor ID). The RDTSC and RDTSCP instructions load the contents of the TSC into the EDX register and the EAX register. The high 32 bits are loaded into EDX, and the low 32 bits are loaded into EAX. The RDTSC and RDTSCP instructions can be executed at any privilege level and from any processor mode. However, system software can disable the RDTSC or RDTSCP instructions for programs that run at CPL > 0 by setting CR4[TSD] (the *time-stamp disable* bit) to 1. When CR4[TSD] = 1 and CPL > 0, attempts to execute RDSTC or RDSTCP result in a general-protection exception (#GP).

The TSC register can be read and written using the RDMSR and WRMSR instructions, respectively. The programmer should use the CPUID instruction to determine whether these features are supported. If EDX bit 4 (as returned by CPUID function 1) is set, then the processor supports TSC, the RDTSC instruction and CR4[TSD]. If EDX bit 27 returned by CPUID function 8000_0001h is set, then the processor supports the RDTSCP instruction.

The TSC register can be used by performance-analysis applications, along with the performance-monitoring registers, to help determine the relative frequency of an event or its duration. Software can also use the TSC to time software routines to help identify candidates for optimization. In general, the TSC should not be used to take very short time measurements, because the resulting measurement is not guaranteed to be identical each time it is made. The RDTSC instruction (unlike the RDTSCP instruction) is not serializing, and can be executed out-of-order with respect to other instructions around it. Even when bound by serializing instructions, the system environment at the time the instruction is executed can cause additional cycles to be counted before the TSC value is loaded into EDX:EAX.

When using the TSC to measure elapsed time, programmers must be aware that for some implementations, the rate at which the TSC is incremented varies based on the processor power management state (Pstate). For other implementations, the TSC increment rate is fixed and is not subject to power-management related changes in processor frequency. CPUID Fn 8000_0007h_EDX[TscInvariant] = 1 indicates that the TSC increment rate is a constant.

For more information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 71.

13.3 Instruction-Based Sampling

Instruction-Based Sampling (IBS) is a hardware facility that can be used to gather specific metrics related to processor instruction fetch and instruction execution activity. Data capture is performed by hardware at a sampling interval specified by values programmed in IBS sampling control registers. The IBS facility can be utilized by software to perform code profiling based on statistical sampling.

There are two independent data gathering components of IBS: instruction fetch sampling and instruction execution sampling. Instruction fetch sampling provides information about instruction address translation look-aside buffer (ITLB) and instruction cache behavior for a randomly selected fetch block, under the control of the IBS Fetch Control Register. Instruction execution sampling provides information about instruction execution behavior by tracking the execution of a single operation (op) that is randomly selected, under the control of the IBS Execution Control Register.

When the programmed interval for fetch sampling has expired, the fetch sampling component of IBS selects and tags the next fetch block. IBS hardware records specific performance information about the tagged fetch. In a similar manner, when the programmed interval for op sampling has expired, the op sampling component of IBS selects and tags the next op being dispatched for execution.

When data collection for the tagged fetch or op is complete, the hardware signals an interrupt. An interrupt handler can then read the performance information that was captured for the fetch or op in IBS MSRs, save it, and re-enable the hardware to take the next sample.

More information about the IBS facility and how software can use it to perform code profiling can be found in the *Software Optimization Guide* for your specific product. The *Software Optimization Guide for AMD Family 15h Processors* is order #47414.

Support for the IBS feature is indicated by the CPUID Fn 8000_0001h_ECX[IBS]. For more information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 71.

13.3.1 IBS Fetch Sampling

When a processor fetches an instruction, it is actually reading a contiguous range of instruction bytes that contains the instruction from memory or from cache. This range of bytes loaded by the processor in one operation is called a *fetch block*. The size and address-alignment characteristics of the fetch block are implementation-dependent. In the following discussion, the term *instruction fetch* or simply *fetch* refers to this operation of reading a fetch block.

Instruction fetch sampling records the following performance information for each tagged fetch:

- If the fetch completed or was aborted
- The number of core clock cycles spent on the fetch

- If the fetch hit or missed the instruction cache
- If the instruction fetch hit or missed the instruction TLBs
- The fetch address translation page size
- The linear and physical address associated with the fetch

IBS selects and tags a fetch at a programmable rate. When enabled by the IBS Fetch Control Register (IbsFetchEn = 1 and IbsFetchVal = 0), an internal 20-bit fetch interval counter increments for every attempted fetch operation. When the value in bits 19:4 of the fetch counter equal the value in the IbsFetchMaxCnt field of the IBS Fetch Control Register, the next fetch block is tagged for data collection.

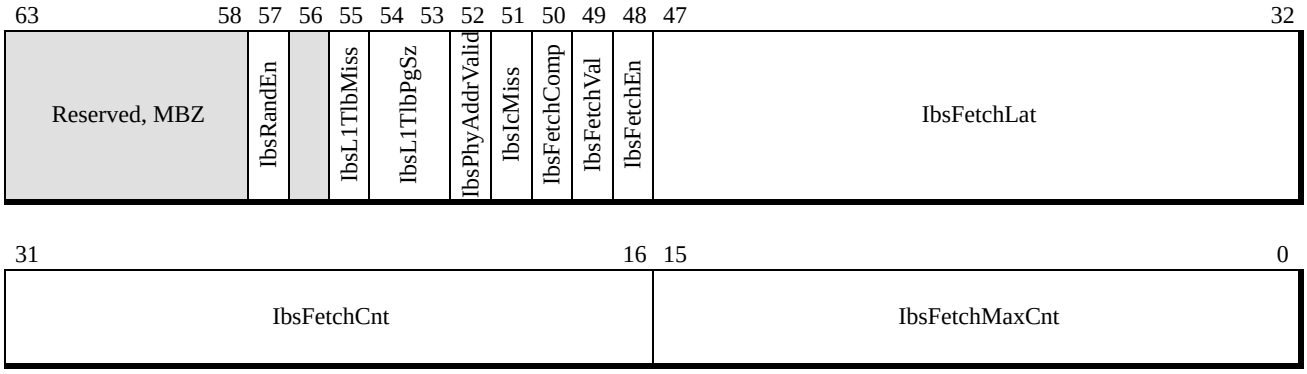
When the tagged fetch completes or is aborted, the status of the fetch is written to the IBS Fetch Control Register and the associated linear address and physical address are written in the IBS Fetch Linear Address Register and IBS Fetch Physical Address Register, respectively. The IbsFetchVal bit is set in the IBS Fetch Control Register and an interrupt is generated as specified by the local APIC.

The interrupt service routine saves the performance information stored in the IBS fetch registers. Software can then initiate another sample by resetting the IbsFetchVal bit in the IBS Fetch Control Register. Hardware initializes bits 19:4 of the internal fetch interval counter with the value in the IbsFetchCnt field. If the IbsFetchCtl[IbsRandEn] bit is set, bits 3:0 of the fetch interval counter are re-initialized by hardware with a pseudo-random value; otherwise bits 3:0 are cleared.

13.3.2 IBS Fetch Sampling Registers

The IBS fetch sampling registers consist of the status and control register (IBS Fetch Control Register) and the associated fetch address registers (IBS Fetch Linear Address Register and IBS Fetch Physical Address Register). The IBS fetch sampling registers are accessed using the RDMSR and WRMSR instructions.

IBS Fetch Control Register



Bit(s)	Field Mnemonic	Descriptive Name	R/W
63:58	—	Reserved, MBZ	
57	IbsRandEn	IBS Randomize Tagging Enable	R/W
56	—	Reserved	
55	IbsL1TlbMiss	IBS Fetch L1 TLB Miss	R/W
54:53	IbsL1TlbPgSz	IBS Fetch L1 TLB Page Size	R/W
52	IbsPhyAddrValid	IBS Fetch Physical Address Valid	R/W
51	IbsIcMiss	IBS Instruction Cache Miss	R/W
50	IbsFetchComp	IBS Fetch Complete	R/W
49	IbsFetchVal	IBS Fetch Valid	R/W
48	IbsFetchEn	IBS Fetch Enable	R/W
47:32	IbsFetchLat	IBS Fetch Latency	R/W
31:16	IbsFetchCnt	IBS Fetch Count	R/W
15:0	IbsFetchMaxCnt	IBS Fetch Maximum Count	R/W

Figure 13-11. IBS Fetch Control Register(IbsFetchCtl)

The fields shown in Figure 13-11 are further described below:

- IbsRandEn (IBS Randomize Tagging Enable)—Bit 57, read/write. Software sets this bit to 1 to add variability to the interval at which fetch operations are selected for tagging. When set, bits 3:0 of the fetch interval counter are set to a pseudo-random value when the IbsFetchCtl register is written. Clearing this bit causes bits 3:0 of the fetch interval counter to be reset to zero.
- IbsL1TlbMiss (IBS Fetch L1 TLB Miss)—Bit 55, read/write. This bit is set if the tagged fetch missed in the L1 TLB.
- IbsL1TlbPgSz[1:0] (IBS Fetch L1 TLB Page Size)—Bits 54:53, read/write. This field indicates the page size of the translation in the L1 TLB for the tagged fetch. This field is valid only if IbsPhyAddrVal = 1. The table below defines the encoding of this two-bit field:

Value	Page Size
00b	4 Kbyte
01b	2 Mbyte
10b	1 Gbyte
11b	Reserved

  Some implementations might not support all page sizes. Note: The page size in the L1 TLB might not match the page size in the page table.
- IbsPhyAddrValid (IBS Fetch Physical Address Valid)—Bit 52, read/write. This bit is set if the physical address of the tagged fetch is valid. When this bit is set, the IbsL1TlbPgSz field and the contents of the IBS Fetch Physical Address Register (see definition of this register below) are both valid.
- IbsIcMiss (IBS Instruction Cache Miss)—Bit 51, read/write. This bit is set if the tagged fetch missed in the instruction cache.

- IbsFetchComp (IBS Fetch Complete)—Bit 50, read/write. This bit is set if the tagged fetch completes and data is available for use by the instruction decoder.
- IbsFetchVal (IBS Fetch Valid)—Bit 49, read/write. This bit is set if the tagged fetch either completes or is aborted. When the bit is set, captured data for the tagged fetch is available and the fetch interval counter stops. An interrupt is generated as specified by the local APIC. The interrupt handler should read and save the captured performance data before clearing the bit.
- IbsFetchEn (IBS Fetch Enable)—Bit 48, read/write. Software sets this bit to enable fetch sampling. Clearing this bit to 0 disables fetch sampling.
- IbsFetchLat[15:0] (IBS Fetch Latency)—Bits 47:32, read/write. This 16-bit field indicates the number of core clock cycles from the initiation of the fetch to the delivery of the instruction bytes to the core. If the fetch is aborted before it completes, this field returns the number of clock cycles from the initiation of the fetch to its abortion.
- IbsFetchCnt[15:0] (IBS Fetch Count)—Bits 31:16, read/write. This 16-bit field returns the current value of bits 19:4 of the fetch interval counter on a read. Bits 19:4 of the fetch interval counter are set to this value on a write.
- IbsFetchMaxCnt[15:0] (IBS Fetch Maximum Count)—Bits 15:0, read/write. This 16-bit field specifies the maximum count value of bits 19:4 of the fetch interval counter. When the value in bits 19:4 of the fetch counter equals the value in this field, the next fetch block is tagged for profiling.

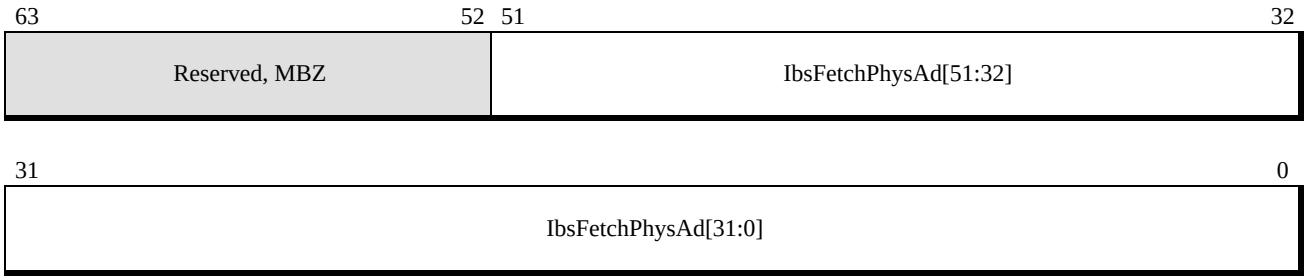
IBS Fetch Linear Address Register

Bit(s)	Field Mnemonic	Descriptive Name	R/W
63:0	IbsFetchLinAd	IBS Fetch Linear Address	RO

Figure 13-12. IBS Fetch Linear Address Register (IbsFetchLinAd)

This is a read-only register. Reading the IbsFetchLinAd MSR returns the 64-bit linear address of the tagged fetch. This address may correspond to the first byte of an AMD64 instruction or the start of the fetch block. The address is valid only if the IbsFetchVal bit is set. The address is in canonical form.

IBS Fetch Physical Address Register



Bit(s)	Field Mnemonic	Descriptive Name	R/W
63:52	—	Reserved, MBZ	n/a
51:0	IbsFetchPhysAd	IBS Fetch Physical Address	RO

Figure 13-13. IBS Fetch Physical Address Register (IbsFetchPhysAd)

This is a read-only register. Reading the IbsFetchPhysAd MSR returns the 52-bit physical address of the tagged fetch. This address may correspond to the first byte of an AMD64 instruction or the start of the fetch block. The address is valid only if both the IbsPhyAddrValid and the IbsFetchVal bits of the IbsFetchCtl register are set. Otherwise, the contents of this register are undefined. The indicated size of 52 bits is an architectural limit. Specific processors may implement fewer bits.

13.3.3 IBS Execution Sampling

Instruction execution performance is measured by tagging an op associated with an instruction. The tagged op joins other ops in a queue waiting to be dispatched and executed. Instructions that decode to more than one op may return different performance data depending upon which op associated with the instruction is tagged. IBS returns the following performance information for each retired tagged op:

- Branch status for branch ops.
- For a load or store op:
  - Whether the load or store missed in the data cache.
  - Whether the load or store address hit or missed in the TLBs.
  - The linear and physical address of the data operand associated with the load or store operation.
  - Source information for cache, DRAM, MMIO, or I/O accesses.

IBS selects and tags an op at a programmable rate. When enabled by the IBS Execution Control Register (IbsOpEn = 1 and IbsOpVal = 0), an internal 27-bit op interval counter increments either once for every core clock cycle, if IbsOpCntCtl is cleared, or once for every dispatched op, if IbsOpCntCtl is set.

When the value in bits 26:4 of the op counter equals the value in the IbsOpMaxCnt field of the IBS Execution Control Register, an op is tagged in the next cycle. When the op is retired, the execution status of the op is written to the IBS execution registers, and IbsOpVal bit of the IBS Execution Control

Register is set. When this is complete, an interrupt is signalled to the local APIC. The local APIC should be programmed to deliver this interrupt to the processor core.

The interrupt service routine must save the performance information stored in IBS execution registers. Software can then initiate another sample by resetting the IbsOpVal bit in the IBS Execution Control Register.

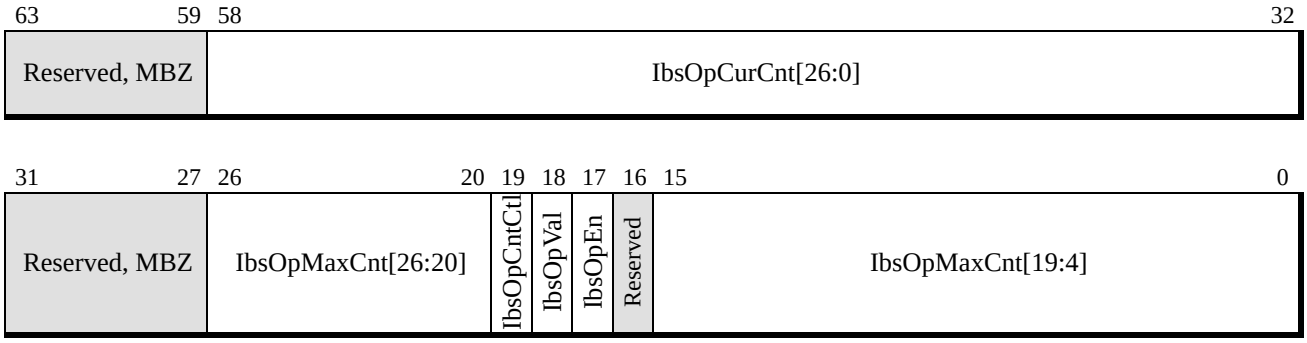
Aborted ops do not produce an IBS execution sample. If the tagged op aborts (i.e., does not retire), hardware resets bits 26:7 of the op interval counter to zero, and bits 6:0 to a random value. The op counter continues to increment and another op is selected when the value in bits 26:4 of the op interval counter equals the value in the IbsOpMaxCnt field.

Randomization of sampling interval: A degree of randomization of the sampling interval is necessary to ensure fairness in sampling, especially for loop-intensive code. For execution sampling this must be done by software. This is accomplished when writing the IbsOpCtl register to clear the IbsOpVal bit and initiate a new sampling interval. At that time software can provide a small random number (4-6 bits) in the IbsOpCurCnt field to offset the starting count, thereby randomizing the point at which the count reaches the IbsOpMaxCnt value and triggers a sample.

13.3.4 IBS Execution Sampling Registers

The IBS execution sampling registers consist of the control register (IBS Execution Control Register), the linear address register (IBS Op Linear Address Register), and three execution data registers (IBS Op Data 1–3). The IBS execution sampling registers are accessed using the RDMSR and WRMSR instructions.

IBS Execution Control Register (IbsOpCtl)



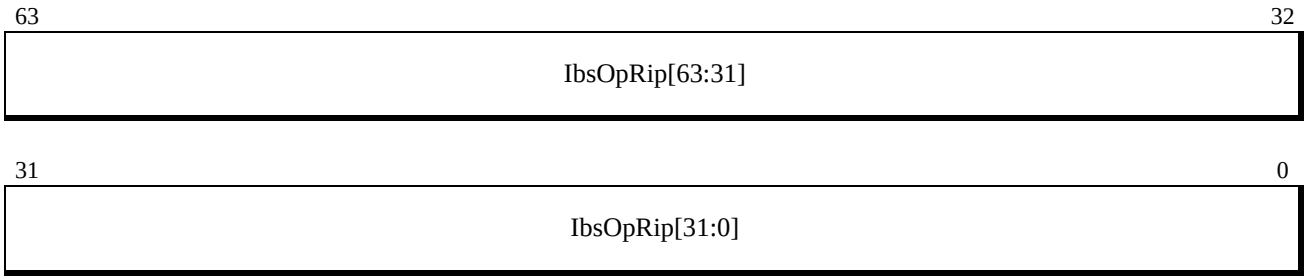
Bit(s)	Field Mnemonic	Descriptive Name	R/W
63:59	—	Reserved, MBZ	
58:32	IbsOpCurCnt[26:0]	IBS Op Current Count, bits 26:0	R/W
31:27	—	Reserved, MBZ	
26:20	IbsOpMaxCnt[26:20]	IBS Op Maximum Count, bits 26:20	R/W
19	IbsOpCntCtl	IBS Op Counter Control	R/W
18	IbsOpVal	IBS Op Sample Valid	R/W
17	IbsOpEn	IBS Op Sampling Enable	R/W
16	—	Reserved, MBZ	
15:0	IbsOpMaxCnt[19:4]	IBS Op Maximum Count, bits 19:4	R/W

Figure 13-14. IBS Execution Control Register (IbsOpCtl)

The fields shown in Figure 13-14 are further described below:

- IbsOpCurCnt[26:0] (IBS Op Current Count)—Bits 58:32, read/write. This field returns the current value of the op counter, and provides the starting value of the counter when software writes this register to clear the IbsOpVal bit and start another sampling interval.
- IbsOpMaxCnt[26:20] (IBS Op Maximum Count[26:20])—Bits 26:20, read/write. This field is used to specify the most significant 7 bits of the IbsOpMaxCnt.
- IbsOpCntCtl (IBS Op Counter Control)—Bit 19, read/write. This bit controls op tagging. When this bit is zero, IBS counts core clock cycles in order to select an op for tagging. When this bit is one, IBS counts dispatched ops in order to select an op for tagging.
- IbsOpVal (IBS Op Sample Valid)—Bit 18, read/write. This bit is set when a tagged op retires and indicates that new instruction execution data is available. The op counter stops counting. An interrupt is generated as specified by the local APIC. The software interrupt handler captures the performance data before clearing the bit to enable the hardware to take another sample.
- IbsOpEn (IBS Op Sample Enable)—Bit read/write. Software sets this bit to enable IBS execution sampling. Clearing this bit disables IBS execution sampling.
- IbsOpMaxCnt[19:4] (IBS Op Maximum Count[19:4]): read/write. This field specifies the maximum count value for bits 19:4 of the op interval counter. When the value in bits 26:4 of the op interval counter equal the value specified by the concatenation of the IbsOpMaxCnt[26:20] field with this field, the next op is tagged for profiling.

IBS Op Linear Address Register (IbsOpRip)



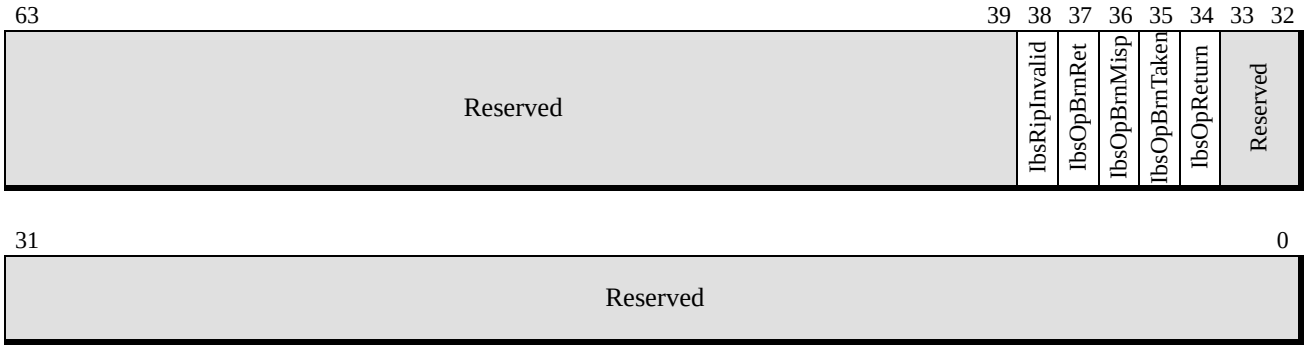
Bit(s)	Field Mnemonic	Descriptive Name	R/W
63:0	IbsOpRip	IBS Op Linear Address	R/W

Figure 13-15. IBS Op Linear Address Register (IbsOpRip)

IbsOpRip[63:0] (IBS Op Linear Address): read/write. Specifies the linear address for the instruction from which the tagged op was issued. The address is valid only if the IbsOpCtl[IbsOpVal] bit is set and the IbsOpData1[IbsRipInvalid] bit is cleared. The address is in canonical form.

IBS Op Data 1 Register (IbsOpData1)

The IBS Op Data 1 Register provides core cycle counts for tagged ops and performance data for tagged ops which perform a branch.



Bit(s)	Field Mnemonic	Descriptive Name	R/W
63:39	—	Reserved	
38	IbsRipInvalid	IbsOpRip Register Invalid	R/W
37	IbsOpBrnRet	IBS Op Branch Retired	R/W
36	IbsOpBrnMisp	IBS Op Branch Mispredicted	R/W
35	IbsOpBrnTaken	IBS Op Branch Taken	R/W
34	IbsOpReturn	IBS Op RET	R/W
33:0	—	Reserved	

Figure 13-16. IBS Op Data 1 Register (IbsOpData1)

The fields shown in Figure 13-16 are further described below:

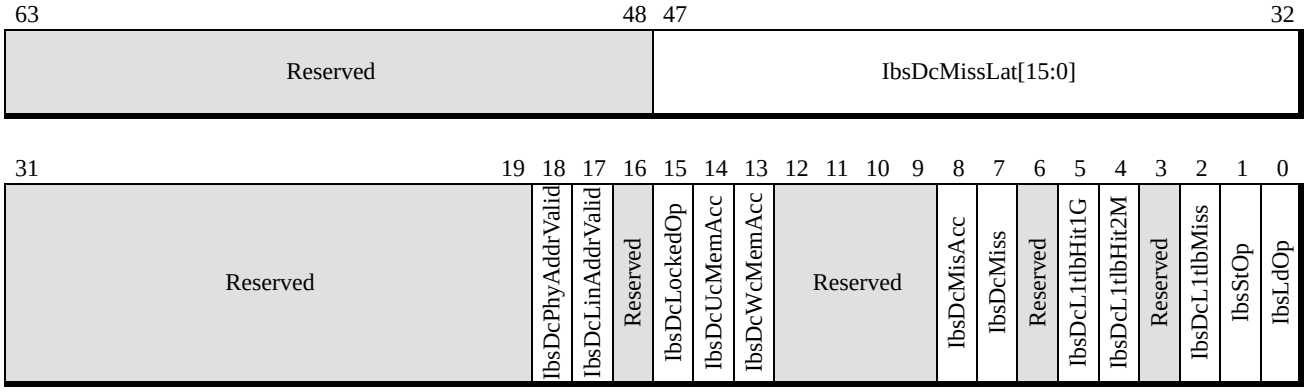
- IbsRipInvalid (IbsOpRip Register Invalid)—Bit 38, read/write. If this bit is set, the contents of the IbsOpRip register are not valid.
- IbsOpBrnRet (IBS Op Branch Retired)—Bit 37, read/write. This bit is set if the tagged op performs a branch that retired.
- IbsOpBrnMisp (IBS Op Branch Mispredicted)—Bit 36, read/write. This bit is set if the tagged op performs a retired mispredicted branch.
- IbsOpBrnTaken (IBS Op Branch Taken)—Bit 35, read/write. This bit is set if the tagged op performs a retired taken branch.
- IbsOpReturn (IBS Op RET)—Bit 34, read/write. This bit is set if the tagged op performs a retired subroutine return (RET).

IBS Op Data 2 Register (IbsOpData2)

The IBS Op Data 2 Register captures northbridge-related performance data. The information captured is implementation-dependent. See the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual (PPR)* applicable to your product for details.

IBS Op Data 3 Register (IbsOpData3)

Data Cache (first-level cache) performance data is captured in IBS Op Data 3 Register. If a load or store operation crosses a 128-bit boundary, the data returned in this register is the data for the access to the data below the 128-bit boundary.



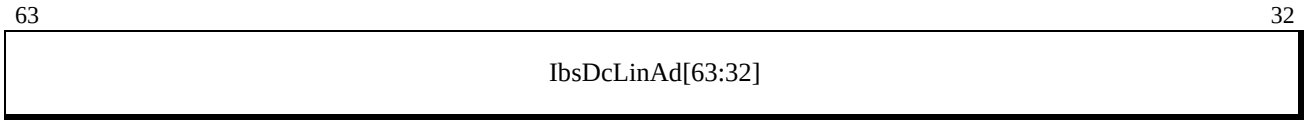
Bit(s)	Field Mnemonic	Descriptive Name	R/W
63:48	—	Reserved	
47:32	IbsDcMissLat[15:0]	IBS Data Cache Miss Latency	R/W
31:19	—	Reserved	
18	IbsDcPhyAddrValid	IBS Data Cache Physical Address Valid	R/W
17	IbsDcLinAddrValid	IBS Data Cache Linear Address Valid	R/W
16	—	Reserved	
15	IbsDcLockedOp	IBS Data Cache Locked Op	R/W
14	IbsDcUcMemAcc	IBS Data Cache UC Memory Access	R/W
13	IbsDcWcMemAcc	IBS Data Cache WC Memory Access	R/W
12:9	—	Reserved	
8	IbsDcMisAcc	IBS Data Cache Misaligned Access Penalty	R/W
7	IbsDcMiss	IBS Data Cache Miss	R/W
6	—	Reserved	
5	IbsDcL1tlbHit1G	IBS Data Cache L1 TLB Hit 1-Gbyte Page	R/W
4	IbsDcL1tlbHit2M	IBS Data Cache L1 TLB Hit 2-Mbyte Page	R/W
3	—	Reserved	
2	IbsDcL1tlbMiss	IBS Data Cache L1 TLB Miss	R/W
1	IbsStOp	IBS Store Operation	R/W
0	IbsLdOp	IBS Load Operation	R/W

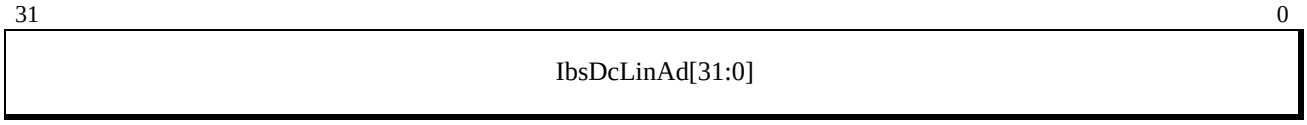
Figure 13-17. IBS Op Data 3 Register (IbsOpData3)

The fields shown in Figure 13-17 are further described below:

- IbsDcMissLat[15:0] (IBS Data Cache Miss Latency)—Bits 47:32, read/write. This field indicates the number of core clock cycles from when a miss is detected in the data cache to when the data is delivered to the core. The value is not valid for data cache store operations.
- IbsDcPhyAddrValid (IBS Data Cache Physical Address Valid)—Bit 18, read/write. This bit is set if the physical address in the IBS DC Physical Address Register is valid for a load or store operation.
- IbsDcLinAddrValid (IBS Data Cache Linear Address Valid)—Bit 17, read/write. This bit is set if the linear address in the IBS DC Linear Address Register is valid for a load or store operation.
- IbsDcLockedOp (IBS Data Cache Locked Op)—Bit 15, read/write. This bit is set if the tagged load or store operation was a locked operation.
- IbsDcUcMemAcc (IBS Data Cache UC Memory Access)—Bit 14, read/write. This bit is set if the tagged load or store operation accessed uncacheable memory.
- IbsDcWcMemAcc (IBS Data Cache WC Memory Access)—Bit 13, read/write. This bit is set if the tagged load or store operation accessed write combining memory.
- IbsDcMisAcc (IBS Data Cache Misaligned Access Penalty)—Bit 8, read/write. This bit is set if a tagged load or store operation incurred a performance penalty due to a misaligned access.
- IbsDcMiss (IBS Data Cache Miss)—Bit 7, read/write. This bit is set if the cache line used by the tagged load or store operation was not present in the data cache.
- IbsDcL1tlbHit1G (IBS Data Cache L1 TLB Hit 1-Gbyte Page)—Bit 5, read/write. This bit is set if the physical address for the tagged load or store operation was present in a 1-Gbyte page table entry in the data cache L1 TLB.
- IbsDcL1tlbHit2M (IBS Data Cache L1 TLB Hit 2-Mbyte Page)—Bit 4, read/write. This bit is set if the physical address for the tagged load or store operation was present in a 2-Mbyte page table entry in the data cache L1 TLB.
- IbsDcL1tlbMiss (IBS Data Cache L1 TLB Miss)—Bit 2, read/write. This bit is set if the physical address for the tagged load or store operation was not present in the data cache L1 TLB.
- IbsStOp (IBS Store Op)—Bit 1, read/write. This bit is set if the tagged op was a store.
- IbsLdOp (IBS Load Op)—Bit 0, read/write. This bit is set if the tagged op was a load.

IBS Data Cache Linear Address Register (IbsDcLinAd)



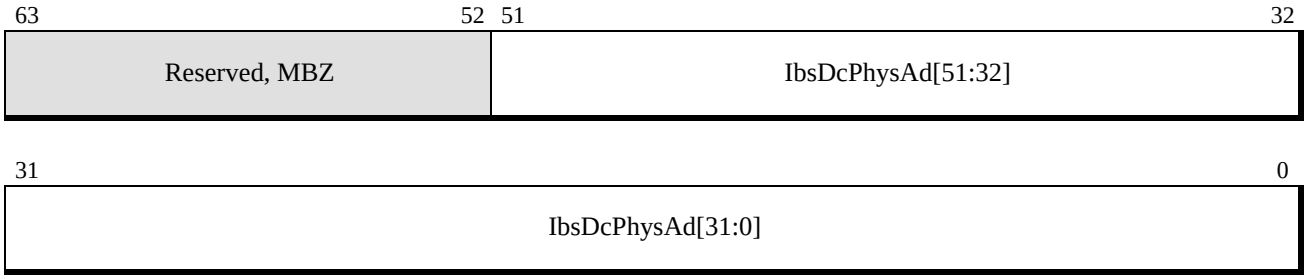


Bit(s)	Field Mnemonic	Descriptive Name	R/W
63:0	IbsDcLinAd	IBS Data Cache Linear Address	R/W

Figure 13-18. IBS Data Cache Linear Address Register (IbsDcLinAd)

IbsDcLinAd[63:0] (IBS Data Cache Linear Address): read/write. Specifies the linear address of the tagged op's memory operand. The address is valid only if IbsOpData3[IbsDcLinAdVal] is set. The address is in canonical form.

IBS Data Cache Physical Address Register (IbsDcPhysAd)



Bit(s)	Field Mnemonic	Descriptive Name	R/W
63:52	—	Reserved, MBZ	
51:0	IbsDcPhysAd	IBS Data Cache Physical Address	R/W

Figure 13-19. IBS Data Cache Physical Address Register (IbsDcPhysAd)

IbsDcPhysAd (IBS Data Cache Physical Address): read/write. Specifies the physical address of the tagged op's memory operand. The address is valid only if IbsOpData3[IbsDcPhyAdVal] is set.

IBS Branch Target Address Register (IbsBrTarget)

Bit(s)	Field Mnemonic	Descriptive Name	R/W
63:0	IbsBrTarget	IBS branch target linear address	R/W

Figure 13-20. IBS Branch Target Address Register (IbsBrTarget)

IbsBrTarget (IBS Branch Target): read/write. Specifies the 64-bit linear address for the branch target. The address is in canonical form. The branch target address is valid if it is non-zero. For a conditional branch not taken, the value supplied in this register is the fall-through address.

13.4 Lightweight Profiling

Lightweight Profiling (LWP) is an AMD64 extension that allows user mode processes to gather performance data about themselves with very low overhead. LWP is supported in both long mode and legacy mode. Modules such as managed runtime environments and dynamic optimizers can use LWP to monitor the running program with high accuracy and high resolution. They can quickly discover performance problems and opportunities and immediately act on this information.

LWP allows a program to gather performance data and examine it either by polling or by taking an occasional interrupt. It introduces minimal additional state to the CPU and the process. LWP differs from the existing performance counters and from Instruction Based Sampling (IBS) because it collects large quantities of data before taking an interrupt. This substantially reduces the overhead of using performance feedback. An application can avoid the need to enable and process interrupts by polling the LWP data.

A program can control LWP data collection entirely in user mode. It can start, stop, and reconfigure profiling without calling the kernel.

LWP runs within the context of a thread, so it can be used by multiple processes in a system at the same time without interference. This also means that if one thread is using LWP and another is not, the latter thread incurs no profiling overhead.

LWP can be programmed to run in one of two modes: *synchronized mode* or *continuous mode*. In synchronized mode the recording of events stops when the buffer set up to hold event records becomes full. In continuous mode, the storing of events wraps in the buffer overwriting older records.

13.4.1 Overview

When enabled, LWP hardware monitors one or more events during the execution of user-mode code and periodically inserts event records into a ring buffer in the address space of the running process. If performance timestamping is supported and enabled, each event record captured is timestamped using

the value read from the performance timestamp counter (PTSC). Timestamping is enabled by setting the Flags.PTSC bit of the Lightweight Profiling Control Block (LWPCB). When the ring buffer is filled beyond a user-specified threshold, the hardware can cause an interrupt which the operating system (OS) uses to signal a process to empty the ring buffer. With proper OS support, the interrupt can even be delivered to a separate process or thread.

LWP only counts instructions that retire in user mode (CPL = 3). Instructions that change to CPL 3 from some other level are not counted, since the instruction address is not an address in user mode space. LWP does not count hardware events while the processor is in system management mode (SMM) and while entering or leaving SMM.

Once LWP is enabled, each user-mode thread uses the LLWPCB and SLWPCB instructions to control LWP operation. These instructions refer to a data structure in application memory called the Lightweight Profiling Control Block, or LWPCB, to specify the profiling parameters and to interact with the LWP hardware. The LWPCB in turn points to a buffer in memory in which LWP stores event records.

Each thread in a multi-threaded process must configure LWP separately. A thread has its own ring buffer and counters which are context switched with the rest of the thread state. However, a single monitor thread could collect and process LWP data from multiple other threads.

LWP may be set up to run in one of two modes:

- Synchronized Mode

  LWP runs in synchronized mode when it is started with LWPCB.Flags.CONT = 0. In this mode, LWP will not advance the ring buffer pointer when the event ring buffer is full. It simply increments LWPCB.MissedEvents to count the number of missed event records. In synchronized mode, a thread can remove event records from the ring buffer by advancing the ring buffer tail pointer without stopping LWP in the executing thread. If the buffer had been full, event records will again be written and the ring buffer pointer will be advanced.

- Continuous Mode

  LWP runs in continuous mode when it is started with LWPCB.Flags.CONT = 1. In this mode, LWP will store an event record even when the event ring buffer is full, wrapping around in the ring buffer and overwriting the oldest event record. In continuous mode, LWPCB.MissedEvents counts the number of times that such wrapping has occurred. The only reliable way to read events from the ring buffer when LWP is in continuous mode is to stop LWP in the running thread before accessing the LWPCB and the ring buffer contents. Support for continuous mode is indicated by CPUID Fn8000_001C_EAX[LwpCont].

During profiling, the LWP hardware monitors and reports on one or more types of events. Following are the steps in this process:

1. **Count—**Each time an instruction is retired, LWP decrements its internal event counters for all of the events associated with the instruction. An instruction can cause zero, one, or multiple events. For instance, an indirect jump through a pointer in memory counts as an instruction retired, a branch retired, and may also cause up to two DCache misses (or more, if there is a TLB miss) and up to two ICache misses.
   - Some events may have filters or conditions on them that regulate counting. For instance, the application may configure LWP so that only cache miss events with latency greater than a specified minimum are eligible to be counted.
2. **Gather—**When an event counter becomes negative, the event should be reported. LWP gathers an event record and, if enabled, samples the value in the PTSC to be included in the record as the TimeStamp value. The event's counter may continue to count below zero until the record is written to the event ring buffer.

   For most events, such as instructions retired, LWP gathers an event record describing the instruction that caused the counter to become negative. However, it is valid for LWP to gather event record data for the *next* instruction that causes the event, or to take other measures to capture a record. Some of these options are described with the individual events.
   - An implementation can choose to gather event information on one or many events at any one time. If multiple event counters become negative, an advanced LWP implementation might gather one event record per event and write them sequentially. A basic LWP implementation may choose one of the eligible events. Other events continue counting but wait until the first event record is written. LWP picks the next eligible instructions for the waiting events. This situation should be extremely uncommon if software chooses large event interval values.
   - LWP may discard an event occurrence. For instance, if the LWPCB or the event ring buffer needs to be paged in from disk, LWP might choose not to preserve the pending event data. If an event is discarded, LWP gathers an event record for the next instruction to cause the event.
   - Similarly, if LWP needs to replay an instruction to gather a complete event record, the replay may abort instead of retiring. The event counter continues counting below zero and LWP gathers an event record for the next instruction to cause the event.
3. **Store—**When a complete event record is gathered, LWP stores it into the event ring buffer in the process' address space and advances the ring buffer head pointer.
   - LWP checks to see if the ring buffer is full, i.e., if advancing the ring buffer head pointer would make it equal to the tail pointer. If the buffer is full, LWP increments the 64-bit counter LWPCB.MissedEvents. If LWP is running in synchronized mode, it does not advance the head pointer. If LWP is running in continuous mode, it always advances the head pointer and LWPCB.MissedEvents counts the number of times that the buffer wrapped.
   - If more than one event record reaches the Store stage simultaneously, only one need be stored. Though LWP might store all such event records, it may delay storing some event records or it may discard the information and proceed to choose the next eligible instruction for the discarded event type(s). This behavior is implementation dependent.
   - The store need not complete synchronously with the instruction retiring. In other words, if LWP buffers the event record contents, the Store stage (and subsequent stages) may complete

some number of cycles after the tagged instruction retires. The data about the event and the instruction are precise, but the Report and Reset steps (below) may complete later.

4. **Report—**If LWP threshold interrupts are enabled and the space used in the event ring buffer exceeds a user-defined threshold, LWP initiates an interrupt. The OS can use this to signal the process to empty the ring buffer. Note that the interrupt may occur significantly later than the event that caused the threshold to be reached.

5. **Reset—**For each event that was stored, the counter is reset to its programmed interval. If requested by the application, LWP applies randomization to the low order bits of the interval. Counting for that event continues. Reset happens if the ring buffer head pointer was advanced or if the missed event counter was incremented. If the event counter went below -1, indicating that additional events occurred between the selected event and the time it was reported, that overrun value reduces the reset value so as to preserve the statistical distribution of events.

   For all events except the LWPVAL instruction, the hardware may impose a minimum on the reset value of an event counter. This prevents the system from spending too much time storing samples rather than making forward progress on the application. Any minimum imposed by the hardware can be detected by examining the EventInterval*n* fields in the LWPCB after enabling LWP.

An application should periodically remove event records from the ring buffer and advance the tail pointer. (If the application does not process the event records quickly enough or often enough, the LWP hardware will detect that the ring buffer is full and will miss events.) There are two ways to process the gathered events: interrupts or polling.

The application can wait until a threshold interrupt occurs to process the event records in the ring buffer. This requires OS or driver support. (As a consequence, interrupts can only be enabled if a kernel mode routine allows it; see “LWP_CFG—LWP Configuration MSR” on page 439) One usage model is to associate the LWP interrupt with a semaphore or mutex. When the interrupt occurs, the OS or driver signals the associated object. A thread waiting on the object wakes up and empties the ring buffer. Other models are possible, of course.

Alternatively, the application can have a thread that periodically polls the ring buffer. The polling thread need not be part of the process that is using LWP. It can be in a separate process that shares the memory containing the LWP control block and ring buffer.

Access to the ring buffer uses a lockless protocol between the LWP hardware and the application. The hardware owns the head pointer and the area in the ring buffer from the head pointer up to (but not including) the tail pointer. The application must not modify the head pointer nor rely on any data in the area of the ring buffer owned by the hardware. If the application has a stale value for the head pointer, it may miss an existing event record but it will never read invalid data. When the application is done emptying the ring buffer, it should refresh its copy of the head pointer to see if the LWP hardware has added any new event records.

Similarly, the application owns the tail pointer and the area in the ring buffer from the tail pointer up to (but not including) the head pointer. The hardware will never modify the tail pointer or overwrite the data in that region of the ring buffer. If the hardware has a stale value for the tail pointer, it may

consider that the ring buffer is full or at its threshold, but it will never overwrite valid data. Instead, it refreshes its copy of the tail pointer and rechecks to see if the full or threshold condition still applies.

When LWP is in continuous mode, this lockless protocol does not work, since the LWP hardware may overwrite the event records in the ring buffer when it advances the head pointer past the tail pointer. Because of this, the application must stop LWP before removing event records from the ring buffer. This prevents the hardware from wrapping through the ring buffer asynchronously from the application's attempt to remove data from it.

To use continuous mode properly, the application should set LWPCB.MissedEvents to 0 and set the head and tail pointers to the start of the ring buffer before starting LWP. To empty the ring buffer, the application should stop LWP. If LWPCB.MissedEvents is 0, the buffer did not wrap and there are event records starting at the tail pointer and continuing up to (but not including) the head pointer. If MissedEvents is not 0, the buffer wrapped and there are event records starting with the oldest one pointed to by the head pointer and continuing (possibly wrapping) all the way around to the newest one just before the head pointer.

13.4.2 Events and Event Records

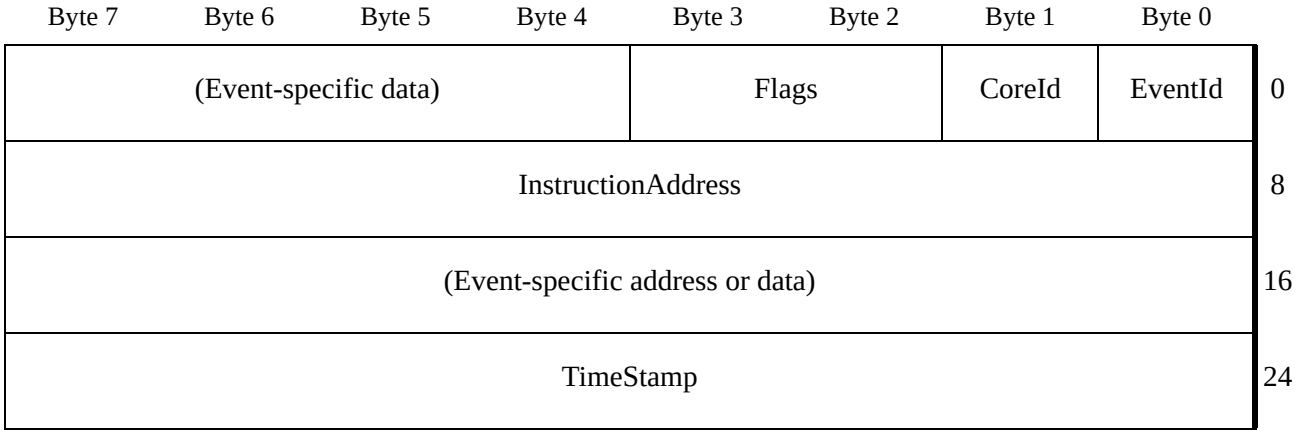
When a monitored event overflows its event counter, LWP puts an event record into the LWP event ring buffer. If event timestamping is supported and enabled, each event record will include a TimeStamp value. This value is a copy of the contents of Performance Timestamp Counter (PTSC) zero-extended if necessary to 64 bits.

The PTSC is a free-running counter that increments at a constant rate of 100MHz and is synchronized across all cores on a node to within +/-1. This counter starts when the processor is initialized and cannot be reset or modified. It is at least 40 bits wide. Privileged code can read the PTSC value via the RDMSR instruction. The size of the counter is indicated by the 2-bit field CPUID Fn8000_0008_ECX[PerfTscSize]. A value of 00b means that the PTSC is 40 bits wide; 01b means 48 bits, 10b means 56 bits, and 11b indicates a full 64 bits.

The PTSC can be correlated to the architectural TSC that runs at the P0 frequency. An application can read the TSC and PTSC, wait a 1000 clock periods or so, then read them again. The ratio of the differences is the scaling factor for the counters.

The event record size is fixed but may vary based on implementation. The event record size for a given processor is discovered by executing CPUID Fn8000_001C and extracting the value of the LwpEventSize field. (See "Detecting LWP Capabilities" on page 436). Current implementations fix the record size at 32 bytes and this size is used in the record format specifications below.

Reserved fields and fields that are not defined for a particular event are set to zero when LWP writes an event record.



Bytes	Field	Description
0	EventId	Event identifier specifying the event record type. Valid identifiers are 1 to 255. 0 is an invalid identifier.
1	CoreId	CPU core identifier value from COREID field of LWP_CFG (see “LWP_CFG—LWP Configuration MSR” on page 439). For multicore systems, this typically identifies the core on which LWP is running. This allows software to aggregate event records from multiple threads into a single data structure without losing CPU information. It also allows software to detect when a thread has migrated from one core to another.
3–2	Flags	Event-specific flags.
7–4		Event-specific data.
15–8	InstructionAddress	The Effective Address of the instruction that triggered this event record. This is the value before adding in the CS base address. If the base is non-zero, software must track it. (Modern operating systems use a CS base of zero, and CS is unused in long mode.)
23–16		Event-specific address or other data.
31–24	TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-21. Generic Event Record

Table 13-6 below lists the event identifiers for the events defined in version 1 of LWP. They are described in detail in the following sections.

Table 13-6. EventId Values

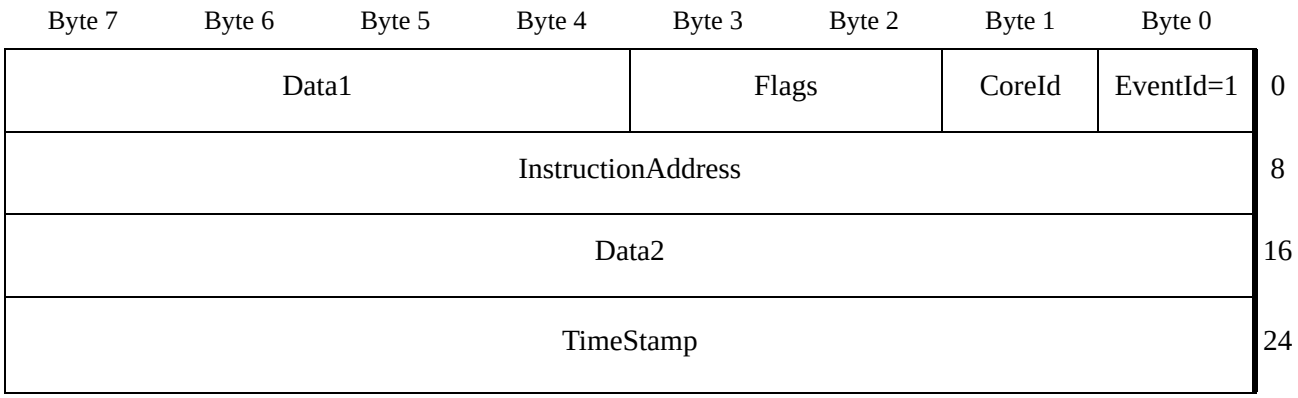
EventId	Description
0	Reserved – invalid event
1	Programmed value sample
2	Instructions retired
3	Branches retired
4	DCache misses

Table 13-6. EventId Values (continued)

EventId	Description
5	CPU clocks not halted
6	CPU reference clocks not halted
255	Programmed event

13.4.2.1 Programmed Value Sample

LWP decrements the event counter each time the program executes the LWPVAL instruction (see “LWPVAL—Insert Value Sample in LWP Ring Buffer” on page 443). When the counter becomes negative, it stores an event record with an EventId of 1. The data in the event record come from the operands to the instruction as detailed in the instruction description.



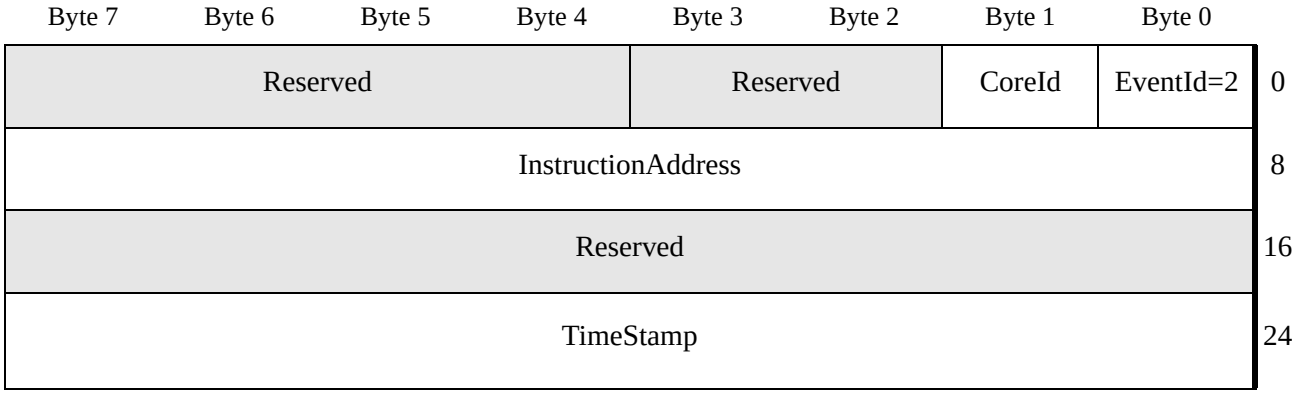
Bytes	Field	Description
0	EventId	Event identifier = 1
1	CoreId	CPU core identifier from LWP_CFG
3–2	Flags	Immediate value (bottom 16 bits)
7–4	Data1	Reg/mem value
15–8	InstructionAddress	Instruction address of LWPVAL instruction
23–16	Data2	Reg value (zero extended if running in legacy mode)
31–24	TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-22. Programmed Value Sample Event Record

13.4.2.2 Instructions Retired

LWP decrements the event counter each time an instruction retires. When the counter becomes negative, it stores a generic event record with an EventId of 2.

Instructions are counted if they execute entirely in user mode (CPL = 3). Instructions that change to CPL 3 from some other level are not counted, since the instruction address is not an address in user mode space. All user mode instructions are counted, including LWPVAL and LWPINS.



Bytes	Bits	Field	Description
0	7:0	EventId	Event identifier = 2
1	7:0	CoreId	CPU identifier from LWP_CFG
7–2			Reserved
15–8		InstructionAddress	Instruction address
23–16			Reserved
31–24		TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-23. Instructions Retired Event Record

13.4.2.3 Branches Retired

LWP decrements the event counter each time a transfer of control retires, regardless of whether or not it is taken. When the counter becomes negative, it stores an event record with an EventId of 3.

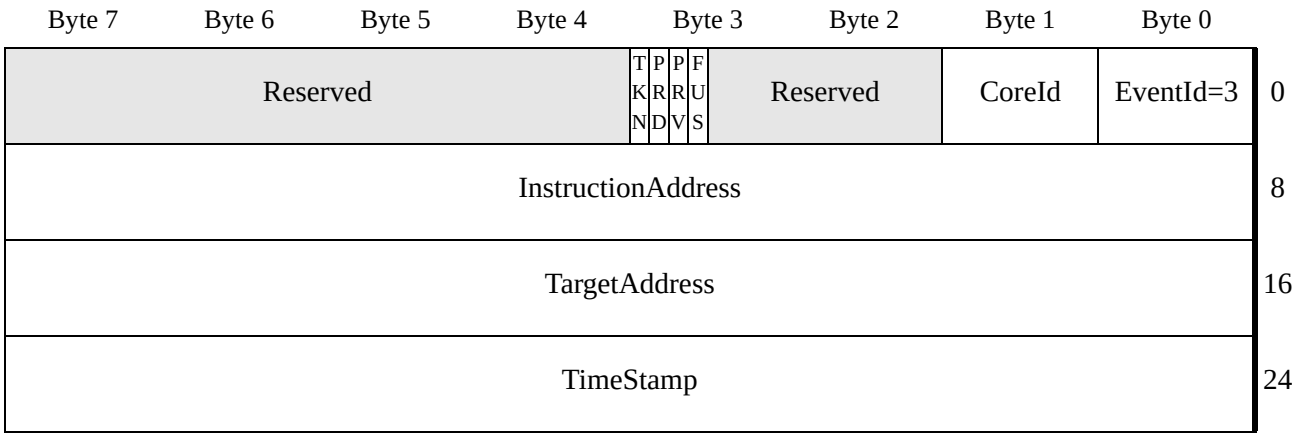
Control transfer instructions that are counted are:

- JMP (near), Jcc, JCXZ, JEXCZ, and JRCXZ
- LOOP, LOOPE, and LOOPNE
- CALL (near) and RET (near)

LWP does not count JMP (far), CALL (far), RET (far), traps, or interrupts (whether synchronous or asynchronous), nor does it count operations that switch to or from ring 3, SMM, or SVM, such as SYSCALL, SYSENTER, SYSEXIT, SYSRET, VMMCALL, INT, or INTO.

Some implementations of the AMD64 architecture perform an optimization called “fusing” when a compare operation (or other operation that sets the condition codes) is followed immediately by a conditional branch. The processor fuses these into a single operation internally before they are executed. While this is invisible to the programmer, the address of the actual branch is not available for LWP to report when the (fused) instruction retires. In this case, LWP sets the FUS bit in the event record and reports the address of the operation that set the condition codes. If FUS is set, software can find the address of the actual branch by decoding the instruction at the reported InstructionAddress and

adding its length to that address. (Note that fused instructions do count as 2 instructions for the Instructions Retired event, since there were 2 x86 instructions originally.)



Bytes	Bits	Field	Description
0	7:0	EventId	Event identifier = 3
1	7:0	CoreId	CPU core identifier from LWP_CFG
3–2	11:0		Reserved
3	4	FUS	1—Fused operation. InstructionAddress points to a compare operation (or other operation that sets the condition codes) immediately preceding the branch. 0—InstructionAddress points to the branch instruction.
3	5	PRV	1—PRD bit is valid 0—Prediction information is not available Some implementations of LWP may be unable to capture branch prediction information on some or all branches.
3	6	PRD	1—Branch was predicted correctly 0—Mispredicted If PRV = 0, the value of PRD is unpredictable and should be ignored. For unconditional branches, PRD=1 if PRV=1.
3	7	TKN	1—Branch was taken 0—Branch not taken Always 1 for unconditional branches.
7–4			Reserved
15–8		InstructionAddress	Instruction address
23–16		TargetAddress	Address of instruction executed after branch. This is the target if the branch was taken and the fall-through address if the branch was a not-taken conditional branch. TargetAddress is the Effective Address value before adding in the CS base address.
31–24		TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-24. Branch Retired Event Record

13.4.2.4 DCache Misses

LWP decrements the event counter each time a load from memory causes a DCache miss whose latency exceeds the LwpCacheLatency threshold and/or whose data come from a level of the cache or memory hierarchy that is selected for counting. When the counter becomes negative, LWP stores an event record with an EventId of 4.

A misaligned access that causes two misses on a single load decrements the event counter by 1 and, if it reports an event, the data are for the lowest address that missed. LWP only counts loads directly caused by the instruction. It does not count cache misses that are indirectly due to TLB walks, LDT or GDT references, TLB misses, etc. Cache misses caused by LWP itself accessing the LWPCB or the event ring buffer are not counted.

Measuring Latency

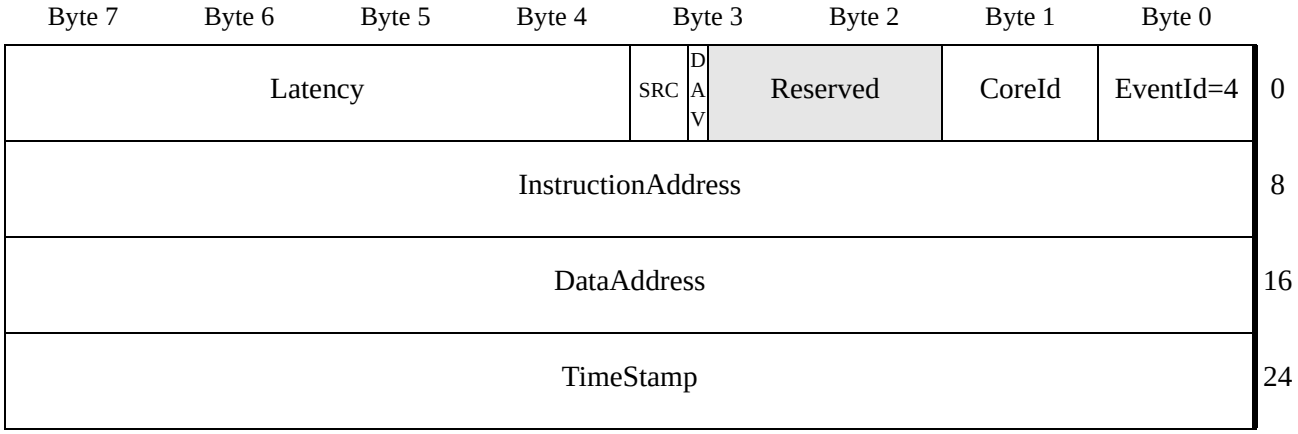
The x86 architecture allows multiple loads to be outstanding simultaneously. An implementation of LWP might not have a full latency counter for every load that is waiting for a cache miss to be resolved. Therefore, an implementation may apply any of the following simplifications. Software using LWP should be prepared for this.

- The implementation may round the latency to a multiple of 2^*j*. This is a small power of 2, and the value of *j* must be 1 to 4. For example, in the rest of this section, assume that *j* = 4, so 2^*j* = 16. The low 4 bits of latency reported in the event record will be 0. The actual latency counter is incremented by 16 every 16 cycles of waiting. The value of *j* is returned as LwpLatencyRnd (see “Detecting LWP Capabilities” on page 436).
- The implementation may do an approximation when starting to count latency. If counting is in increments of 16, the 16 cycles need not start when the load begins to wait. The implementation may bump the latency value from 0 to 16 any time during the first 16 cycles of waiting.
- The implementation may cap total latency to 2^*n*-16 (where *n* >= 10). The latency counter is thus a saturating counter that stops counting when it reaches its maximum value. For example, if *n* = 10, the latency value will count from 0 to 1008 in steps of 16 and then stop at 1008. (If *n* = 10, each counter is only 6 bits wide.) The value of *n* is returned as LwpLatencyMax (see “Detecting LWP Capabilities” on page 436).

Note that the latency threshold used to filter events is a multiple of 16. This value is used in the comparison that decides whether a cache miss event is eligible to be counted.

Reporting the DCache Miss Data Address

The event record for a DCache miss reports the linear address of the data (after adding in the segment base address, if any). The way an implementation records the linear address affects the exact event that is reported and the amount of time it takes to report a cache miss event. The implementation may report the event immediately, report the next eligible event once the counter becomes negative, or replay the instruction.

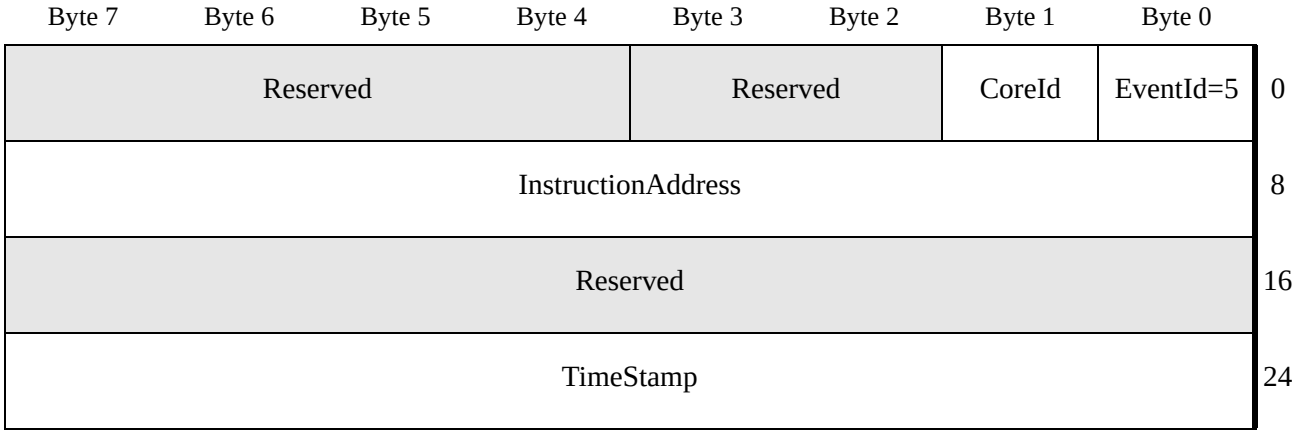


Bytes	Bits	Field	Description
0	7:0	EventId	Event identifier = 4
1	7:0	CoreId	CPU identifier from LWP_CFG
2–3	11:0		Reserved
3	4	DAV	1—DataAddress is valid 0—Address is unavailable
3	5:7	SRC	Data source for the requested data 0 No valid status 1 Local L3 cache 2 Remote CPU or L3 cache 3 DRAM 4 Reserved (for Remote cache) 5 Reserved 6 Reserved 7 Other (MMIO/Config/PCI/APIC)
7–4		Latency	Total latency of cache miss (in cycles)
15–8		InstructionAddress	Instruction address
23–16		DataAddress	Address of memory reference (if flag bit 28 = 1)
31–24		TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-25. DCache Miss Event Record

13.4.2.5 CPU Clocks not Halted

LWP decrements the event counter each clock cycle that the CPU is not in a halted state (due to STPCLK or a HLT instruction). When the counter becomes negative, it stores a generic event record with an EventId of 5. This counter varies in real-time frequency as the core clock frequency changes.



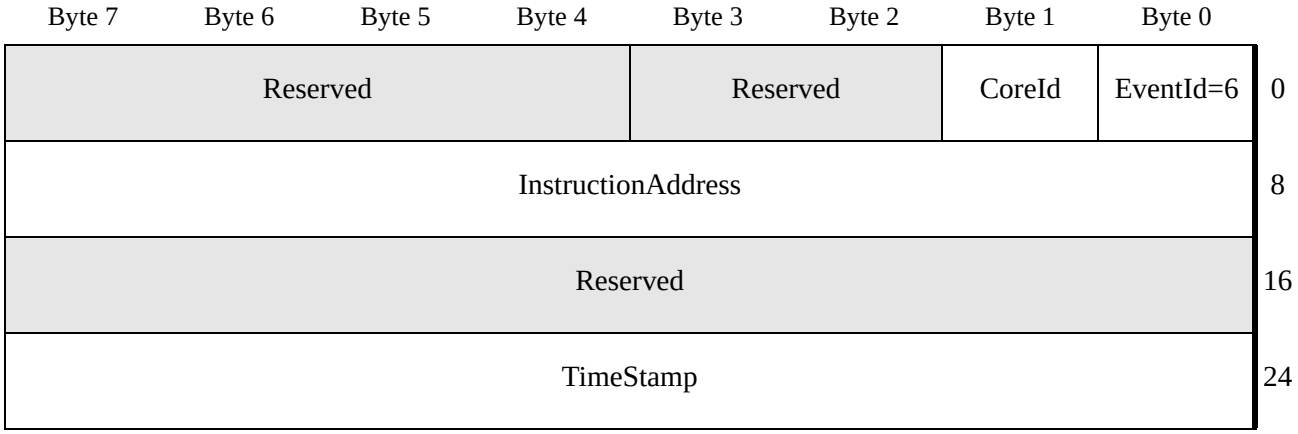
Bytes	Bits	Field	Description
0	7:0	EventId	Event identifier = 5
1	7:0	CoreId	CPU identifier from LWP_CFG
7–2			Reserved
15–8		InstructionAddress	Instruction address
23–16			Reserved
31–24		TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-26. CPU Clocks not Halted Event Record

13.4.2.6 CPU Reference Clocks not Halted

LWP decrements the event counter each reference clock cycle that the CPU is not in a halted state (due to STPCLK or a HLT instruction). When the counter becomes negative, it stores a generic event record with an EventId of 6.

The reference clock runs at a constant frequency that is independent of the core frequency and of the performance state. The reference clock frequency is processor dependent. The processor may implement this event by subtracting the ratio of (reference clock frequency / core clock frequency) each core clock cycle.

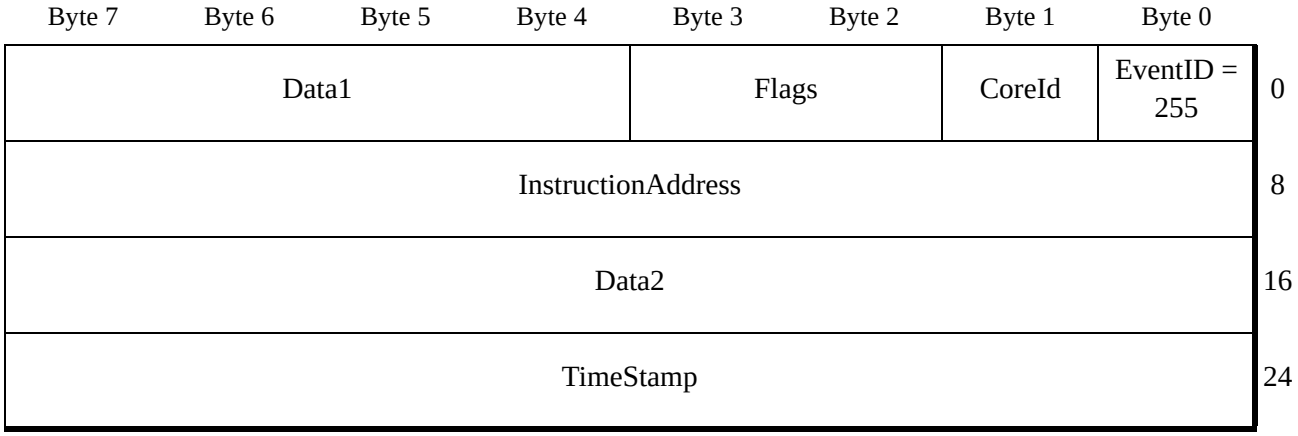


Bytes	Bits	Field	Description
0	7:0	EventId	Event identifier = 6
1	7:0	CoreId	CPU identifier from LWP_CFG
2–7			Reserved
15–8		InstructionAddress	Instruction address
23–16			Reserved
31–24		TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-27. CPU Reference Clocks not Halted Event Record

13.4.2.7 Programmed Event

When a program executes the LWPINS instruction (see “LWPINS—Insert User Event Record in LWP Ring Buffer” on page 444), the processor stores an event record with an event identifier of 255. The data in the event record come from the operands to the instruction as detailed in the instruction description.



Bytes	Field	Description
0	EventId	Event identifier = 255
1	CoreId	CPU identifier from LWP_CFG
3–2	Flags	Imm16 value
7–4	Data1	Reg/mem value
15–8	InstructionAddress	Instruction address of LWPINS instruction
23–16	Data2	Reg value (zero extended if running in legacy mode)
31–24	TimeStamp	Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.

Figure 13-28. Programmed Event Record

13.4.2.8 Other Events

The overall design of LWP allows easy extension to the list of events that it can monitor. The following are possibilities for events that may be added in future versions of LWP:

- DTLB misses
- FPU operations
- ICache misses
- ITLB misses

13.4.3 Detecting LWP

An application uses the CPUID instruction to identify whether Lightweight Profiling is present and which of its capabilities are available for use. An operating system uses CPUID to determine whether LWP is supported on the hardware and to determine which features of LWP are supported and can be made available to applications.

13.4.3.1 Detecting LWP Presence

LWP is supported on a processor if CPUID Fn8000_0001_ECX[LWP] (bit 15) is set. This bit is identical to the value of CPUID Fn0000_000D_EDX_x0[bit 30], which is bit 62 of the XFeatureSupportedMask and indicates XSAVE support for LWP. A system can check either of those bits to determine if LWP is supported. Since LWP requires XSAVE, software can assume that this bit being set implies that CPUID Fn0000_0001_ECX[XSAVE] (bit 26) is also set.

13.4.3.2 Detecting LWP XSAVE Area

The size of the LWP extended state save area used by XSAVE/XRSTOR is 128 bytes (080h). This value is returned by CPUID Fn0000_000D_ EAX_x3E (ECX=62).

The offset of the LWP save area from the beginning of the XSAVE/XRSTOR area is 832 bytes (340h). This value is returned by CPUID Fn0000_000D_ EBX_x3E (ECX=62).

The size of the LWP save area is included in the XFeatureSupportedSizeMax value returned by CPUID Fn0000_000D_ECX_x0 (ECX=0).

If LWP is enabled in the XFEATURE_ENABLED_MASK, the size of the LWP save area is included in the XFeatureEnabledSizeMax value returned by CPUID Fn0000_000D_EBX_x0 (ECX=0).

13.4.3.3 Detecting LWP Capabilities

The values returned by CPUID Fn8000_001C indicate the capabilities of LWP. See Table 13-7, “Lightweight Profiling CPUID Values” for a listing of the returned values.

Bit 0 of EAX is a copy of bit 62 from XFEATURE_ENABLED_MASK and indicates whether LWP is available for use by applications. If it is 1, the processor supports LWP and the operating system has enabled LWP for applications.

Bits 31:1 returned in EAX are taken from the LWP_CFG MSR and reflect the LWP features that are available for use. These are a subset of the bits returned in EDX, which reflect the full capabilities of LWP on current processor. The operating system can make a subset of LWP available if it cannot handle all supported features. For instance, if the OS cannot handle an LWP threshold interrupt, it can disable the feature. User-mode software must assume that the bits in EAX describe the features it can use. Operating systems should use the bits from EDX to determine the supported capabilities of LWP and make all or some of those features available.

Under SVM, if a VMM allows the migration of guests among processors that all support LWP, it must arrange for CPUID to report the logical AND of the supported feature bits over all processors in the migration pool. Other CPUID values must also be reported as the “least common denominator” among the processors.

Table 13-7. Lightweight Profiling CPUID Values

Reg	Bits	Field	Description
EAX	0	LwpAvail	1—LWP is available to application programs. The hardware and the operating system support LWP. 0—LWP is not available. This bit is a copy of bit 62 of the XFEATURE_ENABLED_MASK register (XCR0).
	1	LwpVAL	LWPVAL instruction (EventId = 1) is available.
	2	LwpIRE	Instructions retired event (EventId = 2) is available.
	3	LwpBRE	Branch retired event (EventId = 3) is available.
	4	LwpDME	DCache miss event (EventId = 4) is available.
	5	LwpCNH	CPU clocks not halted event (EventId = 5) is available.
	6	LwpRNH	CPU reference clocks not halted event (EventId = 6) is available.
	28:7		Reserved
	29	LwpCont	Sampling in continuous mode is available.
	30	LwpPTSC	Performance Time Stamp Counter in event records is available.
	31	LwpInt	Interrupt on threshold overflow is available.
EBX	7:0	LwpCbSize	Size in quadwords of the LWPCB. This value is at least (LwpEventOffset / 8) + LwpMaxEvents but an implementation may require a larger control block.
	15:8	LwpEventSize	Size in bytes of an event record in the LWP event ring buffer. (32 for LWP Version 1.)
	23:16	LwpMaxEvents	Maximum supported EventId value (not including EventId 255 used by the LWPINS instruction). Not all events between 1 and LwpMaxEvents are necessarily supported.
	31:24	LwpEventOffset	Offset from the start of the LWPCB to the EventInterval1 field. Software uses this value to locate the area of the LWPCB that describes events to be sampled. This permits expansion of the initial fixed region of the LWPCB. LwpEventOffset is always a multiple of 8.

Table 13-7. Lightweight Profiling CPUID Values

Reg	Bits	Field	Description
ECX	4:0	LwpLatencyMax	Number of bits in cache latency counters (10 to 31). 0 if DCache miss event is not supported (EDX[LwpDME] = 0).
	5	LwpDataAddress	1—Cache miss event records report the data address of the reference. 0—Data address is not reported. 0 if DCache miss event is not supported (EDX[LwpDME] = 0).
	8:6	LwpLatencyRnd	The amount by which cache latency is rounded. The bottom LwpLatencyRnd bits of latency information will be zero. The actual number of bits implemented for the counter is (LwpLatencyMax – LwpLatencyRnd). Must be 0 to 4. 0 if DCache miss event is not supported (EDX[LwpDME] = 0).
	15:9	LwpVersion	Version of LWP implementation. (1 for LWP Version 1.)
	23:16	LwpMinBufferSize	Minimum size of the LWP event ring buffer, in units of 32 event records. At least 32*LwpMinBufferSize records must be allocated for the LWP event ring buffer, and hence the size of the ring buffer must be at least 32 * LwpMinBufferSize * LwpEventSize bytes. If 0, there is no minimum.
	27:24		Reserved
	28	LwpBranchPrediction	1—Branches Retired events can be filtered based on whether the branch was predicted properly. The values of NMB and NPB in the LWPCB enable filtering based on prediction. 0—NMB and NPB fields of the LWPCB are ignored. 0 if Branches Retired event is not supported (EDX[LwpBRE] = 0).
	29	LwpIpFiltering	1—IP filtering is supported. 0—IP filtering is not supported. The IPI, IPF, BaseIP, and LimitIP fields of the LWPCB are ignored.
	30	LwpCacheLevels	1—Cache-related events can be filtered by the cache level that returned the data. The value of CLF in the LWPCB enables cache level filtering. 0—CLF is ignored. An implementation must support filtering either by latency or by cache level. It may support both. 0 if DCache miss event is not supported (EDX[LwpDME] = 0).
	31	LwpCacheLatency	1—Cache-related events can be filtered by latency. The value of MinLatency in the LWPCB controls filtering. 0—MinLatency is ignored. An implementation must support filtering either by latency or by cache level. It may support both. 0 if DCache miss event is not supported (EDX[LwpDME] = 0).
EDX	0	LwpAvail	LWP is supported. If 0, the remainder of the data returned by CPUID should be ignored. This bit is a copy of CPUID Fn8000_0001_ECX[LWP] (bit 15).
	1	LwpVAL	LWPVAL instruction (EventId = 1) is supported.

Table 13-7. Lightweight Profiling CPUID Values

Reg	Bits	Field	Description
	2	LwpIRE	Instructions retired event (EventId = 2) is supported.
	3	LwpBRE	Branch retired event (EventId = 3) is supported.
	4	LwpDME	DCache miss event (EventId = 4) is supported.
	5	LwpCNH	CPU clocks not halted event (EventId = 5) is supported.
	6	LwpRNH	CPU reference clocks not halted event (EventId = 6) is supported.
	28:7		Reserved
	29	LwpCont	Sampling in continuous mode is supported.
	30	LwpPTSC	Performance Time Stamp Counter in event records is supported.
	31	LwpInt	Interrupt on threshold overflow is supported.

For more information on using the CPUID instruction, refer to Section 3.3, “Processor Feature Identification,” on page 71.

13.4.4 LWP Registers

The XFEATURE_ENABLED_MASK register (extended control register XCR0) and the LWP model-specific registers describe and control the LWP hardware. The MSRs are available if CPUID Fn8000_0001_ECX[LWP] (bit 15) is set. LWP can only be used if the system has made support for LWP state management available in XFEATURE_ENABLED_MASK.

13.4.4.1 XFEATURE_ENABLED_MASK Support

LWP requires that the processor support the XSAVE/XRSTOR instructions to manage LWP state, along with the XSETBV/XGETBV instructions that manage the enabled state mask. An operating system uses XSETBV to set bit 62 of XFEATURE_ENABLED_MASK to indicate that it supports management of LWP state and allows applications to use LWP. When the system makes LWP available by setting bit 62 of XFEATURE_ENABLED_MASK, LWP is initially disabled (LWP_CBADDR is zero).

See “Guidelines for Operating Systems” on page 462 for details on how to implement LWP support in an operating system.

13.4.4.2 LWP_CFG—LWP Configuration MSR

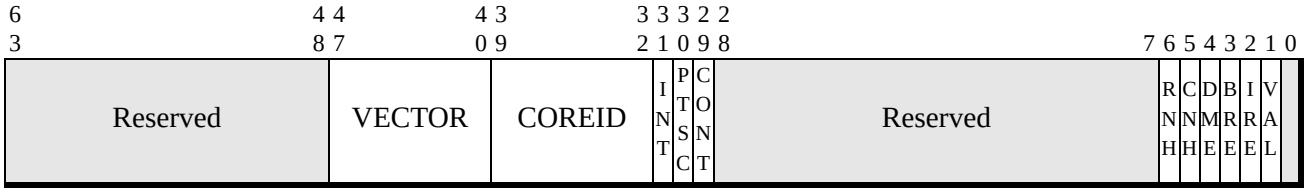
LWP_CFG (MSR C000_0105h) controls which features of LWP are available on the processor. The operating system loads LWP_CFG at start-up time (or at the time an LWP driver is loaded) to indicate its level of support for LWP. Only bits for supported features (those that are set in CPUID Fn8000_001C_EDX) can be turned on in LWP_CFG. Attempting to set other bits causes a #GP fault.

User code can examine LWP_CFG bits 31:1 by reading CPUID Fn8000_001C_EAX.

Bits 39:32 of LWP_CFG contains the COREID value that LWP will store into the CoreId field of every event record written by this core. The operating system should initialize this value to be the local APIC number, obtained from CPUID Fn0000_0001_EBX[LocalApicId] (bits 31:24). COREID is present so that when LWP is used in a virtualized environment, it has access to the core number without needing to enter the hypervisor. On systems that support x2APIC, local APIC numbers may be more than 8 bits wide. The operating system may then assign LWP COREID values that are small and identify the core within a cluster. If the system has more than 256 cores, there will be unavoidable duplication of COREID values.

Bits 47:40 of LWP_CFG specify the vector number that LWP will use when it signals a ring buffer threshold interrupt.

The reset value of LWP_CFG is 0.



Bits	Field	Description
0		Reserved
1	VAL	Allow the LWPVAL instruction.
2	IRE	Allow LWP to count instructions retired.
3	BRE	Allow LWP to count branches retired.
4	DME	Allow LWP to count DCache misses.
5	CNH	Allow LWP to count CPU clocks not halted.
6	RNH	Allow LWP to count CPU reference clocks not halted.
28:7		Reserved
29	CONT	Enable continuous mode. If 0, LWP will always use synchronized mode.
30	PTSC	Enable storing Performance Time Stamp Counter (PTSC) in the TimeStamp field of event records, if PTSC is available.
31	INT	Allow LWP to generate an interrupt when threshold is exceeded.
39:32	COREID	Value to store in CoreId field when writing an event record.
47:40	VECTOR	Interrupt vector number to use for LWP Threshold interrupts. Must be provided if INT=1.
63:48		Reserved

Figure 13-29. LWP_CFG—Lightweight Profiling Features MSR

13.4.4.3 LWP_CBADDR—LWPCB Address MSR

LWP_CBADDR (MSR C000_0106h) provides access to the internal copy of the LWPCB linear address.

RDMSR from this register returns the current LWPCB address without performing any of the operations described for the SLWPCB instruction.

WRMSR to this register with a non-zero value generates a #GP fault; use LLWPCB or XRSTOR to load an LWPCB address.

Writing a zero to LWP_CBADDR immediately disables LWP, discarding any internal state. For instance, an operating system can write a zero to stop LWP when it terminates a thread.

Note that LWP_CBADDR contains the linear address of the control block. All references to the LWPCB that are made by microcode during the normal operation of LWP ignore the DS segment register.

The reset value of LWP_CBADDR is 0. This means that when the system sets bit 62 of XFEATURE_ENABLED_MASK to make LWP available, it is initially disabled.

13.4.5 LWP Instructions

This section describes the instructions included in the AMD64 architecture to support LWP. These instructions raise #UD if LWP is not supported or if bit 62 of XFEATURE_ENABLED_MASK is 0 indicating that LWP is not available.

The LLWPCB instruction enables or disables Lightweight Profiling and controls the events being profiled. The SLWPCB instruction queries the current state of Lightweight Profiling.

LWP provides two instructions for inserting user data into the event ring buffer. The LWPINS instruction unconditionally stores an event record into the ring buffer, while the LWPVAL instruction uses an LWP event counter to sample program values at defined intervals.

The instructions LLWPCB, SLWPCB, LWPINS, and LWPVAL are also described in the chapter "General-Purpose Instruction Reference" of Volume 3. Refer to reference pages for the individual instruction for information on instruction encoding, flags affected, and exception behavior.

13.4.5.1 LLWPCB—Load LWPCB Address

Parses the Lightweight Profiling Control Block at the address contained in the specified register. If the LWPCB is valid, writes the address into the LWP_CBADDR MSR and enables Lightweight Profiling.

The LWPCB must be in memory that is readable and writable in user mode. For better performance, it should be aligned on a 64-byte boundary in memory and placed so that it does not cross a page boundary, though neither of these suggestions is required.

Action

1. If LWP is not available or if the machine is not in protected mode, LLWPCB immediately causes a #UD exception.
2. If LWP is already enabled, the processor flushes the LWP state to memory in the old LWPCB. See “SLWPCB—Store LWPCB Address” on page 443 for details on saving the active LWP state.

If the flush causes a #PF exception, LWP remains enabled with the old LWPCB still active. Note that the flush is done before LWP attempts to access the new LWPCB.

3. If the specified LWPCB address is 0, LWP is disabled and the execution of LLWPCB is complete.
4. The LWPCB address is non-zero. LLWPCB validates it as follows:
   - If any part of the LWPCB or the ring buffer is beyond the data segment limit, LLWPCB causes a #GP exception.
   - If the ring buffer size is below the implementation's minimum ring buffer size, LLWPCB causes a #GP exception.
   - While doing these checks, LWP reads and writes the LWPCB, which may cause a #PF exception.

   If any of these exceptions occurs, LLWPCB aborts and LWP is left disabled. Usually, the operating system will handle a #PF exception by making the memory available and returning to retry the LLWPCB instruction. The #GP exceptions indicate application programming errors.
5. LWP converts the LWPCB address and the ring buffer address to linear address form by adding the DS base address and stores the addresses internally.
6. LWP examines the LWPCB.Flags field to determine which events should be enabled and whether threshold interrupts should be taken. It clears the bits for any features that are not available and stores the result back to LWPCB.Flags to inform the application of the actual LWP state.
7. For each event being enabled, LWP examines the EventInterval*n* value and, if necessary, sets it to an implementation-defined minimum. (The minimum event interval for LWPVAL is zero.) It loads its internal counter for the event from the value in EventCounter*n*. A zero or negative value in EventCounter*n* means that the next event of that type will cause an event record to be stored. To count every j^{th} event, a program should set EventInterval*n* to *j-1* and EventCounter*n* to some starting value (where *j-1* is a good initial count). If the counter value is larger than the interval, the first event record will be stored after a larger number of events than subsequent records.
8. LWP is started. The execution of LLWPCB is complete.

Notes

If none of the bits in the LWPCB.Flags specifies an available event, LLWPCB still enables LWP to allow the use of the LWPINS instruction. However, no other event records will be stored.

A program can temporarily disable LWP by executing SLWPCB to obtain the current LWPCB address, saving that value, and then executing LLWPCB with a register containing 0. It can later re-enable LWP by executing LLWPCB with a register containing the saved address.

When LWP is enabled, it is typically an error to execute LLWPCB with the address of the active LWPCB. When the hardware flushes the existing LWP state into the LWPCB, it may overwrite fields that the application may have set to new LWP parameter values. The flushed values will then be loaded as LWP is restarted. To reuse an LWPCB, an application should stop LWP by passing a zero to LLWPCB, then prepare the LWPCB with new parameters and execute LLWPCB again to restart LWP.

Internally, LWP keeps the linear address of the LWPCB and the ring buffer. If the application changes the value of DS, LWP will continue to collect samples even if the new DS value would no longer allows it to access the LWPCB or the ring buffer. However, a #GP fault will occur if the application uses XRSTOR to restore LWP state saved by XSAVE. Programs should avoid using XSAVE/XRSTOR on LWP state if DS has changed. This only applies when the CPL ≠ 0; kernel mode operation of XRSTOR is unaffected by changes to DS. See “XSAVE/XRSTOR” on page 455 for details.

Operating system and hypervisor code that runs when the CPL ≠ 3 should use XSAVE and XRSTOR to control LWP rather than using LLWPCB (see below). Use WRMSR to write 0 to LWP_CBADDR to immediately stop LWP without saving its current state (see “LWP_CBADDR—LWPCB Address MSR” on page 440).

It is possible to execute LLWPCB when the CPL ≠ 3 or when SMM is active, but the system software must ensure that the LWPCB and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Furthermore, if LWP is enabled when a kernel executes LLWPCB, both the old and new control blocks and ring buffers must be accessible. Using LLWPCB in these situations is not recommended.

13.4.5.2 SLWPCB—Store LWPCB Address

Flushes LWP state to memory and returns the current effective address of the LWPCB in the specified register.

If LWP is not currently enabled, SLWPCB sets the specified register to zero.

The flush operation stores the internal event counters for active events and the current ring buffer head pointer into the LWPCB. If there is an unwritten event record pending, it is written to the event ring buffer.

If LWP_CBADDR is not zero, the value returned is an effective address that is calculated by subtracting the current DS.Base address from the linear address kept in LWP_CBADDR. Note that if DS has changed between the time LLWPCB was executed and the time SLWPCB is executed, this might result in an address that is not currently accessible by the application.

SLWPCB generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

It is possible to execute SLWPCB when the CPL ≠ 3 or when SMM is active, but if the LWPCB pointer is not zero, the system software must ensure that the LWPCB and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF fault. Using SLWPCB in these situations is not recommended.

13.4.5.3 LWPVAL—Insert Value Sample in LWP Ring Buffer

Decrements the event counter associated with the Programmed Value Sample event (see “Programmed Value Sample” on page 428). If the resulting counter value is negative, inserts an event record into the

LWP event ring buffer in memory and advances the ring buffer pointer. If the counter is not negative and the ModRM operand specifies a memory location, that location is not accessed.

The event record has an EventId of 1. The value in the register specified by vvvv (first operand) is stored in the Data2 field at bytes 23–16 (zero extended if the operand size is 32). The value in a register or memory location (second operand) is stored in the Data1 field at bytes 7–4. The immediate value (third operand) is truncated to 16 bits and stored in the Flags field at bytes 3–2. See Figure 13-22 on page 428.

If the ring buffer is not full or if LWP is running in continuous mode, the head pointer is advanced and the event counter is reset to the interval for the event (subject to randomization). If the ring buffer threshold is exceeded and threshold interrupts are enabled, an interrupt is signaled. If LWP is in continuous mode and the new head pointer equals the tail pointer, the MissedEvents counter is incremented to indicate that the buffer wrapped.

If the ring buffer is full and LWP is running in synchronized mode, the event record overwrites the last record in the buffer, the MissedEvents counter in the LWPCB is incremented, and the head pointer is not advanced.

LWPVAL generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

LWPVAL does nothing if LWP is not enabled or if the Programmed Value Sample event is not enabled in LWPCB.Flags. This allows LWPVAL instructions to be harmlessly ignored if profiling is turned off.

It is possible to execute LWPVAL when the CPL ≠ 3 or when SMM is active, but the system software must ensure that the memory operand (if present), the LWPCB, and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Using LWPVAL in these situations is not recommended.

LWPVAL can be used by a program to perform value profiling. This is the technique of sampling the value of some program variable at a predetermined frequency. For example, a managed runtime might use LWPVAL to sample the value of the divisor for a frequently executed divide instruction in order to determine whether to generate specialized code for a common division. It might sample the target location of an indirect branch or call to see if one destination is more frequent than others. Since LWPVAL does not modify any registers or condition codes, it can be inserted harmlessly between any instructions.

Note

When LWPVAL completes (whether or not it stored an event record in the event ring buffer), it counts as an instruction retired. If the Instructions Retired event is active, this might cause that counter to become negative and immediately store an event record. If LWPVAL also stored an event record, the buffer will contain two records with the same instruction address (but different EventId values).

13.4.5.4 LWPINS—Insert User Event Record in LWP Ring Buffer

Inserts a record into the LWP event ring buffer in memory and advances the ring buffer pointer.

The record has an EventId of 255. The value in the register specified by vvvv (first operand) is stored in the Data2 field at bytes 23–16 (zero extended if the operand size is 32). The value in a register or memory location (second operand) is stored in the Data1 field at bytes 7–4. The immediate value (third operand) is truncated to 16 bits and stored in the Flags field at bytes 3–2. See Figure 13-28 on page 435.

If the ring buffer is not full or if LWP is running in continuous mode, the head pointer is advanced and the CF flag is cleared. If the ring buffer threshold is exceeded and threshold interrupts are enabled, an interrupt is signaled. If LWP is in continuous mode and the new head pointer equals the tail pointer, the MissedEvents counter is incremented to indicate that the buffer wrapped.

If the ring buffer is full and LWP is running in synchronized mode, the event record overwrites the last record in the buffer, the MissedEvents counter in the LWPCB is incremented, the head pointer is not advanced, and the CF flag is set.

LWPINS generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

LWPINS simply clears CF if LWP is not enabled. This allows LWPINS instructions to be harmlessly ignored if profiling is turned off.

It is possible to execute LWPINS when the CPL $\neq 3$ or when SMM is active, but the system software must ensure that the memory operand (if present), the LWPCB, and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Using LWPINS in these situations is not recommended.

LWPINS can be used by a program to mark significant events in the ring buffer as they occur. For instance, a program might capture information on changes in the process' address space such as library loads and unloads, or changes in the execution environment such as a change in the state of a user-mode thread of control.

Note that when the LWPINS instruction finishes writing a event record in the event ring buffer, it counts as an instruction retired. If the Instructions Retired event is active, this might cause that counter to become negative and immediately store another event record with the same instruction address (but different EventId values).

13.4.6 LWP Control Block

An application uses the LWP Control Block (LWPCB) to specify the details of Lightweight Profiling operation. It is an interactive region of memory in which some fields are controlled and modified by the LWP hardware and others are controlled and modified by the software that processes the LWP event records.

Most of the fields in the LWPCB are constant for the duration of a LWP session (the time between enabling LWP and disabling it). This means that they are loaded into the LWP hardware when it is enabled, and may be periodically reloaded from the same location as needed. The contents of the

constant fields must not be changed during a LWP run or results will be unpredictable. Changing the LWPCB memory to read-only or unmapped will cause an exception the next time LWP attempts to access it. To change values in the LWPCB, disable LWP, change the LWPCB (or create a new one), and re-enable LWP.

A few fields are modified by the LWP hardware to communicate progress to the software that is emptying the event ring buffer. Software may read them but should never modify them during an LWP session. Other fields are for software to modify to indicate that progress has been made in emptying the ring buffer. Software writes these fields and the LWP hardware reads them as needed.

For efficiency, some of the LWPCB fields may be shadowed internally in the LWP hardware unit when profiling is enabled. LWP refreshes these fields from (or flushes them to) memory as needed to allow software to make progress. For more information, refer to “LWPCB Access” on page 461.

The BufferTailOffset field is at offset 64 in the LWPCB in order to place it in a separate cache line on most implementations, assuming that the LWPCB itself is aligned properly. This allows the software thread that is emptying the ring buffer to retain write ownership of that cache line without colliding with the changes made by LWP when writing BufferHeadOffset. In addition, most implementations will use a value of 128 as the offset to the EventInterval1 field, since that places the event information in a separate cache line.

All fields in the LWPCB (as shown in Figure 13-30) that are marked as “Reserved” (or “Rsvd”) should be zero.

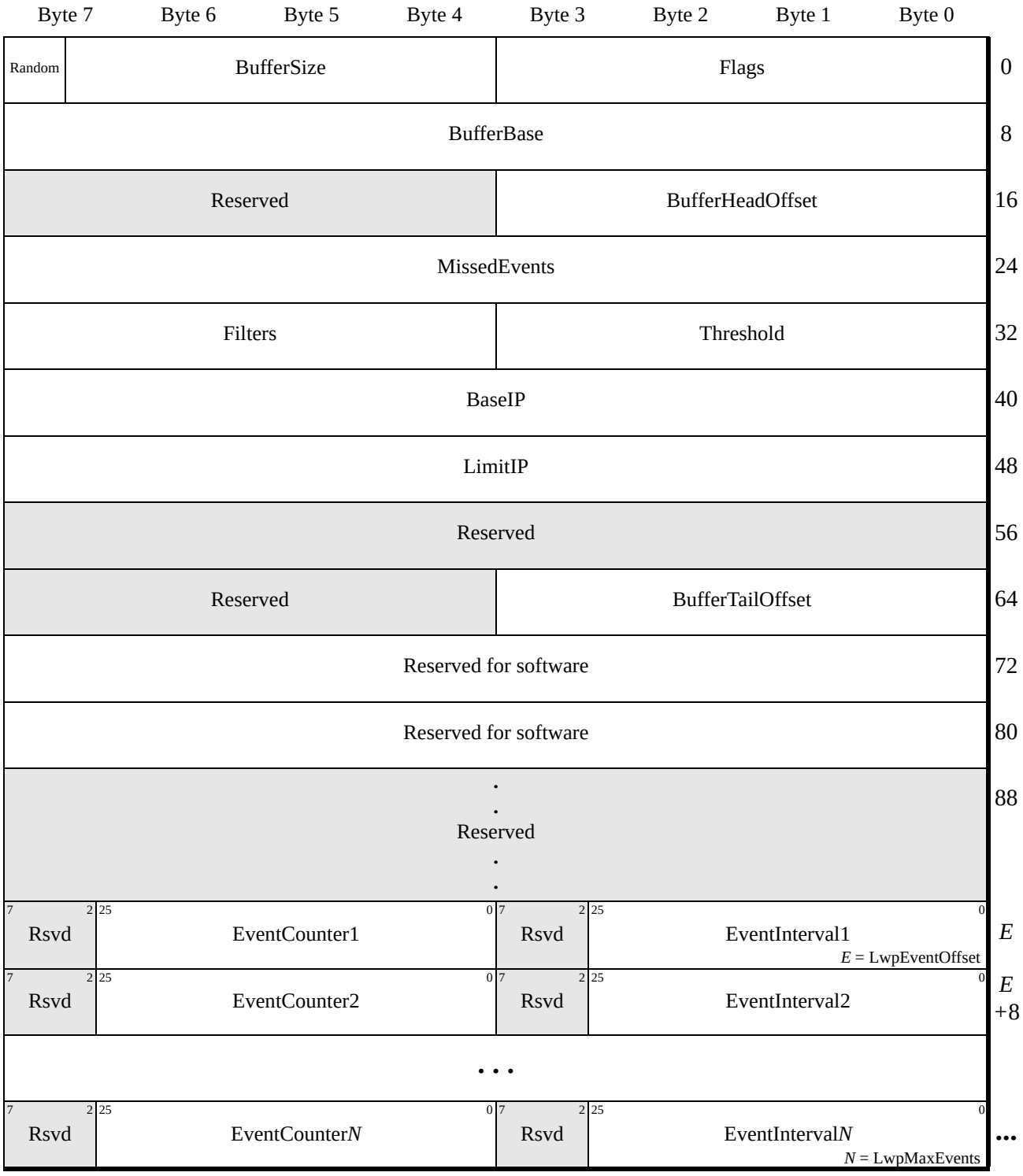


Figure 13-30. LWPCB—Lightweight Profiling Control Block

The R/W column in Table 13-8 below indicates how a field is used while LWP is enabled:

- LWP—hardware modifies the field; software may read it, but must not change it
- Init—hardware reads and modifies the field while executing LLWPCB; the field must then remain unchanged as long as the LWPCB is in use
- SW—software may modify the field; hardware may read it, but does not change it
- No—field must remain unchanged as long as the LWPCB is in use

Table 13-8. LWPCB—Lightweight Profiling Control Block Fields

Bytes	Bits	Field	Description	R/W
3–0		Flags	Flags indicating which events should be or are being counted (see Figure 13-31, “LWPCB Flags”) and whether threshold interrupts should be enabled. Before executing LLWPCB, the application sets Flags to a bit mask of the events (and interrupt) that should be enabled. LLWPCB does a logical “and” of this mask with the available feature bits in LWP_CFG and rewrites Flags with the mask of features actually enabled.	Init
7–4	27:0	BufferSize	Total size of the event ring buffer (in bytes). Must be a multiple of the event record size LwpEventSize (the value used internally will be rounded down if not). BufferSize must be at least (32 * LwpMinBufferSize * LwpEventSize).	No
7	7:4	Random	Number of bits of randomness to use in counters. Each time a counter is loaded from an interval to start counting down to the next event to record, the bottom Random bits are set to a random value. This avoids fixed patterns in events.	No
15–8		BufferBase	The Effective Address of the event ring buffer. Should be aligned on a 64-byte boundary for reasonable performance. Software is encouraged to align the ring buffer to a page boundary for best performance. If the default address size is less than 64 bits, the upper bits of BufferBase must be zero. LLWPCB converts BufferBase to a linear address and stores it internally. LWPCB.BufferBase is not modified.	No
19–16		BufferHeadOffset	Unsigned offset from BufferBase specifying where the LWP hardware will store the next event record. When BufferHeadOffset == BufferTailOffset, the ring buffer is empty. BufferHeadOffset must always be less than BufferSize; LWP will use a value of 0 if BufferHeadOffset is too large. Also, it must always be a multiple of LwpEventSize; LWP will round it down if not.	LWP
23–20			Reserved	
31–24		MissedEvents	The 64-bit count of the number of events that were missed. A missed event occurs when LWP stores an event record, attempts to advance BufferHeadOffset, and discovers that it would be equal to BufferTailOffset. In this case, LWP leaves BufferHeadOffset unchanged and instead increments the MissedEvents counter. Thus, when the ring buffer is full, the last event record is overwritten.	LWP

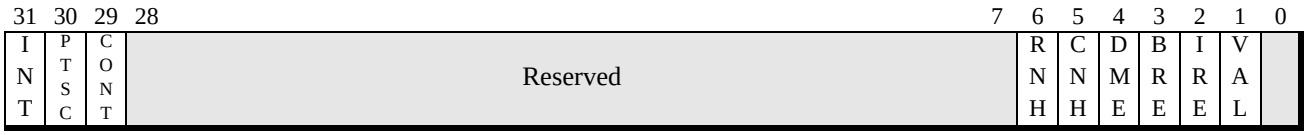
Table 13-8. LWPCB—Lightweight Profiling Control Block Fields (continued)

Bytes	Bits	Field	Description	R/W
35–32		Threshold	Threshold for signaling an interrupt to indicate that the ring buffer is filling up. If threshold interrupts are enabled in Flags, then when LWP advances BufferHeadOffset, it computes the space used as ((BufferHeadOffset – BufferTailOffset) % BufferSize). If the space used equals or exceeds Threshold, LWP causes an interrupt. If Threshold is greater than BufferSize, no interrupt will ever be taken. If Threshold is zero, an interrupt will be taken every time an event record is stored in the ring buffer. Threshold is an unsigned integer multiple of LwpEventSize (the value used internally will be rounded down if not). Ignored if threshold interrupts are not available in LWP_CFG or if they are not enabled in Flags	No
39–36		Filters	Filters to qualify which events are eligible to be counted. This field includes bits to filter branch events by type and prediction status, and bits and values to filter cache events by type and latency. See Figure 13-32, “LWPCB Filters” for details.	
47–40		BaseIP	Low limit of the IP filtering range. An instruction must start at a location greater than or equal to BaseIP to be in range. Ignored if IPF is zero or if the CPUID LwpIpFiltering bit is 0 to indicate that IP filtering is not supported.	No
55–48		LimitIP	High limit of the IP filtering range. An instruction must start at a location less than or equal to LimitIP to be in range. Ignored if IPF is zero or if the CPUID LwpIpFiltering bit is 0 to indicate that IP filtering is not supported.	No
63–56			Reserved	
67–64		BufferTailOffset	Unsigned offset from BufferBase to the oldest event record in the ring buffer. BufferTailOffset is maintained by software and must always be less than BufferSize and a multiple of LwpEventSize. If software stores a value of BufferTailOffset into the LWPCB that violates these rules, the LWP hardware might not detect ring buffer overflow or threshold conditions properly.	SW
71–68			Reserved	
72–87			Reserved for software use. These bytes are never read or written by the LWP hardware	SW
(*E*-1) – 88			Reserved area between the fixed portion of the LWPCB and the event specifiers. Should be zero. The EventInterval1 field is at offset *E* = LwpEventOffset.	

Table 13-8. LWPCB—Lightweight Profiling Control Block Fields (continued)

Bytes	Bits	Field	Description	R/W
(*E*+3)–*E*	25:0	EventInterval1	Reset value for counting events of type EventId = 1 (Programmed Value Sample). A value of *n* specifies that after *n+1* (modified by Random) LWPVAL instructions, LWP will store an event record in the ring buffer. EventInterval1 is a signed value. If it is negative, LLWPCB will use zero and will store zero into EventInterval1 in the LWPCB. The Programmed Value Sample event is the only one which allows an interval to be below the implementation minimum interval value.	Init
E+3	7:2		Reserved	
(*E*+7)–(*E*+4)	25:0	EventCounter1	Starting (LLWPCB) or current (SLWPCB) value of counter. This is a signed number. LLWPCB treats a negative value as zero.	LWP
E+7	7:2		Reserved	
(*E*+11)–(*E+8*)	25:0	EventInterval2	Reset value for counting events of type EventId = 2 (Instructions Retired). A value of *n* specifies that after *n+1* (modified by Random) instructions are retired, LWP will store an event record in the ring buffer. EventInterval2 is a signed value. If it is negative or is below the implementation minimum, LLWPCB will use the minimum and will store that value into EventInterval2 in the LWPCB.	Init
E+11	7:2		Reserved	
(*E*+15)–(*E*+12)	57:32	EventCounter2	Starting (LLWPCB) or current (SLWPCB) value of counter. This is a signed number. LLWPCB treats a negative value as zero.	LWP
E+15	7:2		Reserved	
		Event3…	Repeat event configuration similar to EventInterval2 and EventCounter2 for EventId values from 3 to LwpMaxEvents.	

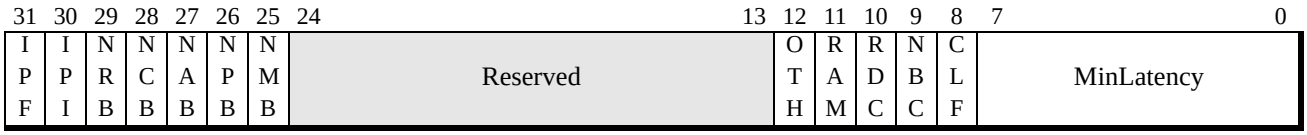
The LLWPCB instruction reads the Flags word from the LWPCB to determine which events to profile and whether threshold interrupts should be enabled. LLWPCB writes the Flags word after turning off bits corresponding to features which are not currently available.



Bit	Field	Input to LLWPCB	Value after LLWPCB
0		Reserved	
1	VAL	Enable LWPVAL instruction	LWPVAL instruction enabled
2	IRE	Enable Instructions Retired event	Instructions Retired event enabled
3	BRE	Enable Branches Retired event	Branches Retired event enabled
4	DME	Enable DCache miss event	DCache Miss event enabled
5	CNH	Enable CPU clocks not halted event	CPU Clocks Not Halted event enabled
6	RNH	Enable CPU reference clocks not halted event	CPU Reference Clocks Not Halted event enabled
28:7		Reserved	
29	CONT	1—Use continuous mode. If the ring buffer overflows, LWP continues to store events and advance BufferHead. Software must stop LWP in order to empty the ring buffer. 0—Use synchronized mode.	LWP operates in continuous mode if input bit is set and continuous mode is available. Otherwise, LWP operates in synchronous mode.
30	PTSC	1—Store the Performance Time Stamp Counter (PTSC) in the TimeStamp field of each event record, if PTSC is available. 0—Store 0 in the TimeStamp field.	Performance Time Stamp Counter value will be stored if input bit is set and PTSC feature is available. Otherwise 0 is stored.
31	INT	Enable threshold interrupts.	Threshold interrupts are enabled.

Figure 13-31. LWPCB Flags

Event counting can be filtered by a number of conditions which are specified in the Filters word of the LLWPCB. The IP filtering applies to all events. Cache level filtering applies to all events that interact with the caches. Branch filtering applies to the Branches Retired event.



Bits	Field	Description
7:0	MinLatency	Minimum latency for a cache-related event
8	CLF	Cache level filtering
9	NBC	Northbridge cache events
10	RDC	Remote data cache events
11	RAM	DRAM cache events
12	OTH	Other cache events
24:13		Reserved
25	NMB	No mispredicted branches
26	NPB	No predicted branches
27	NAB	No absolute branches
28	NCB	No conditional branches
29	NRB	No unconditional relative branches
30	IPI	IP filtering invert
31	IPF	IP filtering

Figure 13-32. LWPCB Filters

The following table provides detailed descriptions of the fields in the Filters word.

Table 13-9. LWPCB Filters Fields

Bits	Field	Description
7:0	MinLatency	Minimum latency for a cache-related event to be eligible for LWP counting. Applies to all cache-related events being monitored. MinLatency is multiplied by 16 to get the actual latency in cycles, providing less resolution but a larger range for filtering. An implementation may have a maximum for the latency value. If MinLatency*16 exceeds this maximum value, the maximum is used instead. A value of 0 disables filtering by latency. Ignored if no cache latency event is enabled or if the CPUID LwpCacheLatency bit is 0 to indicate that the implementation does not filter by latency (use the CLF bits to get a similar effect). At least one of these mechanisms is supported if any cache miss events are supported.
8	CLF	Cache level filtering. 1—Enables filtering cache-related events by the cache level or memory level that returned the data. It enables the next 4 bits. Cache-related events are only eligible for counting if the bit describing the memory level is on. 0—Disables cache level filtering. The next 4 bits are ignored, and any cache or memory level is eligible. Ignored if no cache latency event is enabled or if the CPUID LwpCacheLevels bit is 0 to indicate that the implementation does not filter by cache level (use the MinLatency field to get a similar effect). At least one of these mechanisms is supported if any cache miss events are supported.
9	NBC	Northbridge cache events. 1—Count cache-related events that are satisfied from data held in a cache that resides on the northbridge. 0—Ignore northbridge cache events Ignored if CLF is 0.
10	RDC	Remote data cache events. 1—Count cache-related events that are satisfied from data held in a remote data cache. 0—Ignore remote cache events. Ignored if CLF is 0.
11	RAM	DRAM cache events. 1—Count cache-related events that are satisfied from DRAM. 0—Ignore DRAM cache events. Ignored if CLF is 0.
12	OTH	Other cache events. 1—Count cache-related events that are satisfied from other sources, such as MMIO, Config space, PCI space, or APIC. 0—Ignore such cache events Ignored if CLF is 0.
24:13		Reserved

Table 13-9. LWPCB Filters Fields (continued)

Bits	Field	Description
25	NMB	No mispredicted branches. 1—Mispredicted branches will not be counted. 0—Mispredicted branches will be counted if not suppressed by other filter conditions. Caution: If NMB and NPB are both set, no branches will be counted. Ignored if the Branches Retired event is not enabled or if the CPUID LwpBranchPrediction bit is 0 to indicate that the implementation does not filter by prediction.
26	NPB	No predicted branches. 1—Correctly predicted branches will not be counted. Note that since direct branches are always predicted correctly, this is a superset of the NDB filter. 0—Correctly predicted branches will be counted if not suppressed by other filter conditions. Caution: If NMB and NPB are both set, no branches will be counted. Ignored if the Branches Retired event is not enabled or if the CPUID LwpBranchPrediction bit is 0 to indicate that the implementation does not filter by prediction.
27	NAB	No absolute branches. 1—Absolute branches will not be counted. This only applies to jumps through a register or memory (JMP opcode FF /4) and calls through a register or memory (CALL opcode FF /2). Relative branches (both conditional and unconditional) are counted normally if not disabled via the NRB or NCB bits. 0—Absolute branches will be counted if not suppressed by other filter conditions. Caution: If NRB, NCB, and NAB are all set, no branches will be counted. Ignored if the Branches Retired event is not enabled.
28	NCB	No conditional branches. 1—Conditional branches will not be counted. This only applies to conditional jumps (Jcc) and loops (LOOPcc). Unconditional relative branches, indirect jumps through a register or memory, and returns are counted normally if not disabled via the NRB or NAB bits. 0—Conditional branches will be counted if not suppressed by other filter conditions. Caution: If NRB, NCB, and NAB are all set, no branches will be counted. Ignored if the Branches Retired event is not enabled.

Table 13-9. LWPCB Filters Fields (continued)

Bits	Field	Description
29	NRB	No unconditional relative branches. 1—Unconditional relative branches will not be counted. This applies to unconditional jumps (JMP), calls (CALL), and returns (RET). Conditional branches and indirect jumps or calls through a register or memory are counted normally if not disabled via the NCB or NAB bits. 0—Direct branches will be counted if not suppressed by other filter conditions. Caution: If NRB, NCB, and NAB are all set, no branches will be counted. Ignored if the Branches Retired event is not enabled.
30	IPI	IP filtering invert. 1—IP filtering inverted. Only instructions outside the range from BaseIP to LimitIP are eligible for LWP counting. 0—IP filtering normal. Only instructions inside the range from BaseIP to LimitIP are eligible for LWP counting. Ignored if IPF is zero or if the CPUID LwpIpFiltering bit is 0 to indicate that IP filtering is not supported.
31	IPF	IP filtering. 1—IP filtering enabled. The values of the BaseIP and LimitIP fields specify a range of instruction addresses that are eligible for LWP event counting and reporting. The range is inclusive if IPI is 0 and exclusive if IPI is 1. 0—IP filtering disabled; instructions at every address are eligible for LWP counting. Ignored if the CPUID LwpIpFiltering bit is 0 to indicate that IP filtering is not supported.

13.4.7 XSAVE/XRSTOR

LWP requires that the processor support the XSAVE/XRSTOR instructions for managing extended processor state components.

13.4.7.1 Configuration

The processor uses bit 62 of XFEATURE_ENABLED_MASK (register XCR0) to indicate whether LWP state can be saved and restored, and thus whether LWP is available to applications. The LWP XSAVE area length and offset from the beginning of the XSAVE area are available from the CPUID instruction (see “Detecting LWP XSAVE Area” on page 436). In Version 1 of LWP, the LWP XSAVE area is 128 (080h) bytes long and the offset is 832 (340h) bytes.

13.4.7.2 XSAVE Area

Figure 13-33 below shows the layout of the XSAVE area for LWP. It is large enough to allow for future expansion of the number of event counters. Details of the fields are in Table 13-10.

All fields in the XSAVE area that are marked as “Reserved” (or “Rsvd”) must be zero.

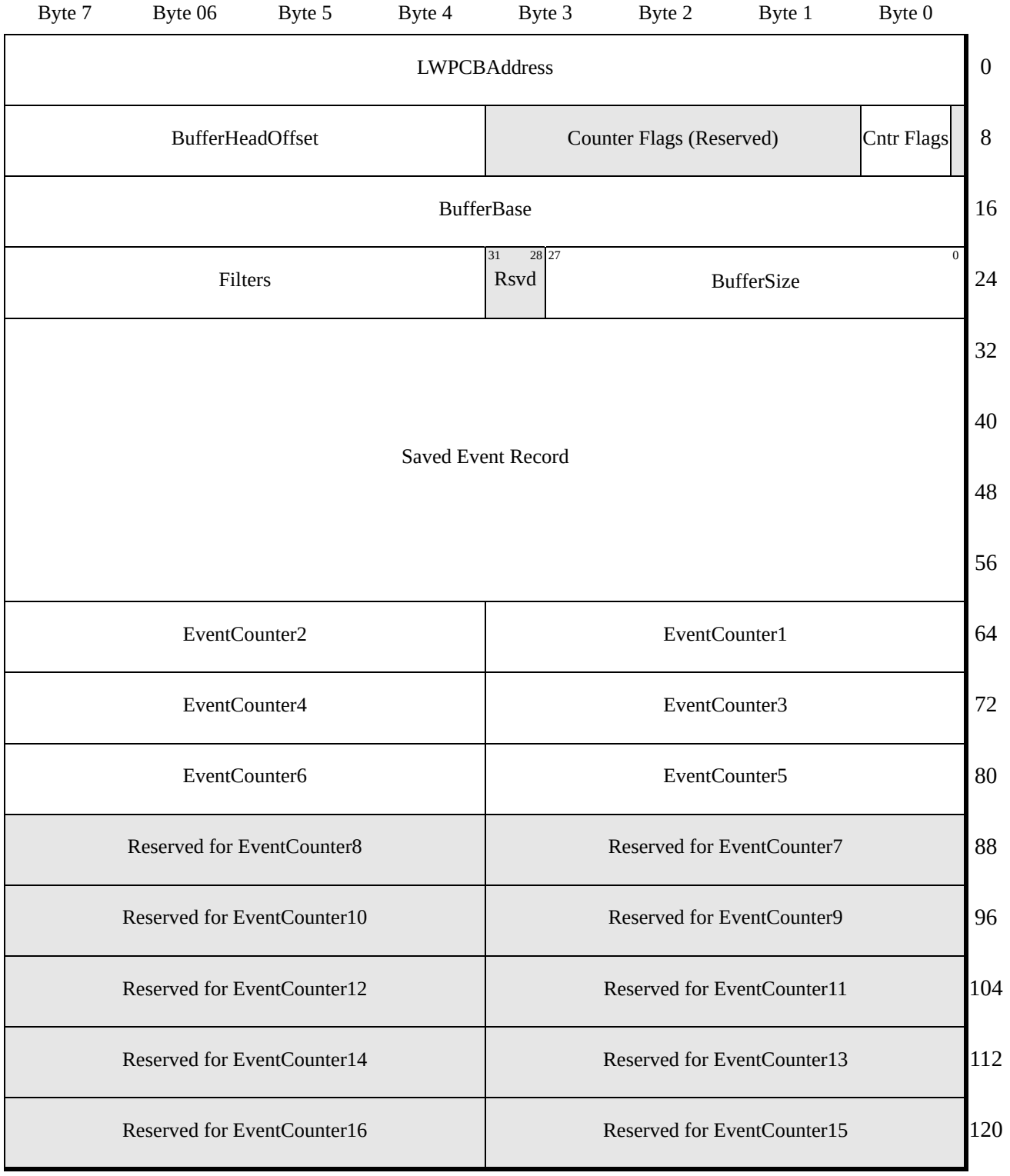


Figure 13-33. XSAVE Area for LWP

Table 13-10. XSAVE Area for LWP Fields

Bytes	Bits	Field	Description
7–0		LWPCBAddress	Address of LWPCB. 0 if LWP is disabled, in which case the rest of the save area is ignored. This is a linear address.
9–8	0	—	Reserved
9–8	1	CntrFlags.Counter1	1—Event with EventId 1 is active. XRSTOR will make the event active and restore its counter from EventCounter1. 0—Event 1 is not active. XRSTOR will make the event inactive.
9–8	6:2	CntrFlags.Counter*n*	Bit flags defined as above for EventCounter2–6.
9–8	15:7	—	Reserved for counter flags
11–10	15:0	—	Reserved for counter flags
15–12		BufferHeadOffset	BufferHeadOffset value
23–16		BufferBase	Address of the event ring buffer. This is a linear address.
27–24	27:0	BufferSize	Size of the event ring buffer
27–24	31:28	—	Reserved
31–28		Filters	Profiling filters (same as the Filters field in the LWPCB)
63–32		SavedEventRecord	If an event record is pending, the data to write. May be sparse. Zero in the EventId field means no record pending.
67–64		EventCounter1	Counter for event 1 (valid if CntrFlags.Counter1 bit is set)
87–68		EventCounter*n*	Counters for events 2–6 (valid if the respective Counter*n* bit is set)
127–88		—	Reserved for future event counters

13.4.7.3 XSAVE operation

If LWP is not currently enabled (i.e., if LWP_CBADDR = 0), no state needs to be stored. XSAVE sets bit 62 in XSAVE.HEADER.XSTATE_BV to 0 so that an attempt to restore state from this save area will use the processor supplied values. See “Processor supplied values” on page 459.

If LWP is enabled, XSAVE stores the various internal LWP values into the XSAVE area with no checking or conversion and sets bit 62 in XSAVE.HEADER.XSTATE_BV to 1.

13.4.7.4 XRSTOR operation

If bit 62 in XFEATURE_ENABLED_MASK (XCR0) is 0 or if bit 62 of EDX:EAX (EDX[30]) is 0, XRSTOR does not alter the LWP state.

If the above bits are 1 but bit 62 in XSAVE.HEADER.XSTATE_BV is 0, XRSTOR writes the LWP state using the processor supplied values, disabling LWP. See “Processor supplied values” on page 459.

If all of the above bits are 1, XRSTOR loads LWP state from the XSAVE area as follows:

1. The internal pointers and sizes are loaded.

- If BufferSize is below the implementation minimum, LWP is disabled and XRSTOR of LWP state terminates.
- If BufferSize is not a multiple of the event record size, it is rounded down.
- If BufferHeadOffset is greater than (BufferSize - LwpEventSize), a value of 0 is used instead.
- If BufferHeadOffset is not a multiple of the event record size, it is rounded down.

2. For each bit that is set in the Flags field that corresponds to an available event (as currently set in the LWP_CFG MSR), the corresponding event is enabled and the event counter is loaded from the EventCounter*n* field. All other events are disabled.
3. If the EventId field in the SavedEventRecord is non-zero, there was a pending event when XSAVE was executed. XRSTOR loads the event record into hardware. LWP will store it into the event ring buffer as soon as possible once the CPL is 3.

   Software should not alter the SavedEventRecord field. An implementation may ignore a saved event record if it was not constructed by XSAVE. Storing an event into SavedEventRecord and then executing XRSTOR is not a reliable way of injecting an event into the ring buffer.

Note that if LWP is already enabled when executing XRSTOR, the old LWP state is overwritten without being saved.

No interrupt is generated by XRSTOR if the restored value of BufferHeadOffset results in a buffer that is filled beyond the threshold. The interrupt will occur the next time an event record is stored.

XRSTOR may not restore all of the state necessary for LWP to operate. The LWP hardware will read additional state from the LWPCB when it stores then next event record.

If the CPL = 0, XRSTOR simply reloads the LWPCB address and the ring buffer address from the XSAVE area. Kernel software is trusted not to alter the area in such a way as to allow access to memory that the application could not otherwise read or write. The linear addresses in the XSAVE area were validated when the application executed LLWPCB.

If the CPL ≠ 0, XRSTOR first validates the LWPCB and ring buffer pointers. This prevents an application from altering the XSAVE area in order to gain access to memory that it could not otherwise read or write (based on the current values in the DS segment register). Note that if a program's DS value changes after doing a successful LLWPCB, it might be incapable of doing an XSAVE and then an XRSTOR of LWP state. The XRSTOR will fail if the new DS value no longer allows access to the linear addresses corresponding to the LWPCB or the ring buffer. Programs should avoid this behavior.

If XRSTOR is executed when the CPL ≠ 0, the system performs additional checks on the LWPCB and ring buffer addresses according to the pseudo-code below. A "Store-type Segment_check" fails if the limit check fails (address is beyond the segment limit) or if the segment is read-only.

```
bool Check(uint64 addr, uint32 size) { // Utility function
      if (!64bit_Mode)
            addr = truncate32(addr - DS.BASE)
      uint64 top = addr + size - 1;
      if (! Store-type Segment_check on DS:[addr] || // Check lower bound
          ! Store-type Segment_check on DS:[top])    // and upper bound
```

```
            return false;
      return true;
}

if (! Check(XSAVE.LWPCBAddress, sizeof(LWPCB)) ||
    ! Check(XSAVE.BufferAddress, XSAVE.BufferSize))
      Disable LWP
```

If any of the address checks fails, LWP is disabled. No fault is generated. A program that executes XRSTOR when the CPL ≠ 0 and DS has changed can use SLWPCB to check whether LWP is running.

As with all features that use XSAVE and XRSTOR, if bit 62 of XFEATURE_ENABLED_MASK (XCR0) is 0 but bit 62 of XSAVE.HEADER.XSTATE_BV is 1, XRSTOR will cause a #GP(0) exception.

13.4.7.5 Processor supplied values

If XRSTOR is executed when bit 62 of XFEATURE_ENABLED_MASK (XCR0) and EDX:EAX are both 1, but the corresponding bit in XSAVE.HEADER.XSTATE_BV is 0, it indicates that there is no LWP state to restore. In this case, LWP_CBADDR is set to 0 and LWP is disabled. Other processor internal state for LWP is set to 0 as necessary to avoid security issues.

13.4.8 Implementation Notes

The following subsections describe other LWP considerations.

13.4.8.1 Multiple Simultaneous Events

Multiple events are possible when an instruction retires. For instance, an indirect jump through a pointer in memory can trigger the instructions retired, branches retired, and DCache miss events simultaneously. LWP counts all events that apply to the instruction, but might not store event records for all events whose event counters became negative. It is implementation dependent as to how many event records are stored when multiple event counters simultaneously become negative. If not all events cause event records to be stored, the choice of which event(s) to report is implementation dependent and may vary from run to run on the same processor.

13.4.8.2 Processor State for Context Switch, SVM, and SMM

Implementations of LWP have internal state to hold information such as the current values of the counters for the various events, a pointer into the event ring buffer, and a copy of the tail pointer for quick detection of threshold and overflow states.

There are times when the system must preserve the volatile LWP state. When the operating system context switches from one user thread to another, the old user state must be saved with the thread's context and the new state must be loaded. When a hypervisor decides to switch from one guest OS to another, the same must be done for the guest systems' states. Finally, state must be stored and reloaded when the system enters and exits SMM, since the SMM code may decide to shut off power to the core.

Hardware does not maintain the LWP state in the active LWPCB. This is because the counters change with every event (not just every reported event), so keeping them in memory would generate a large amount of unnecessary memory traffic. Also, the LWPCB is in user memory and may be paged out to disk at any time, so the memory may not be available when needed.

Saving State at Thread Context Switches

LWP requires that an operating system use the XSAVE and XRSTOR instructions to save and restore LWP state across context switches.

XRSTOR restores the LWP volatile state when restoring other system state. Some additional LWP state will be restored from the LWPCB when operations in ring 3 require that information.

LWP does not support the “lazy” state save and restore that is possible for floating point and SSE state. It does not interact with the CR0[TS] bit. Operating systems that support LWP must always do an XSAVE to preserve the old thread’s LWP context and an XRSTOR to set up the new LWP context. The OS can continue to do a lazy switch of the FP and SSE state by ensuring that the corresponding bits in EDX:EAX are clear when it executes the XSAVE and XRSTOR to handle the LWP context.

Saving State at SVM Worldswitch to a Different Guest

Hypervisors that allow guests to use LWP must save and restore LWP state when the guest OS changes. In addition to the usual information in the VMCB, the hypervisor must use XSAVE/XRSTOR to maintain the volatile LWP state and must also save and restore LWP_CFG. When switching between a guest that uses LWP and one that does not, the hypervisor changes the value of XFEATURE_ENABLED_MASK (XCR0), which ensures that LWP is only enabled in the appropriate guest(s).

A hypervisor need not modify the LWP state if the guest OS is not changed.

Enabling SVM Live Migration

Some hypervisors support live migration of a guest virtual machine. Live migration is when a hypervisor preserves the entire state of the guest running on one physical machine, copies that state to another physical machine, and then resumes execution of the guest on the new hardware.

To allow live migration among machines which may have different internal implementations of LWP, the hypervisor must present the common subset of features among all the hosts in the pool of machines that can be used. Furthermore, since the hypervisor may XSAVE LWP state on one machine and XRSTOR it on another machine, the contents of the XSAVE area must be consistent across all implementations.

This means that an implementation of LWP keeps all event counters internally, not in the LWPCB. If implementations were permitted to differ in this detail, a counter might not get properly restored after migrating the guest machine.

Saving State at SMM Entry and Exit

SMM entry and exit must save and restore LWP state when the processor is going to change power state. SMM must use XSAVE/XRSTOR and must also save and restore LWP_CFG. Since LWP is ring 3 only and is inactive in System Management Mode, its state should not need to be saved and restored otherwise.

Notes on Restoring LWP State

The LWPCB may not be in memory at all times. Therefore, the LWP hardware does not attempt to access it while still in the OS kernel/VMM/SMM, since that access might fault. Some LWP state is restored once the processor is in ring 3 and can take a #PF exception without crashing. This usually happens the next time LWP needs to store an event record into the ring buffer.

13.4.8.3 LWPCB Access

Several LWPCB fields are written asynchronously by the LWP hardware and by the user software. This section discusses techniques for reducing the associated memory traffic. This is interesting to software because it influences what state is kept internally in LWP, and it explains the protocol between the hardware filling the event ring buffer and the software emptying it.

The hardware keeps an internal copy of the event ring buffer head pointer. It need not flush the head pointer to the LWPCB every time it stores an event record. The flush can be done periodically or it can be deferred until a threshold or buffer full condition happens or until the application executes LLWPCB or SLWPCB. Exceeding the buffer threshold always forces the head pointer to memory so that the interrupt handler emptying the ring buffer sees the threshold condition.

The hardware may keep an internal copy of the event ring buffer tail pointer. It need not read the software-maintained tail pointer unless it detects a threshold or buffer full condition. At that point, it rereads the tail pointer to see if software has emptied some records from the ring buffer. If so, it recomputes the condition and acts accordingly. This implies that software polling the ring buffer should begin processing event records when it detects a threshold condition itself. To avoid a race condition with software, the hardware rereads the tail pointer every time it stores an event record while the threshold condition appears to be true. (An implementation can relax this to “every n^{th} time” for some small value of n.) It also rereads it whenever the ring buffer appears to be full.

The interval values used to reset the counters can be cached in the hardware when the LLWPCB instruction is executed, or they can be read from the LWPCB each time the counter overflows.

The ring buffer base and size are cached in the hardware.

The MissedEvents value is a counter for an exceptional condition and is kept in memory.

The cached LWP state is refreshed from the LWPCB when LWP is enabled either explicitly via LLWPCB or implicitly when needed in ring 3 after LWP state is restored via XRSTOR.

Caching implies that software cannot reliably change sampling intervals or other cached state by modifying the LWPCB. The change might not be noticed by the LWP hardware. On the other hand, changing state in the LWPCB while LWP is running may change the operation at an unpredictable

moment in the future if LWP context is saved and restored due to context switching. Software must stop and restart LWP to ensure that any changes reliably take effect.

13.4.8.4 Security

The operating system must ensure that information does not leak from one process to another or from the kernel to a user process. Hence, if it supports LWP at all, the operating system must ensure that the state of the LWP hardware is set appropriately when a context switch occurs and when a new process or thread is created. LWP state for a new thread can be initialized by executing XRSTOR with bit 62 of XSAVE.HEADER.XSTATE_BV set to 0 and the corresponding bit in EDX:EAX set to 1.

13.4.8.5 Interrupts

The LWP threshold interrupt vector number is specified in the LWP_CFG MSR. The operating system must assign a vector for LWP threshold interrupts and fill in the corresponding entry in the interrupt-descriptor table. Note that the LWP interrupt is not shared with the performance counter interrupt, since the system allows concurrent and independent use of those two mechanisms.

13.4.8.6 Memory Access During LWP Operation

When LWP needs to save an event record in the event ring buffer, it accesses the user memory containing the ring buffer and sometimes the memory containing the LWPCB. This causes a Page Fault (#PF) exception if those pages are not in memory.

A particular implementation of LWP has several ways to deal with page faults when storing an event record. These may include saving the event record in the XSAVE area and retrying the store later, reexecuting the instruction, or discarding the event and reporting the next event of the appropriate type.

Note that this reinforces the notion that LWP is a sampling mechanism. Programs cannot rely on it to precisely capture every n^{th} instance of an event. It captures *approximately* every n^{th} instance.

13.4.8.7 Guidelines for Operating Systems

To support LWP, an operating system should follow the following guidelines. Most of these operations should be done on each core of a multi-core system.

System initialization

1. Use CPUID Fn0000_0000 to ensure that the system is running on an “Authentic AMD” processor, and then check CPUID Fn8000_0001_ECX[LWP] to ensure that the processor supports LWP.

   Alternatively, check CPUID Fn0000_000D_EDX_x0[30] to ensure that the system supports the LWP XSAVE area, indicating that the processor supports LWP.

2. Enable XSAVE operations by setting CR4[OSXSAVE].
3. Enable LWP by executing XSETBV to set bit 62 of XCR0.

4. Assign a unique interrupt vector number for LWP threshold interrupts and load the corresponding entry in the interrupt-descriptor table with the address of the interrupt handler. This handler should use some system-specific method to forward any threshold interrupts to the application.
5. Make LWP available by setting LWP_CFG. To enable all supported LWP features, set LWP_CFG[31:0] to the value returned by CPUID Fn8000_001C_EDX. Set LWP_CFG[COREID] to the APIC core number (or some other value unique to the core) and LWP_CFG[VECTOR] to the assigned interrupt vector number.

Thread support

- For each thread, allocate an XSAVE area that is at least as big as the XFeatureEnabledSizeMax value returned by CPUID Fn0000_000D_EBX_x0 (ECX=0). This is good practice for any system that supports XSAVE.
- When creating a new process or thread, execute XRSTOR with bit 62 of EDX:EAX set to 1 and bit 62 of XSAVE.HEADER.XSTATE_BV set to 0. This ensures that LWP is turned off for any new thread. Alternatively, use WRMSR to write 0 into LWP_CBADDR before starting the thread.
- When saving a running thread's context, execute XSAVE with bit 62 of EDX:EAX set to 1 to save the thread's LWP state. It takes almost no time or resources if the thread is not using LWP.
- When restoring a thread's context, execute XRSTOR with bit 62 of EDX:EAX set to 1. This restores the LWP state for the thread or disables LWP if the thread is not using it.
- When a thread exits or aborts, use WRMSR to store 0 into LWP_CBADDR. This ensures that LWP is turned off.

13.4.8.8 Summary of LWP State

LWP adds the following visible state to the AMD64 architecture:

- CPUID Fn8000_0001_ECX[LWP] (bit 15) to indicate LWP support.
- CPUID Fn8000_001C to indicate LWP features.
- Two new MSRs: LWP_CFG, LWP_CBADDR,.
- Four new instructions: LLWPCB, SLWPCB, LWPINS, and LWPVAL.
- Bit 62 in XCR0 (XFEATURE_ENABLED_MASK)
- A new XSAVE area for LWP state.
- New fields for LWP state in the SVM and SMM context, whether in the VMCB and SMM save area or elsewhere.

See Section 3.3, "Processor Feature Identification," on page 71 for information on using the CPUID instruction to obtain information about processor capabilites.

14 Processor Initialization and Long Mode Activation

This chapter describes the hardware actions taken following a processor reset and the steps that must be taken to initialize processor resources and activate long mode. In some cases the actions required are implementation-specific with references made to the appropriate implementation-specific documentation.

14.1 Processor Initialization

System logic can initialize the processor in either of two ways. One method, called RESET, is usually initiated by the assertion of an external signal (typically designated RESET#). The other method, called INIT, is typically initiated by another processor by means of an INIT interprocessor interrupt (IPI). See “Interprocessor Interrupts (IPI)” on page 615 for more information.

Both initialization techniques place the processor in real mode and initialize processor resources to a known, consistent state from which software can begin execution. The processor begins execution when the RESET# pin is deasserted or the INIT state is exited.

The RESET method places the processor in a known state and prepares it to begin execution in real mode. The INIT method is similar except it does not modify the state of certain registers. See Section 14.1.3 on page 466 for a comparison of these initialization methods.

System logic ensures that the processor transitions through the RESET state whenever power is reapplied after a planned or unplanned interruption. A RESET can also be performed when power is stable. An INIT can be performed at any time after the processor is powered up.

14.1.1 Built-In Self Test (BIST)

An optional built-in self-test can be performed after the processor is reset. The mechanism for triggering the BIST is implementation-specific, and can be found in the hardware documentation for the implementation. The number of processor cycles BIST can consume before completing is also implementation-specific but typically consumes several million cycles.

BIST can be used by system implementations to assist in verifying system integrity, thereby improving system reliability, availability, and serviceability. The internal BIST hardware generally tests all internal array structures for errors. These structures can include (but are not limited to):

- All internal caches, including the tag arrays as well as the data arrays.
- All TLBs.
- Internal ROMs, such as the microcode ROM and floating-point constant ROM.
- Branch-prediction structures.

EAX is loaded with zero if BIST completes without detecting errors. If any hardware faults are detected during BIST, a non-zero value is loaded into EAX.

14.1.2 Clock Multiplier Selection

The internal processor clock runs at some multiple of the system clock. The processor-to-system clock multiple does not have to be fixed by a processor implementation but instead can be programmable through hardware or software, or some combination of the two. For information on selecting the processor-clock multiplier, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.

14.1.3 Processor Initialization State

Table 14-1 shows the initial processor state following either RESET or INIT. Except as indicated, processor resources generally are set to the same value after either RESET or INIT.

Table 14-1. Initial Processor State

Processor Resource	Value After RESET	Value After INIT
CR0	0000_0000_6000_0010h	CD and NW are unchanged Bit 4 (reserved) = 1 All others = 0
CR2, CR3, CR4	0	
CR8	0	Not modified
RFLAGS	0000_0000_0000_0002h	
EFER	0	
RIP	0000_0000_0000_FFF0h	
CS	Selector = F000h Base = 0000_0000_FFFF_0000h Limit = FFFFh Attributes = See Table 14-2 on page 468	
DS, ES, FS, GS, SS	Selector = 0000h Base = 0 Limit = FFFFh Attributes = See Table 14-2 on page 468	
GDTR, IDTR	Base = 0 Limit = FFFFh	
LDTR, TR	Selector = 0000h Base = 0 Limit = FFFFh Attributes = See Table 14-2 on page 468	
RAX	0 (non-zero if BIST is run and fails)	0

Table 14-1. Initial Processor State (continued)

Processor Resource	Value After RESET	Value After INIT
RDX	Family/Model/Stepping, including extended family and extended model—see “Processor Implementation Information” on page 469	
RBX, RCX, RBP, RSP, RDI, RSI, R8, R9, R10, R11, R12, R13, R14, R15	0	
x87 Floating-Point State	FPR0–FPR7 = 0 Control Word = 0040h Status Word = 0000h Tag Word = 5555h Instruction CS = 0000h Instruction Offset = 0 x87 Instruction Opcode = 0 Data-Operand DS = 0000h Data-Operand Offset = 0	Not modified
64-Bit Media State	MMX0–MMX7 = 0	Not modified
SSE State	XMM0–XMM15 = 0 MXCSR = 1F80h	Not modified
Memory-Type Range Registers	See “Memory-Typing MSRs” on page 669	Not modified
Machine-Check Registers	See “Machine-Check MSRs” on page 671	Not modified
DR0, DR1, DR2, DR3	0	
DR6	0000_0000_FFFF_0FF0h	
DR7	0000_0000_0000_0400h	
Time-Stamp Counter	0	Not modified
Performance-Monitor Resources	See “Performance-Monitoring MSRs” on page 673	Not modified
Other Model-Specific Registers	See “MSR Cross-Reference” on page 661	Not modified
Instruction and Data Caches	Invalidated	Not modified
Instruction and Data TLBs		
APIC	Disabled, see Table 16-2 on page 605.	Enabled, see Table 16-2 on page 605.
SMRAM Base Address (SMBASE)	0003_0000h	Not modified
XCR0	0000_0000_0000_0001h	Not modified
PKRU	0000_0000h	Not modified

Table 14-2 on page 468 shows the initial state of the segment-register attributes (located in the hidden portion of the segment registers) following either RESET or INIT.

Table 14-2. Initial State of Segment-Register Attributes

Attribute		Value (Binary)	Description
G		0	Byte Granularity
D/B		0	16-Bit Segment
L (CS Only)		0	Legacy-Mode Segment
P		1	Segment is Present
DPL		00	Privilege-Level 0
S and Type	Code Segment	S = 1 Type = 1010	Executable/Readable Code Segment
	Data Segment	S = 1 Type = 0010	Read/Write Data Segment
	LDTR	S = 0 Type = 0010	LDT
	TR	S = 0 Type = 0011	Busy 16-Bit TSS

14.1.4 Multiple Processor Initialization

Following reset in multiprocessor configurations, the processors use a multiple-processor initialization protocol to negotiate which processor becomes the *bootstrap* processor. This bootstrap processor then executes the system initialization code while the remaining processors wait for software initialization to complete. For further information, see the documentation for particular implementations of the architecture.

14.1.5 Fetching the First Instruction

After a RESET or INIT, the processor is operating in 16-bit real mode. Normally within real mode, the code-segment base-address is formed by shifting the CS-selector value left four bits. The base address is then added to the value in EIP to form the physical address into memory. As a result, the processor can only address the first 1 Mbyte of memory when in real mode.

However, immediately following RESET or INIT, the CS-selector register is loaded with F000h, but the CS base-address is *not* formed by left-shifting the selector. Instead, the CS base-address is initialized to FFFF_0000h. EIP is initialized to FFF0h. Therefore, the first instruction fetched from memory is located at physical-address FFFF_FFF0h (FFFF_0000h + 0000_FFF0h).

The CS base-address remains at this initial value until the CS-selector register is loaded by software. This can occur as a result of executing a far jump instruction or call instruction, for example. When CS is loaded by software, the new base-address value is established as defined for real mode (by left shifting the selector value four bits).

14.2 Hardware Configuration

14.2.1 Processor Implementation Information

Software can read processor-identification information from the EDX register immediately following RESET or INIT. This information can be used to initialize software to perform processor-specific functions. The information stored in EDX is defined as follows:

- *Stepping ID (bits 3:0)*—This field identifies the processor-revision level.
- *Extended Model (bits 19:16) and Model (bits 7:4)*—These fields combine to differentiate processor models within a instruction family. For example, two processors may share the same microarchitecture but differ in their feature set. Such processors are considered different models within the same instruction family. This is a split field, comprising an extended-model portion in bits 19:16 with a legacy portion in bits 7:4
- *Extended Family (bits 27:20) and Family (bits 11:8)*—These fields combine to differentiate processors by their microarchitecture.

The CPUID instruction can be used to obtain the same information. This is done by executing CPUID with either function 1 or function 8000_0001h. Additional information about the processor and the features supported can be gathered using CPUID with other feature codes. See Section 3.3, “Processor Feature Identification,” on page 71 for additional information.

14.2.2 Enabling Internal Caches

Following a RESET (but not an INIT), all instruction and data caches are disabled, and their contents are invalidated (the MOESI state is set to the invalid state). Software can enable these caches by clearing the cache-disable bit (CR0.CD) to zero (RESET sets this bit to 1). Software can further refine caching based on individual pages and memory regions. Refer to “Cache Control Mechanisms” on page 202 for more information on cache control.

Memory-Type Range Registers (MTRRs). Following a RESET (but not an INIT), the MTRRdefType register is cleared to 0, which disables the MTRR mechanism. The variable-range and fixed-range MTRR registers are not initialized and are therefore in an undefined state. Before enabling the MTRR mechanism, the initialization software (usually platform firmware) must load these registers with a known value to prevent unexpected results. Clearing these registers, for example, sets memory to the uncacheable (UC) type.

14.2.3 Initializing Media and x87 Processor State

Some resources used by x87 floating-point instructions and media instructions must be initialized by software before being used. Initialization software can use the CPUID instruction to determine whether the processor supports these instructions, and then initialize their resources as appropriate.

x87 Floating-Point State Initialization. Table 14-3 on page 470 shows the differences between the initial x87 floating-point state following a RESET and the state established by the FINIT/FNINIT instruction. An INIT does not modify the x87 floating-point state. The initialization software can

execute an FINIT or FNINIT instruction to prepare the x87 floating-point unit for use by application software. The FINIT and FNINIT instructions have no effect on the 64-bit media state.

Table 14-3. x87 Floating-Point State Initialization

x87 Floating-Point Resource	RESET	FINIT/FNINIT Instructions
FPR0–FPR7	0	Not modified
Control Word	0040h • Round to nearest • Single precision • Unmask all exceptions	037Fh • Round to nearest • Extended precision • Mask all exceptions
Status Word	0000h	
Tag Word	5555h (FPR*n* contain zero)	FFFFh (FPR*n* are empty)
Instruction CS	0000h	
Instruction Offset	0	
x87 Instruction Opcode	0	
Data-Operand DS	0000h	
Data-Operand Offset	0	

Initialization software should also load the MP, EM, and NE bits in the CR0 register as appropriate for the operating system. The recommended settings are:

- *MP=1*—Setting MP to 1 causes a device-not-available exception (#NM) to occur when the FWAIT/WAIT instruction is executed and the task-switched bit (CR0.TS) is set to 1. This supports operating systems that perform lazy context-switching of x87 floating-point state.
- *EM=0*—Clearing EM to 0 allows the x87 floating-point unit to execute instructions rather than causing a #NM exception (CR0.EM=1). System software sets EM to 1 only when software emulation of x87 instructions is desired.
- *NE=1*—Setting NE to 1 causes x87 floating-point exceptions to be handled by the floating-point exception-pending exception (#MF) handler. Clearing this bit causes the processor to externally indicate the exception occurred, and an external device can then cause an external interrupt to occur in response.

Refer to “CR0 Register” on page 42 for additional information on these control bits.

64-Bit Media State Initialization. There are no special requirements placed on software to initialize the processor state used by 64-bit media instructions. This state is initialized completely by the processor following a RESET. System software should leave CR0.EM cleared to 0 to allow execution of the 64-bit media instructions. If CR0.EM is set to 1, attempted execution of the 64-bit media instructions causes an invalid-opcode exception (#UD).

The 64-bit media state is not modified by an INIT.

SSE State Initialization. Platform firmware or system software must also prepare the processor to allow execution of SSE instructions. The required preparations include:

- Leaving CR0.EM cleared to 0 to allow execution of the SSE instructions. If CR0.EM is set to 1, attempted execution of the SSE instructions except FXSAVE/FXRSTOR causes an invalid-opcode exception (#UD). An attempt to execute either of these instructions when CR0.EM is set results in a #NM exception.
- Enabling the SSE instructions by setting CR4.OSFXSR to 1. Software cannot execute the SSE instructions unless this bit is set. Setting this bit also indicates that system software uses the FXSAVE and FXRSTOR instructions to save and restore, respectively, the SSE state. These instructions also save and restore the 64-bit media state and x87 floating-point state.
- Indicating that system software uses the SIMD floating-point exception (#XF) for handling SSE floating-point exceptions. This is done by setting CR4.OSXMMEXCPT to 1.
- Setting (optionally) the MXCSR mask bits to mask or unmask SSE floating-point exceptions as desired. Because this register can be read and written by application software, it is not absolutely necessary for system software to initialize it.

Refer to “CR4 Register” on page 47 for additional information on these CR4 control bits.

14.2.4 Model-Specific Initialization

Implementations of the AMD64 architecture can contain model-specific features and registers that are not initialized by the processor and therefore require system-software initialization. System software must use the CPUID instruction to determine which features are supported. Model-specific features are generally configured using model-specific registers (MSRs), which can be read and written using the RDMSR and WRMSR instructions, respectively.

Some of the model-specific features are pervasive across many processor implementations of the AMD64 architecture and are therefore described within this volume. These include:

- System-call extensions, which must be enabled in the EFER register before using the SYSCALL and SYSRET instructions. See “System-Call Extension (SCE) Bit” on page 57 for information on enabling these instructions.
- Memory-typing MSRs. See “Memory-Type Range Registers (MTRRs)” on page 469 for information on initializing and using these registers.
- The machine-check mechanism. See “Initializing the Machine-Check Mechanism” on page 311 for information on enabling and using this capability.
- Extensions to the debug mechanism. See “Software-Debug Resources” on page 386 for information on initializing and using these extensions.
- The performance-monitoring resources. See “Performance Monitoring Counters” on page 400 for information on initializing and using these resources.

Initialization of other model-specific features used by the page-translation mechanism and long mode are described throughout the remainder of this section.

Some model-specific features are not pervasive across processor implementations and are therefore not described in this volume. For more information on these features and their initialization requirements, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.

14.3 Initializing Real Mode

A basic real-mode (real-address-mode) operating environment must be initialized so that system software can initialize the protected-mode operating environment. This real-mode environment must include:

- A real-mode IDT for vectoring interrupts and exceptions to the appropriate handlers while in real mode. The IDT base-address value in the IDTR initialized by the processor can be used, or system software can relocate the IDT by loading a new base-address into the IDTR.
- The real-mode interrupt and exception handlers. These must be loaded before enabling external interrupts.

  Because the processor can always accept a non-maskable interrupt (NMI), it is possible an NMI can occur before initializing the IDT or the NMI handler. System hardware must provide a mechanism for disabling NMIs to allow time for the IDT and NMI handler to be properly initialized. Alternatively, the IDT and NMI handler can be stored in non-volatile memory that is referenced by the initial values loaded into the IDTR.

  Maskable interrupts can be enabled by setting EFLAGS.IF after the real-mode IDT and interrupt handlers are initialized.
- A valid stack pointer (SS:SP) to be used by the interrupt mechanism should interrupts or exceptions occur. The values of SS:SP initialized by the processor can be used.
- One or more data-segment selectors for storing the protected-mode data structures that are created in real mode.

Once the real-mode environment is established, software can begin initializing the protected-mode environment.

14.4 Initializing Protected Mode

Protected mode must be entered before activating long mode. A minimal protected-mode environment must be established to allow long-mode initialization to take place. This environment must include the following:

- A protected-mode IDT for vectoring interrupts and exceptions to the appropriate handlers while in protected mode.
- The protected-mode interrupt and exception handlers referenced by the IDT. Gate descriptors for each handler must be loaded in the IDT.
- A GDT which contains:

- A code descriptor for the code segment that is executed in protected mode.
- A read/write data segment that can be used as a protected-mode stack. This stack can be used by the interrupt mechanism if interrupts or exceptions occur.

Software can optionally load the GDT with one or more data segment descriptors, a TSS descriptor, and an LDT descriptor for use by long-mode initialization software.

After the protected-mode data structures are initialized, system software must load the IDTR and GDTR with pointers to those data structures. Once these registers are initialized, protected mode can be enabled by setting CR0.PE to 1.

If legacy paging is used during the long-mode initialization process, the page-translation tables must be initialized before enabling paging. At a minimum, one page directory and one page table are required to support page translation. The CR3 register must be loaded with the starting physical address of the highest-level table supported in the page-translation hierarchy. After these structures are initialized and protected mode is enabled, paging can be enabled by setting CR0.PG to 1.

14.5 Initializing Long Mode

From protected mode, system software can initialize the data structures required by long mode and store them anywhere in the first 4 Gbytes of physical memory. These data structures can be relocated above 4 Gbytes once long mode is activated. The data structures required by long mode include the following:

- An IDT with 64-bit interrupt-gate descriptors. Long-mode interrupts are always taken in 64-bit mode, and the 64-bit gate descriptors are used to transfer control to interrupt handlers running in 64-bit mode. See “Long-Mode Interrupt Control Transfers” on page 272 for more information.
- The 64-bit mode interrupt and exception handlers to be used in 64-bit mode. Gate descriptors for each handler must be loaded in the 64-bit IDT.
- A GDT containing segment descriptors for software running in 64-bit mode and compatibility mode, including:
  - Any LDT descriptors required by the operating system or application software.
  - A TSS descriptor for the single 64-bit TSS required by long mode.
  - Code descriptors for the code segments that are executed in long mode. The code-segment descriptors are used to specify whether the processor is operating in 64-bit mode or compatibility mode. See “Code-Segment Descriptors” on page 97, “Long (L) Attribute Bit” on page 98, and “CS Register” on page 79 for more information.
  - Data-segment descriptors for software running in compatibility mode. The DS, ES, and SS segments are ignored in 64-bit mode. See “Data-Segment Descriptors” on page 98 for more information.
  - FS and GS data-segment descriptors for 64-bit mode, if required by the operating system. If these segments are used in 64-bit mode, system software can also initialize the full 64-bit base

addresses using the WRMSR instruction. See “FS and GS Registers in 64-Bit Mode” on page 80 for more information.

The existing protected-mode GDT can be used to hold the long-mode descriptors described above.

- A single 64-bit TSS for holding the privilege-level 0, 1, and 2 stack pointers, the interrupt-stack-table pointers, and the I/O-redirection-bitmap base address (if required). This is the only TSS required, because hardware task-switching is not supported in long mode. See “64-Bit Task State Segment” on page 371 for more information.
- The 4-level page-translation tables required by long mode. Long mode also requires the use of physical-address extensions (PAE) to support physical-address sizes greater than 32 bits. See “Long-Mode Page Translation” on page 141 for more information.

If paging is enabled during the initialization process, it *must* be disabled before enabling long mode. After the long-mode data structures are initialized, and paging is disabled, software can enable and activate long mode.

14.6 Enabling and Activating Long Mode

Long mode is *enabled* by setting the long-mode enable control bit (EFER.LME) to 1. However, long mode is not *activated* until software also enables paging. When software enables paging while long mode is enabled, the processor activates long mode, which the processor indicates by setting the long-mode-active status bit (EFER.LMA) to 1. The processor behaves as a 32-bit x86 processor in all respects until long mode is activated, even if long mode is enabled. None of the new 64-bit data sizes, addressing, or system aspects available in long mode can be used until EFER.LMA=1.

Table 14-4 shows the control-bit settings for enabling and activating the various operating modes of the AMD64 architecture. The default address and data sizes are shown for each mode. For the methods of overriding these default address and data sizes, see “Instruction Prefixes” in Volume 1.

Table 14-4. Processor Operating Modes

Mode		Encoding			Default Address Size (bits)[2]	Default Data Size (bits)[2]
		EFER.LMA[1]	CS.L	CS.D		
Long Mode	64-Bit Mode	1	1	0	64	32
	Compatibility Mode		0	1	32	32
				0	16	16
Legacy Mode		0	x	1	32	32
				0	16	16

Note:
1. *EFER.LMA is set by the processor when software sets EFER.LME and CR0.PG according to the sequence described in "Activating Long Mode" on page 475.*
2. *See "Instruction Prefixes" in Volume 1 for overrides to default sizes.*

Long mode uses two code-segment-descriptor bits, CS.L and CS.D, to control the operating submodes. If long mode is active, CS.L = 1, and CS.D = 0, the processor is running in 64-bit mode, as shown in Table 14-4 on page 475. With this encoding (CS.L=1, CS.D=0), default operand size is 32 bits and default address size is 64 bits. Using instruction prefixes, the default operand size can be overridden to 64 bits or 16 bits, and the default address size can be overridden to 32 bits.

The final encoding of CS.L and CS.D in long mode (CS.L=1, CS.D=1) is reserved for future use.

When long mode is active and CS.L is cleared to 0, the processor is in compatibility mode, as shown in Table 14-4 on page 475. In compatibility mode, CS.D controls default operand and address sizes exactly as it does in the legacy x86 architecture. Setting CS.D to 1 specifies default operand and address sizes as 32 bits. Clearing CS.D to 0 specifies default operand and address sizes as 16 bits.

14.6.1 Activating Long Mode

Switching the processor to long mode requires several steps. In general, the sequence involves disabling paging (CR0.PG=0), enabling physical-address extensions (CR4.PAE=1), loading CR3, enabling long mode (EFER.LME=1), and finally enabling paging (CR0.PG=1).

Specifically, software must follow this sequence to activate long mode:

1. If starting from page-enabled protected mode, disable paging by clearing CR0.PG to 0. This requires that the MOV CR0 instruction used to disable paging be located in an identity-mapped page (virtual address equals physical address).
2. In any order:

- Enable physical-address extensions by setting CR4.PAE to 1. Long mode requires the use of physical-address extensions (PAE) in order to support physical-address sizes greater than 32 bits. Physical-address extensions must be enabled before enabling paging.
- Load CR3 with the physical base-address of the level-4 page-map-table (PML4). See “Long-Mode Page Translation” on page 141 for details on creating the 4-level page translation tables required by long mode.
- Enable long mode by setting EFER.LME to 1.

3. Enable paging by setting CR0.PG to 1. This causes the processor to set the EFER.LMA bit to 1. The instruction following the MOV CR0 that enables paging must be a branch, and both the MOV CR0 and the following branch instruction must be located in an identity-mapped page.

14.6.2 Consistency Checks

The processor performs long-mode consistency checks whenever software attempts to modify any of the control bits directly involved in activating long mode (EFER.LME, CR0.PG, and CR4.PAE). A general-protection exception (#GP) occurs when a consistency check fails. Long-mode consistency checks ensure that the processor does not enter an undefined mode or state that results in unpredictable behavior.

Long-mode consistency checks cause a general-protection exception (#GP) to occur if:

- An attempt is made to enable or disable long mode while paging is enabled.
- Long mode is enabled, and an attempt is made to enable paging before enabling physical-address extensions (PAE).
- Long mode is enabled, and an attempt is made to enable paging while CS.L=1.
- Long mode is active and an attempt is made to disable physical-address extensions (PAE).

Table 14-5 summarizes the long-mode consistency checks made during control-bit transitions.

Table 14-5. Long-Mode Consistency Checks

Control Bit	Transition	Check
EFER.LME	0 Ⓡ 1	If (CR0.PG=1) then #GP(0)
	1 Ⓡ 0	If (CR0.PG=1) then #GP(0)
CR0.PG	0 Ⓡ 1	If ((EFER.LME=1) & (CR4.PAE=0) then #GP(0) If ((EFER.LME=1) & (CS.L=1)) then #GP(0)
CR4.PAE	1 Ⓡ 0	If (EFER.LMA=1) then #GP(0)

14.6.3 Updating System Descriptor Table References

Immediately after activating long mode, the system-descriptor-table registers (GDTR, LDTR, IDTR, TR) continue to reference legacy descriptor tables. The tables referenced by these descriptors all reside in the lower 4 Gbytes of virtual-address space. After activating long mode, 64-bit operating-system software should use the LGDT, LLDT, LIDT, and LTR instructions to load the system descriptor-table

registers with references to the 64-bit versions of the descriptor tables. See “Descriptor Tables” on page 82 for details on descriptor tables in long mode.

Long mode requires 64-bit interrupt-gate descriptors to be stored in the interrupt-descriptor table (IDT). Software must not allow exceptions or interrupts to occur between the time long mode is activated and the subsequent update of the interrupt-descriptor-table register (IDTR) that establishes a reference to the 64-bit IDT. This is because the IDTR continues to reference a 32-bit IDT immediately after long mode is activated. If an interrupt or exception occurred before updating the IDTR, a legacy 32-bit interrupt gate would be referenced and interpreted as a 64-bit interrupt gate, with unpredictable results.

External interrupts can be disabled using the CLI instruction. Non-maskable interrupts (NMI) and system-management interrupts (SMI) must be disabled using external hardware. See “Long-Mode Interrupt Control Transfers” on page 272 for more information on long mode interrupts.

14.6.4 Relocating Page-Translation Tables

The long-mode page-translation tables must be located in the first 4 Gbytes of physical-address space before activating long mode. This is necessary because the MOV CR3 instruction used to initialize the page-map level-4 base address must be executed in legacy mode before activating long mode. Because the MOV CR3 is executed in legacy mode, only the low 32 bits of the register are written, which limits the location of the page-map level-4 translation table to the low 4 Gbytes of memory. Software can relocate the page tables anywhere in physical memory, and re-initialize the CR3 register, after long mode is activated.

14.7 Leaving Long Mode

To return from long mode to legacy protected mode with paging enabled, software must deactivate and disable long mode using the following sequence:

1. Switch to compatibility mode and place the processor at the highest privilege level (CPL=0).
2. Deactivate long mode by clearing CR0.PG to 0. This causes the processor to clear the LMA bit to 0. The MOV CR0 instruction used to disable paging must be located in an identity-mapped page. Once paging is disabled, the processor behaves as a standard 32-bit x86 processor.
3. Load CR3 with the physical base-address of the legacy page tables.
4. Disable long mode by clearing EFER.LME to 0.
5. Enable legacy page-translation by setting CR0.PG to 1. The instruction following the MOV CR0 that enables paging must be a branch, and both the MOV CR0 and the following branch instruction must be located in an identity-mapped page.

14.8 Long-Mode Initialization Example

Following is sample code that outlines the steps required to place the processor in long mode.

```
mydata  segment para
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; This generic data-segment holds pseudo-descriptors used
; by the LGDT and LIDT instructions.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Establish a temporary 32-bit GDT and IDT.
;
pGDT32  label   fword                   ; Used by LGDT.
                dw      gdt32_limit     ; GDT limit ...
                dd      gdt32_base      ; and 32-bit GDT base
pIDT32  label   fword                   ; Used by LIDT.
                dw      idt32_limit     ; IDT limit ...
                dd      idt32_base      ; and 32-bit IDT base
;
; Establish a 64-bit GDT and IDT (64-bit linear base-
; address)
;
pGDT64  label   tbyte                   ; Used by LGDT.
                dw      gdt64_limit     ; GDT limit ...
                dq      gdt64_base      ; and 64-bit GDT base
pIDT64  label   tbyte                   ; Used by LIDT.
                dw      idt64_limit     ; IDT limit ...
                dq      idt64_base      ; and 64-bit IDT base
mydata  ends                ; end of data segment
code16  segment para use16 ; 16-bit code segment
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 16-bit code, real mode
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Initialize DS to point to the data segment containing
; pGDT32 and PIDT32. Set up a real-mode stack pointer, SS:SP,
; in case of interrupts and exceptions.
;
   cli
   mov   ax, seg mydata
   mov   ds, ax
   mov   ax, seg mystack
   mov   ss, ax
   mov   sp, esp0
;
; Use CPUID to determine if the processor supports long mode. ;

   mov   eax, 80000000h ; Extended-function 8000000h.
   cpuid                ; Is largest extended function
   cmp   eax, 80000000h ; any function > 80000000h?
   jbe   no_long_mode   ; If not, no long mode.
   mov   eax, 80000001h ; Extended-function 8000001h.
```

```
   cpuid                ; Now EDX = extended-features flags.
   bt    edx, 29        ; Test if long mode is supported.
   jnc   no_long_mode   ; Exit if not supported.
;
; Load the 32-bit GDT before entering protected mode.
; This GDT must contain, at a minimum, the following
; descriptors:
;  1) a CPL=0 16-bit code descriptor for this code segment.
;  2) a CPL=0 32/64-bit code descriptor for the 64-bit code.
;  3) a CPL=0 read/write data segment, usable as a stack
;  (referenced by SS).
;
; Load the 32-bit IDT, in case any interrupts or exceptions
; occur after entering protected mode, but before enabling
; long mode).
;
; Initialize the GDTR and IDTR to point to the temporary
; 32-bit GDT and IDT, respectively.
;
   lgdt  ds:[pGDT32]
   lidt  ds:[pIDT32]
;
; Enable protected mode (CR0.PE=1).
;
   mov   eax, 000000011h
   mov   cr0, eax
;
; Execute a far jump to turn protected mode on.
; code16_sel must point to the previously-established 16-bit
; code descriptor located in the GDT (for the code currently
; being executed).
;
   db    0eah              ;Far jump...
   dw    offset now_in_prot;to offset...
   dw    code16_sel        ;in current code segment.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; At this point we are in 16-bit protected mode, but long
; mode is still disabled.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
now_in_prot:
;
; Set up the protected-mode stack pointer, SS:ESP.
; Stack_sel must point to the previously-established stack
; descriptor (read/write data segment), located in the GDT.
; Skip setting DS/ES/FS/GS, because we are jumping right to
; 64-bit code.
;
   mov   ax, stack_sel
   mov   ss, ax
   mov   esp, esp0
```

```
;
; Enable the 64-bit page-translation-table entries by
; setting CR4.PAE=1 (this is _required_ before activating
; long mode). Paging is not enabled until after long mode
; is enabled.
;
   mov   eax, cr4
   bts   eax, 5
   mov   cr4, eax
;
; Create the long-mode page tables, and initialize the
; 64-bit CR3 (page-table base address) to point to the base
; of the PML4 page table. The PML4 page table must be located
; below 4 Gbytes because only 32 bits of CR3 are loaded when
; the processor is not in 64-bit mode.
;
   mov   eax, pml4_base ; Pointer to PML4 table (<4GB).
   mov   cr3, eax       ; Initialize CR3 with PML4 base.
;
; Enable long mode (set EFER.LME=1).
;
   mov   ecx, 0c0000080h   ; EFER MSR number.
   rdmsr                   ; Read EFER.
   bts   eax, 8            ; Set LME=1.
   wrmsr                   ; Write EFER.
;
; Enable paging to activate long mode (set CR0.PG=1)
;
   mov   eax, cr0          ; Read CR0.
   bts   eax, 31           ; Set PE=1.
   mov   cr0, eax          ; Write CR0.
;
; At this point, we are in 16-bit compatibility mode
; ( LMA=1, CS.L=0, CS.D=0 ).
; Now, jump to the 64-bit code segment. The offset must be
; equal to the linear address of the 64-bit entry point,
; because 64-bit code is in an unsegmented address space.
; The selector points to the 32/64-bit code selector in the
; current GDT.
;
   db    066h
   db    0eah
   dd    start64_linear
   dw    code64_sel
code16ends     ; End of the 16-bit code segment
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;;    Start of 64-bit code
;;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
code64  para use64
start64:       ; At this point, we're using 64-bit code
;
; Point the 64-bit RSP register to the stack's _linear_
; address. There is no need to set SS here, because the SS
; register is not used in 64-bit mode.
;
   mov   rsp, stack0_linear
;
; This LGDT is only needed if the long-mode GDT is to be
; located at a linear address above 4 Gbytes.  If the long
; mode GDT is located at a 32-bit linear address, putting
; 64-bit descriptors in the GDT pointed to by [pGDT32] is
; just fine. pGDT64_linear is the _linear_ address of the
; 10-byte GDT pseudo-descriptor.
;
; The new GDT should have a valid CPL0 64-bit code segment
; descriptor at the entry-point corresponding to the current
; CS selector. Alternatively, a far transfer to a valid CPL0
; 64-bit code segment descriptor in the new GDT must be done
; before enabling interrupts.
;
   lgdt  [pGDT64_linear]
;
; Load the 64-bit IDT. This is _required_, because the 64-bit
; IDT uses 64-bit interrupt descriptors, while the 32-bit
; IDT used 32-bit interrupt descriptors. pIDT64_linear is
; the _linear_ address of the 10-byte IDT pseudo-descriptor.
;
   lidt  [pIDT64_linear]
;
; Set the current TSS. tss_sel should point to a 64-bit TSS
; descriptor in the current GDT. The TSS is used for
; inner-level stack pointers and the IO bit-map.
;
   mov   ax, tss_sel
   ltr   ax
;
; Set the current LDT. ldt_sel should point to a 64-bit LDT
; descriptor in the current GDT.
;
   mov   ax, ldt_sel
   lldt  ax
;
; Using fs: and gs: prefixes on memory accesses still uses
; the 32-bit fs.base and gs.base.  Reload these 2 registers
; before using the fs: and gs: prefixes. FS and GS can be
; loaded from the GDT using a normal "mov fs,foo" type
; instructions, which loads a 32-bit base into FS or GS.
; Alternatively, use WRMSR to assign 64-bit base values to
; MSR_FS_base or MSR_GS_base.
```

```
;
   mov   ecx, MSR_FS_base
   mov   eax, FsbaseLow
   mov   edx, FsbaseHi
   wrmsr
;
; Reload CR3 if long-mode page tables are to be located above
; 4 Gbytes. Because the original CR3 load was done in 32-bit
; legacy mode, it could only load 32 bits into CR3. Thus, the
; current page tables are located in the lower 4 Gbytes of
; physical memory. This MOV to CR3 is only needed if the
; actual long-mode page tables should be located at a linear
; address above 4 Gbytes.
;
   mov   rax, final_pml4_base  ; Point to PML4
   mov   cr3, rax              ; Load 64-bit CR3
;
; Enable interrupts.
;
   sti                         ; Enabled INTR
   <insert 64-bit code here>
```

15 Secure Virtual Machine

The AMD Virtualization™ (AMD-V™) architecture is designed to support enterprise-class server virtualization software technology and facilitate virtualization development and deployment on any type of system, through the Secure Virtual Machine (SVM) extension. An SVM-enabled virtual machine architecture provides hardware resources that allow a single physical machine to run multiple operating systems efficiently, while maintaining secure, hardware-enforced isolation.

15.1 The Virtual Machine Monitor

A *virtual machine monitor* (VMM), also known as a *hypervisor*, consists of software that controls the execution of multiple *guest* operating systems on a single physical machine. The VMM provides each guest the appearance of full control over a complete computer system (memory, CPU, and all peripheral devices). The use of the term *host* refers to the execution context of the VMM. *World switch* refers to the operation of switching between the host and guest. A guest may have one or more virtual CPUs (vCPUs) managed by the guest OS, just as on a non-virtualized system, and a VMM may run any mix of vCPUs from the same or different guests on different logical processors simultaneously with no hardware-imposed constraints.

Fundamentally, VMMs work by *intercepting* and emulating in a safe manner sensitive operations in the guest (such as changing the page tables, which could give a guest access to memory it is not allowed to access, or accessing peripheral devices that are shared among multiple guests). The AMD SVM architecture provides hardware assists to improve performance and facilitate implementation of virtualization.

15.2 SVM Hardware Overview

SVM processor support provides a set of hardware extensions designed to enable economical and efficient implementation of virtual machine systems. Generally speaking, hardware support falls into two complementary categories: *virtualization* support and *security* support.

15.2.1 Virtualization Support

The AMD virtual machine architecture is designed to provide:

- A guest/host tagged TLB to reduce virtualization overhead
- External (DMA) access protection for memory
- Assists for interrupt handling, virtual interrupt support, and enhanced pause filter
- The ability to intercept selected instructions or events in the guest
- Mechanisms for fast world switch between VMM and guest

15.2.2 Guest Mode

This new processor mode is entered through the VMRUN instruction. When in guest mode, the behavior of some x86 instructions changes to facilitate virtualization.

The CPUID function numbers 4000_0000h–4000_00FFh have been reserved for software use. Hypervisors can use these function numbers to provide an interface to pass information from the hypervisor to the guest. This is similar to extracting information about a physical CPU by using CPUID. Hypervisors use the CPUID Fn 400000[FF:00] bit to denote a virtual platform.

Feature bit CPUID Fn0000_0001_ECX[31] has been reserved for use by hypervisors to indicate the presence of a hypervisor. Hypervisors set this bit to 1 and physical CPU's set this bit to zero. This bit can be probed by the guest software to detect whether they are running inside a virtual machine.

15.2.3 External Access Protection

Guests may be granted direct access to selected I/O devices. Hardware support is designed to prevent devices owned by one guest from accessing memory owned by another guest (or the VMM).

15.2.4 Interrupt Support

To facilitate efficient virtualization of interrupts, the following support is provided under control of VMCB flags:

Intercepting physical interrupt delivery. The VMM can request that physical interrupts cause a running guest to exit, allowing the VMM to process the interrupt.

Virtual interrupts. The VMM can inject virtual interrupts into the guest. Under control of the VMM, a virtual copy of the EFLAGS.IF interrupt mask bit, and a virtual copy of the APIC's task priority register are used transparently by the guest instead of the physical resources.

Sharing a physical APIC. SVM allows multiple guests to share a physical APIC with isolation of each guest's manipulation of APIC state from the other guests' views of their own APIC state, so that no guest can interfere with delivery of interrupts to another guest.

Direct interrupt delivery. On models that support it, the Advanced Virtual Interrupt Controller (AVIC) extension virtualizes the APIC's interrupt delivery functions. This provides for delivery of device or inter-processor interrupts directly to a target vCPU or vCPUs, which avoids the overhead of having the VMM to determine interrupt routing and speeds up interrupt delivery. (see section 15.29).

15.2.5 Restartable Instructions

SVM is designed to safely restart, with the exception of task switches, any intercepted instruction (either atomic or idempotent) after the intercept.

15.2.6 Security Support

To further support secure initialization and execution, SVM provides additional system support through a variety of extensions.

Attestation. The SKINIT instruction and associated system support (the Trusted Platform Module, or TPM) allow for verifiable startup of trusted software (such as a hypervisor, or a native operating system), based on secure hash comparison. (section 15.27).

Encrypted memory. On models that support it, the Secure Encrypted Virtualization (SEV) and SEV Encrypted State (SEV-ES) extensions guard against inspection of guest memory and (for SEV-ES) guest register state by malicious hypervisor code, memory bus tracing or memory device removal through encryption of guest memory and register contents (section 15.34 and section 15.35).

Secure Nested Paging. On models that support it, the SEV-SNP extension provides additional protection for guest memory against malicious manipulation of address translation mechanisms by hypervisor code. (section 15.36).

15.3 SVM Processor and Platform Extensions

SVM hardware extensions can be grouped into the following categories:

- State switch—VMRUN, VMSAVE, VMLOAD instructions, global interrupt flag (GIF), and instructions to manipulate the latter (STGI, CLGI). (section 15.5, section 15.5.2, section 15.17)
- Intercepts—allow the VMM to intercept sensitive operations in the guest. (section 15.7 through section 15.14)
- Interrupt and APIC assists—physical interrupt intercepts, virtual interrupt support, APIC.TPR virtualization. (section 15.17 and section 15.21)
- SMM intercepts and assists (section 15.22)
- External (DMA) access protection (section 15.24)
- Nested paging support for two levels of address translation. (section 15.25)
- Security—SKINIT instruction. (section 15.27)

15.4 Enabling SVM

The VMRUN, VMLOAD, VMSAVE, CLGI, VMMCALL, and INVLPGA instructions can be used when the EFER.SVME is set to 1; otherwise, these instructions generate a #UD exception. The SKINIT and STGI instructions can be used when either the EFER.SVME bit is set to 1 or the feature flag CPUID Fn8000_0001_ECX[SKINIT] is set to 1; otherwise, these instructions generate a #UD exception.

Before enabling SVM, software should detect whether SVM can be enabled using the following algorithm:

```
if (CPUID Fn8000_0001_ECX[SVM] == 0)
  return SVM_NOT_AVAIL;

if (VM_CR.SVMDIS == 0)
  return SVM_ALLOWED;

if (CPUID Fn8000_000A_EDX[SVML]==0)
  return SVM_DISABLED_AT_BIOS_NOT_UNLOCKABLE
  // the user must change a platform firmware setting to enable SVM
else return SVM_DISABLED_WITH_KEY;
  // SVMLock may be unlockable; consult platform firmware or TPM to obtain the
key.
```

For more information on using the CPUID instruction to obtain processor capability information, see Section 3.3, “Processor Feature Identification,” on page 71.

15.5 VMRUN Instruction

The VMRUN instruction is the cornerstone of SVM. VMRUN takes, as a single argument, the physical address of a 4KB-aligned page, the *virtual machine control block* (VMCB), which describes a virtual machine (guest) to be executed. The VMCB contains:

- a list of instructions or events in the guest (e.g., write to CR3) to intercept,
- various control bits that specify the execution environment of the guest or that indicate special actions to be taken before running guest code, and
- guest processor state (such as control registers, etc.).

Note that VMRUN is not supported inside the SMM handler and the behavior is undefined.

15.5.1 Basic Operation

The VMRUN instruction has an implicit addressing mode of [rAX]. Software must load RAX (EAX in 32-bit mode) with the physical address of the VMCB, a 4-Kbyte-aligned page that describes a virtual machine to be executed. The portion of RAX used in forming the address is determined by the current effective address size.

The VMCB is accessed by physical address and should be mapped as writeback (WB) memory.

VMRUN is available only at CPL 0. A #GP(0) exception is raised if the CPL is greater than 0. Furthermore, the processor must be in protected mode and EFER.SVME must be set to 1, otherwise, a #UD exception is raised.

The VMRUN instruction saves some host processor state information in the host state-save area in main memory at the physical address specified in the VM_HSAVE_PA MSR; it then loads corresponding guest state from the VMCB state-save area. VMRUN also reads additional control bits from the VMCB that allow the VMM to flush the guest TLB, inject virtual interrupts into the guest, etc.

The VMRUN instruction then checks the guest state just loaded. If an illegal state has been loaded, the processor exits back to the host (section 15.6).

Otherwise, the processor now runs the guest code until an intercept event occurs, at which point the processor suspends guest execution and resumes host execution at the instruction following the VMRUN. This is called a #VMEXIT and is described in detail in (section 15.6).

VMRUN saves or restores a minimal amount of state information to allow the VMM to resume execution after a guest has exited. This allows the VMM to handle simple intercept conditions quickly. If additional guest state information must be saved or restored (e.g., to handle more complex intercepts or to switch to a different guest), the VMM must use the VMLOAD and VMSAVE instructions to handle the additional guest state (section 15.5.2).

Saving Host State. To ensure that the host can resume operation after #VMEXIT, VMRUN saves at least the following host state information:

- CS.SEL, NEXT_RIP—The CS selector and rIP of the instruction following the VMRUN. On #VMEXIT the host resumes running at this address.
- RFLAGS, RAX—Host processor mode and the register used by VMRUN to address the VMCB.
- SS.SEL, RSP—Stack pointer for host.
- CR0, CR3, CR4, EFER—Paging/operating mode for host.
- IDTR, GDTR—The pseudo-descriptors. VMRUN does not save or restore the host LDTR.
- ES.SEL and DS.SEL.

Processor implementations may store only part or none of host state in the memory area pointed to by VM_HSAVE_PA MSR and may store some or all host state in hidden on-chip memory. Different implementations may choose to save the hidden parts of the host's segment registers as well as the selectors. For these reasons, software must not rely on the format or contents of the host state save area, nor attempt to change host state by modifying the contents of the host save area.

Loading Guest State. After saving host state, VMRUN loads the following guest state from the VMCB:

- CS, rIP—Guest begins execution at this address. The hidden state of the CS segment register is also loaded from the VMCB.
- RFLAGS, RAX.
- SS, RSP—Includes the hidden state of the SS segment register.
- CR0, CR2, CR3, CR4, EFER—Guest paging mode. Writing paging-related control registers with VMRUN does *not* flush the TLB since address spaces are switched. (section 15.16.)
- INTERRUPT_SHADOW—This flag indicates whether the guest is currently in an interrupt lockout shadow; (section 15.21.5).
- IDTR, GDTR.
- ES and DS—Includes the hidden state of the segment registers.

- DR6 and DR7—The guest's breakpoint state.
- V_TPR—The guest's virtual TPR.
- V_IRQ—The flag indicating whether a virtual interrupt is pending in the guest.
- CPL—If the guest is in real mode, the CPL is forced to 0; if the guest is in v86 mode, the CPL is forced to 3. Otherwise, the CPL saved in the VMCB is used.

The processor checks the loaded guest state for consistency. If a consistency check fails while loading guest state, the processor performs a #VMEXIT. For additional information, see "Canonicalization and Consistency Checks" on page 489.

If the guest is in PAE paging mode according to the registers just loaded and nested paging is not enabled, the processor will also read the four PDPEs pointed to by the newly loaded CR3 value; setting any reserved bits in the PDPEs also causes a #VMEXIT.

It is possible for the VMRUN instruction to load a guest rIP that is outside the limit of the guest code segment or that is non-canonical (if running in long mode). If this occurs, a #GP fault is delivered inside the guest; the rIP falling outside the limit of the guest code segment is not considered illegal guest state.

After all guest state is loaded, and intercepts and other control bits are set up, the processor reenables interrupts by setting GIF to 1. It is assumed that VMM software cleared GIF some time before executing the VMRUN instruction, to ensure an atomic state switch.

Some processor models allow the VMM to designate certain guest VMCB fields as "clean," meaning that they haven't been modified relative to the current state of hardware. This allows the hardware to optimize execution of VMRUN. See section 15.15 for details on which fields may be affected by this. The descriptions below assume all fields are loaded.

Control Bits. Besides loading guest state, the VMRUN instruction reads various control fields from the VMCB; most of these fields are not written back to the VMCB on #VMEXIT, since they cannot change during guest execution:

- TSC_OFFSET—an offset to add when the guest reads the TSC (time stamp counter). Guest writes to the TSC can be intercepted and emulated by changing the offset (without writing the physical TSC). This offset is cleared when the guest exits back to the host.
- V_INTR_PRIO, V_INTR_VECTOR, V_IGN_TPR—fields used to describe a virtual interrupt for the guest (see "Injecting Virtual (INTR) Interrupts" on page 518).
- V_INTR_MASKING—controls whether masking of interrupts (in EFLAGS.IF and TPR) is to be virtualized (section 15.21).
- The address space ID (ASID) to use while running the guest.
- A field to control flushing of the TLB during a VMRUN (see Section 15.16).
- The intercept vectors describing the active intercepts for the guest. On exit from the guest, the internal intercept registers are cleared so no host operations will be intercepted.

The maximum ASID value supported by a processor is implementation specific. The value returned in EBX after executing CPUID Fn8000_000A is the number of ASIDs supported by the processor.

See Section 3.3, “Processor Feature Identification,” on page 71 for more information on using the CPUID instruction.

Segment State in the VMCB. The segment registers are stored in the VMCB in a format similar to that for SMM: both base and limit are fully expanded; segment attributes are stored as 12-bit values formed by the concatenation of bits 55:52 and 47:40 from the original 64-bit (in-memory) segment descriptors; the descriptor “P” bit is used to signal NULL segments (P=0) where permissible and/or relevant. The loading of segment attributes from the VMCB (which may have been overwritten by software) may result in attribute bit values that are otherwise not allowed. However, only some of the attribute bits are actually observed by hardware, depending on the segment register in question:

- CS—D, L, P, and R.
- SS—B, P, E, W, and Code/Data
- DS, ES, FS, GS —D, P, DPL, E, W, and Code/Data.
- LDTR—P, S, and Type (LDT)
- TR—P, S, and Type (32- or 16-bit TSS)

NOTE: For the Stack Segment attributes, P is observed in legacy and compatibility mode. In 64-bit mode, P is ignored because all stack segments are treated as present.

The VMM should follow these rules when storing segment attributes into the VMCB:

- For NULL segments, set all attribute bits to zero; otherwise, write the concatenation of bits 55:52 and 47:40 from the original 64-bit (in-memory) segment descriptors.
- The processor reads the current privilege level from the CPL field in the VMCB. The CS.DPL will match the CPL field.
- When in virtual x86 or real mode, the processor ignores the CPL field in the VMCB and forces the values of 3 and 0, respectively.

When examining segment attributes after a #VMEXIT:

- Test the Present (P) bit to check whether a segment is NULL; note that CS and TR never contain NULL segments and so their P bit is ignored;
- Retrieve the CPL from the CPL field in the VMCB, not from any segment DPL.

Canonicalization and Consistency Checks. The VMRUN instruction performs consistency checks on guest state and #VMEXIT performs the appropriate subset of these consistency checks on host state. Illegal guest state combinations cause a #VMEXIT with error code VMEXIT_INVALID. The following conditions are considered illegal state combinations (note that some checks may be subject to VMCB Clean field settings, see below):

- EFER.SVME is zero.
- CR0.CD is zero and CR0.NW is set.

- CR0[63:32] are not zero.
- Any MBZ bit of CR3 is set.
- Any MBZ bit of CR4 is set.
- DR6[63:32] are not zero.
- DR7[63:32] are not zero.
- Any MBZ bit of EFER is set.
- EFER.LMA or EFER.LME is non-zero and this processor does not support long mode.
- EFER.LME and CR0.PG are both set and CR4.PAE is zero.
- EFER.LME and CR0.PG are both non-zero and CR0.PE is zero.
- EFER.LME, CR0.PG, CR4.PAE, CS.L, and CS.D are all non-zero.
- The VMRUN intercept bit is clear.
- The MSR or IOIO intercept tables extend to a physical address that is greater than or equal to the maximum supported physical address.
- Illegal event injection (section 15.20).
- ASID is equal to zero.
- Any reserved bit is set in S_CET
- CR4.CET=1 when CR0.WP=0
- CR4.CET=1 and U_CET.SS=1 when EFLAGS.VM=1
- any reserved bit set in U_CET (SEV_ES only):

  - VMRUN results in VMEXIT(INVALID)

  - VMEXIT forces reserved bits to 0

VMRUN can load a guest value of CR0 with PE = 0 but PG = 1, a combination that is otherwise illegal (see Section 15.19).

In addition to consistency checks, VMRUN and #VMEXIT canonicalize (i.e., sign-extend to bit 63):

- All base addresses in the segment registers that have been loaded.
- SSP
- ISST_ADDR
- PL0_SSP, PL1_SSP, PL2_SSP, PL3_SSP

VMCB Clean field behavior: On processor models that support designation of clean fields, the final merged hardware state is used for consistency checks. This may include state from fields marked as clean, if the processor chooses to ignore the indication.

VMRUN and TF/RF Bits in EFLAGS. When considering interactions of VMRUN with the TF and RF bits in EFLAGS, one must distinguish between the behavior of host as opposed to that of the guest.

From the host point of view, VMRUN acts like a single instruction, even though an arbitrary number of guest instructions may execute before a #VMEXIT effectively completes the VMRUN. As a single host instruction, VMRUN interacts with EFLAGS.RF and EFLAGS.TF like ordinary instructions. EFLAGS.RF suppresses any potential instruction breakpoint match on the VMRUN, and EFLAGS.TF causes a #DB trap after the VMRUN completes on the host side (i.e., after the #VMEXIT from the guest). As with any normal instruction, completion of the VMRUN instruction clears the host EFLAGS.RF bit.

The value of EFLAGS.RF from the VMCB affects the first guest instruction. When VMRUN loads a guest value of 1 for EFLAGS.RF, that value takes effect and suppresses any potential (guest) instruction breakpoint on the first guest instruction. When VMRUN loads a guest value of 1 in EFLAGS.TF, that value does *not* cause a trace trap between the VMRUN and the first guest instruction, but rather *after* completion of the first guest instruction.

Host values of EFLAGS have no effect on the guest and guest values of EFLAGS have no effect on the host.

See also section 15.7.1 regarding the value of EFLAGS.RF saved on #VMEXIT.

15.5.2 VMSAVE and VMLOAD Instructions

These instructions transfer additional guest register context, including hidden context that is not otherwise accessible, between the processor and a guest's VMCB for a more complete context switch than VMRUN and #VMEXIT perform. The system physical address of the VMCB is specified in rAX. When these operations are needed, VMLOAD would be executed as desired prior to executing a VMRUN, and VMSAVE at any desired point after a #VMEXIT.

The VMSAVE and VMLOAD instructions take the physical address of a VMCB in rAX. These instructions complement the state save/restore abilities of VMRUN instruction and #VMEXIT. They provide access to hidden processor state that software cannot otherwise access, as well as additional privileged state.

These instructions handle the following register state:

- FS, GS, TR, LDTR (including all hidden state)
- KernelGsBase
- STAR, LSTAR, CSTAR, SFMASK
- SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP

Like VMRUN, these instructions are only available at CPL0 (otherwise causing a #GP(0) exception) and are only valid in protected mode with SVM enabled via EFER.SVME (otherwise causing a #UD exception).

15.6 #VMEXIT

When an intercept triggers, the processor performs a #VMEXIT (i.e., an exit from the guest to the host context).

On #VMEXIT, the processor:

- Disables interrupts by clearing the GIF, so that after the #VMEXIT, VMM software can complete the state switch atomically.
- Writes back to the VMCB the current guest state—the same subset of processor state as is loaded by the VMRUN instruction, including the V_IRQ, V_TPR, and the INTERRUPT_SHADOW bits.
- Saves the reason for exiting the guest in the VMCB's EXITCODE field; additional information may be saved in the EXITINFO1 or EXITINFO2 fields, depending on the intercept. Note that the contents of the EXITINFO1 and EXITINFO2 fields are undefined for intercepts where their use is not indicated.
- Clears all intercepts.
- Resets the current ASID register to zero (host ASID).
- Clears the V_IRQ and V_INTR_MASKING bits inside the processor.
- Clears the TSC_OFFSET inside the processor.
- Reloads the host state previously saved by the VMRUN instruction. The processor reloads the host's CS, SS, DS, and ES segment registers and, if required, re-reads the descriptors from the host's segment descriptor tables, depending on the implementation. The segment descriptor tables must be mapped as present and writable by the host's page tables. Software should keep the host's segment descriptor tables consistent with the segment registers when executing VMRUN instructions. Immediately after #VMEXIT, the processor still contains the guest value for LDTR. So for CS, SS, DS, and ES, the VMM must only use segment descriptors from the global descriptor table. (The VMSAVE instruction can be used for a more complete context switch, allowing the VMM to then load LDTR and other registers not saved by #VMEXIT with desired values; see section 15.5.2 for details.) Any exception encountered while reloading the host segments causes a shutdown.
- If the host is in PAE mode, the processor reloads the host's PDPEs from the page table indicated by the host's CR3. If the PDPEs contain illegal state, the processor causes a shutdown.
- Forces CR0.PE = 1, RFLAGS.VM = 0.
- Sets the host CPL to zero.
- Disables all breakpoints in the host DR7 register.
- Checks the reloaded host state for consistency; any error causes the processor to shutdown. If the host's rIP reloaded by #VMEXIT is outside the limit of the host's code segment or non-canonical (in the case of long mode), a #GP fault is delivered inside the host.

15.7 Intercept Operation

Various instructions and events (such as exceptions) in the guest can be intercepted by means of control bits in the VMCB “Layout of VMCB” on page 679. The two primary classes of intercepts supported by SVM are instruction and exception intercepts.

Exception intercepts. Exception intercepts are checked when normal instruction processing must raise an exception before resolving possible double-fault conditions and before attempting delivery of the exception (which includes pushing an exception frame, accessing the IDT, etc.).

For some exceptions, the processor still writes certain exception-specific registers even if the exception is intercepted. (See the descriptions in section 15.12 and following for details.) When an external or virtual interrupt is intercepted, the interrupt is left pending.

When an intercept occurs while the guest is in the process of delivering a non-intercepted interrupt or exception using the IDT, SVM provides additional information on #VMEXIT (See section 15.7.2).

Instruction intercepts. These occur at well-defined points in instruction execution—before the results of the instruction are committed, but ordered in an intercept-specific priority relative to the instruction’s exception checks. Generally, instruction intercepts are checked after simple exceptions (such as #GP—when CPL is incorrect—or #UD) have been checked, but before exceptions related to memory accesses (such as page faults) and exceptions based on specific operand values. There are several exceptions to this guideline, e.g., the RSM instruction. Instruction breakpoints for the current instruction and pending data breakpoint traps from the previous instruction are designed to be checked before instruction intercepts.

15.7.1 State Saved on Exit

When triggered, intercepts write an EXITCODE into the VMCB identifying the cause of the intercept. The EXITINTINFO field signals whether the intercept occurred while the guest was attempting to deliver an interrupt or exception through the IDT; a VMM can use this information to transparently complete the delivery (section 15.20). Some intercepts provide additional information in the EXITINFO1 and EXITINFO2 fields in the VMCB; see the individual intercept descriptions for details.

The guest state saved in the VMCB is the processor state as of the moment the intercept triggers. In the x86 architecture, traps (as opposed to faults) are detected and delivered after the instruction that triggered them has completed execution. Accordingly, a trap intercept takes place after the execution of the instruction that triggered the trap in the first place. The saved guest state thus includes the effects of executing that instruction.

Example: Assume a guest instruction triggers a data breakpoint (#DB) trap which is in turn intercepted. The VMCB records the guest state after execution of that instruction, so that the saved CS:rIP points to the following instruction, and the saved DR7 includes the effects of matching the data breakpoint.

The next sequential instruction pointer (nRIP) is saved in the guest VMCB control area at location C8h on all #VMEXITs that are due to instruction intercepts, as defined in section 15.9, as well as MSR and IOIO intercepts and exceptions caused by the INT3, INTO, and BOUND instructions. For all other intercepts, nRIP is reset to zero.

The nRIP is the RIP that would be pushed on the stack if the current instruction were subject to a trap-style debug exception, if the intercepted instruction were to cause no change in control flow. If the intercepted instruction would have caused a change in control flow, the nRIP points to the next sequential instruction rather than the target instruction.

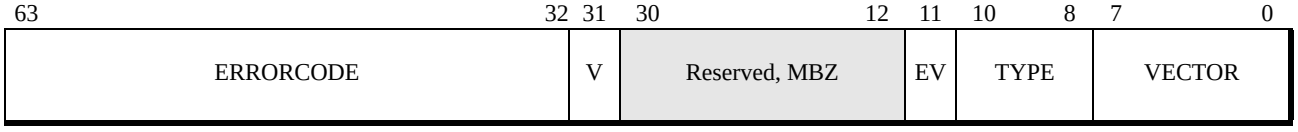
Some exceptions write special registers even when they are intercepted; see the individual descriptions in section 15.12 for details.

Support for the NRIP save on #VMEXIT is indicated by CPUID Fn8000_000A_EDX[NRIPS]. See Section 3.3, “Processor Feature Identification,” on page 71 for more information on using the CPUID instruction.

15.7.2 Intercepts During IDT Interrupt Delivery

It is possible for an intercept to occur while the guest is attempting to deliver an exception or interrupt through the IDT (e.g., #PF because the VMM has paged out the guest’s exception stack). In some cases, such an intercept can result in the loss of information necessary for transparent resumption of the guest. In the case of an external interrupt, for example, the processor will already have performed an interrupt acknowledge cycle with the PIC or APIC to obtain the interrupt type and vector, and the interrupt is thus no longer pending.

To recover from such situations, all intercepts indicate (in the EXITINTINFO field in the VMCB) whether they occurred during exception or interrupt delivery though the IDT. This mechanism allows the VMM to complete the intercepted interrupt delivery, even when it is no longer possible to recreate the event in question.



Bits	Mnemonic	Description
63:32	ERRORCODE	Error Code
31	V	Valid
30:12	—	Reserved, MBZ
11	EV	Error Code Valid
10:8	TYPE	Qualifies the guest exception or interrupt. Table 15-1 shows possible values returned and their corresponding interrupt or exception types. Values not indicated are unused and reserved.
7:0	VECTOR	8-bit IDT vector of the interrupt or exception.

Figure 15-1. EXITINTINFO for All Intercepts

Table 15-1. Guest Exception or Interrupt Types

Value	Type
0	External or virtual interrupt (INTR)
2	NMI
3	Exception (fault or trap)
4	Software interrupt (caused by INT*n* instruction)

Despite the instruction name, the events raised by the INT1 (also known as ICEBP), INT3 and INTO instructions (opcodes F1h, CCh and CEh) are considered exceptions for the purposes of EXITINTINFO, not software interrupts. Only events raised by the INT*n* instruction (opcode CDh) are considered software interrupts.

- Error Code Valid—Bit 11. Set to 1 if the guest exception would have pushed an error code; otherwise cleared to zero.
- Valid—Bit 31. Set to 1 if the intercept occurred while the guest attempted to deliver an exception through the IDT; otherwise cleared to zero.
- Errorcode—Bits 63:32. If EV is set to 1, holds the error code that the guest exception would have pushed; otherwise is undefined.

In the case of multiple exceptions, EXITINTINFO records the aggregate information on all exceptions but the last (intercepted) one.

Example: A guest raises a #GP during delivery of which a #NP is raised (a scenario that, according to x86 rules, resolves to a #DF), and an intercepted #PF occurs during the attempt to deliver the #DF. Upon intercept of the #PF, EXITINTINFO indicates that the guest was in the process of delivering a #DF when the #PF occurred. The information about the intercepted page fault itself is encoded in the EXITCODE, EXITINFO1 and EXITINFO2 fields. If the VMM decides to repair and dismiss the #PF, it can resume guest execution by re-injecting (see section 15.20) the fault recorded in EXITINTINFO. If the VMM decides that the #PF should be reflected back to the guest, it must combine the event in EXITINTINFO with the intercepted exception according to x86 rules. In this case, a #DF plus a #PF would result in a triple fault or shutdown.

15.7.3 EXITINTINFO Pseudo-Code

When delivering exceptions or interrupts in a guest, the processor checks for exception intercepts and updates the value of EXITINTINFO should an intercept occur during exception delivery. The following pseudo-code outlines how the processor delivers an event (exception or interrupt) E.

```
if E is an exception and is intercepted:
    #VMEXIT(E)
E = (result of combining E with any prior events)

if (result was #DF and #DF is intercepted):
    #VMEXIT(#DF)
if (result was shutdown and shutdown is intercepted):
```

```
    #VMEXIT(#shutdown)
EXITINTINFO = E // Record the event the guest is delivering.

Attempt delivery of E through the IDT
Note that this may cause secondary exceptions

Once an exception has been successfully taken in the guest:

EXITINTINFO.V = 0 // Delivery succeeded; no #VMEXIT.
Dispatch to first instruction of handler
```

When an exception triggers an intercept, the EXITCODE, and optionally EXITINFO1 and EXITINFO2, fields always reflect the intercepted exception, while EXITINTINFO, if marked valid, indicates the prior exception the guest was attempting to deliver when the intercept occurred.

15.8 Decode Assists

Decode assists are provided to allow hypervisors to decode guest instructions more efficiently. CPUID Fn8000_000A_EDX[DecodeAssists] = 1 indicates support for this feature. See Section 3.3, "Processor Feature Identification," on page 71 for more information on using the CPUID instruction.

15.8.1 MOV CR*x*/DR*x* Intercepts

The EXITINFO1 field holds a flag indicating whether the instruction was a MOV CRx and the number of the GPR operand. MOV-to-CR instructions always set bit 63 and provide the GPR number, except for CR0 as specified below.

Table 15-2. EXITINFO1 for MOV CR*x*

Bit Offsets	Field Contents
3:0	GPR number
62:4	0
63	Instruction was MOV CR*x*—set to1 if the instruction was a MOV CR*x* instruction; cleared to 0 otherwise.

Table 15-3. EXITINFO1 for MOV DR*x*

Bit Offsets	Field Contents
3:0	GPR number
63:4	0

MOV-to-CR0 Special Case. If the instruction is MOV-to-CR, the GPR number is provided; if the instruction is LMSW or CLTS, no additional information is provided and bit 63 is not set.

MOV-from-CR0 Special Case. If the instruction is MOV-from-CR, the GPR number is provided and bit 63 is set; if the instruction is SMSW, no information is provided and bit 63 is not set.

15.8.2 INT*n* Intercepts

EXITINFO1 records the immediate value of the interrupt number for INT *n* instructions. See Table 15-4.

Table 15-4. EXITINFO1 for INT*n*

Bit Offsets	Field Contents
7:0	Software interrupt number
63:8	0

15.8.3 INVLPG and INVLPGA Intercepts

For an INVLPG intercept, EXITINFO1 provides the linear address after segment base addition and address size masking produce the effective address size. See Table 15-5. For an INVLPGA intercept, the linear address is available directly from the guest rAX register and is not provided in EXITINFO1.

Table 15-5. EXITINFO1 for INVLPG

Bit Offsets	Field Contents
63:0	Linear address

15.8.4 Nested and intercepted #PF

In the case of a Nested Page Fault or intercepted #PF, guest instruction bytes at guest CS:RIP are stored into the 16-byte wide field Guest Instruction Bytes located at offset 0D0h in the VMCB. The format of this field is summarized in Table 15-6 below. Up to 15 bytes are recorded, read from guest CS:RIP. If a faulting condition occurs, such as not-present page or exceeding the CS limit, then the Guest Instruction Bytes field records as many bytes as could be fetched. The number of bytes fetched is put into the first byte of this field. Zero indicates that no bytes were fetched. The default number of bytes is always 15. Fewer bytes are returned only if a fault occurs while fetching.

This field is filled in only during data page faults. Instruction-fetch page faults provide no additional information.

All other intercepts clear bits 0:7 in this field to zero (to indicate an invalid condition); implementations may leave the other bytes untouched.

Table 15-6. Guest Instruction Bytes

Bit Offsets	Field Contents
3:0	Number of bytes fetched
4:7	0
127:8	Instruction bytes

15.9 Instruction Intercepts

Table 15-7 specifies the instructions that check a given intercept and, where relevant, how the intercept is prioritized relative to exceptions.

Table 15-7. Instruction Intercepts

Instruction Intercept	Checked By	Priority
Read/Write of CR0	MOV TO/FROM CR0, LMSW, SMSW, CLTS	Checks non-memory exceptions (CPL, illegal bit combinations, etc.) before the intercept. For LMSW and SMSW, checks SVM intercepts before checking memory exceptions.
Read/Write of CR3 (excluding task switch)	MOV TO/FROM CR3 (not checked by task switch operations)	Checks non-memory exceptions first, then the intercept. If the intercept triggers on a write, the intercept happens *before* the TLB is flushed. If PAE is enabled, the loading of the four PDPEs can cause a #GP; that exception is checked *after* the intercept check, so the VMM handling a CR3 intercept cannot rely on the PDPEs being legal; it must examine them in software if necessary. The reads and writes of CR3 that occur in VMRUN, #VMEXIT or task switches are *not* subject to this intercept check.
Read/Write of other CRs	MOV TO/FROM CR*n*	All normal exception checks take precedence over the SVM intercepts.
Read/Write of Debug Registers, DR*n*	MOV TO/FROM DR*n*. (Not checked by implicit DR6/DR7 writes.)	All normal exception checks take precedence over the SVM intercepts.

Table 15-7. Instruction Intercepts (continued)

Instruction Intercept	Checked By	Priority
Selective CR0 Write Intercept	MOV TO CR0, LMSW	Checks non-memory exceptions (CPL, illegal bit combinations, etc.) before the intercept. For LMSW and SMSW, checks SVM intercepts before checking memory exceptions. The selective write intercept on CR0 triggers only if a bit other than CR0.TS or CR0.MP is being changed by the write. In particular, this means that CLTS does not check this intercept. When both selective and non-selective CR0-write intercepts are active at the same time, the non-selective intercept takes priority. With respect to exceptions, the priority of this intercept is the same as the generic CR0-write intercept. The LMSW instruction treats the selective CR0-write intercept as a non-selective intercept (i.e., it intercepts regardless of the value being written).
Reading or Writing IDTR, GDTR, LDTR, TR	LIDT, SIDT, LGDT, SGDT, LLDT, SLDT, LTR, STR	The SVM intercept is checked after #UD and #GP exception checks, but before any memory access is performed.
RDTSC	RDTSC	Checks all exceptions before the SVM intercept.
RDPMC	RDPMC	Checks all exceptions before the SVM intercept.
PUSHF	PUSHF	Takes priority over any exceptions.
POPF	POPF	Takes priority over any exceptions.
CPUID	CPUID	No exceptions to check.
RSM	RSM	The intercept takes priority over any exceptions.
IRET	IRET	The intercept takes priority over any exceptions.
Software Interrupt	INT*n*	The intercept occurs before any exceptions are checked. The CS:rIP reported on #VMEXIT are those of the intercepted INTn instruction. Though the INT*n* instruction may dispatch through IDT vectors in the range of 0–31, those events cannot be intercepted by means of exception intercepts (see “Exception Intercepts” on page 504).
INVD	INVD	Exceptions (#GP) are checked before the intercept.

Table 15-7. Instruction Intercepts (continued)

Instruction Intercept	Checked By	Priority
PAUSE	PAUSE	No exceptions to check. VMRUN copies the VMCB.PauseFilterCount into an internal counter. Each PAUSE instruction decrements the counter, and the PAUSE intercept only occurs if the counter goes below zero while the PAUSE intercept is enabled. The VMCB.PauseFilterCount field is not written by the processor. Certain events, including SMI, can cause the internal count to be reloaded from the VMCB. VMCB.PauseFilterCount support is indicated by EDX[10] as returned by CPUID extended function 8000_000A. If This feature is not supported or VMCB.PauseFilterCount = 0, then the first PAUSE instruction can be intercepted.
HLT	HLT	Checks all exceptions before checking for this intercept.
INVLPG	INVLPG	Checks all exceptions (#GP) before the intercept.
INVLPGA	INVLPGA	Checks all exceptions (#GP) before the intercept.
VMRUN	VMRUN	Checks exceptions (#GP) before the intercept. *The current implementation requires that the VMRUN intercept always be set in the VMCB.*
VMLOAD	VMLOAD	Checks exceptions (#GP) before the intercept.
VMSAVE	VMSAVE	Checks exceptions (#GP) before the intercept.
VMMCALL	VMMCALL	The intercept takes priority over exceptions. VMMCALL causes #UD in the guest if it is not intercepted.
STGI	STGI	Checks exceptions (#GP) before the intercept.
CLGI	CLGI	Checks exceptions (#GP) before the intercept.
SKINIT	SKINIT	Checks exceptions (#GP) before the intercept.
RDTSCP	RDTSCP	Checks all exceptions before the SVM intercept.
ICEBP	ICEBP(opcode F1h).	Although the ICEBP instruction dispatches through IDT vector 1, that event is not interceptable by means of the #DB exception intercept.
WBINVD	WBINVD, WBNOINVD	Checks exceptions (#GP) before the intercept.
MONITOR	MONITOR, MONITORX	Checks all exceptions before the intercept.

Table 15-7. Instruction Intercepts (continued)

Instruction Intercept	Checked By	Priority
MWAIT	MWAIT, MWAITX	Checks all exceptions before the intercept. There are conditional and unconditional MWAIT intercepts. The conditional MWAIT intercept is checked before the unconditional MWAIT intercept. When both conditional and unconditional MWAIT intercepts are active, the conditional intercept is checked first. A hypervisor that sets both intercepts will receive the conditional MWAIT intercept exit code for a guest MWAIT instruction that would have entered a low-power state, and will receive the unconditional MWAIT intercept exit code for a guest MWAIT instruction that would not have entered the low-power state. These checks also apply to MWAITX.
XSETBV	XSETBV	Checks intercept before exceptions (#GP).
RDPRU	RDPRU	Check all exceptions before the intercept.
INVLPGB	INVLPGB	Intercept takes priority over all exceptions except #GP for CPL<>0.
INVLPGB_ILLEGAL	INVLPGB exception cases	Intercept takes priority over all exceptions except #GP for CPL<>0.
INVPCID	INVPCID	Intercept takes priority over all exceptions except #GP for CPL<>0.
TLBSYNC	TLBSYNC	Checks exceptions (#GP) before the intercept.

15.10 IOIO Intercepts

The VMM can intercept IOIO instructions (IN, OUT, INS, OUTS) on a port-by-port basis by means of the SVM I/O permissions map.

15.10.1 I/O Permissions Map

The I/O Permissions Map (IOPM) occupies 12 Kbytes of contiguous physical memory. The map is structured as a linear array of 64K+3 bits (two 4-Kbyte pages, and the first three bits of a third 4-Kbyte page) and must be aligned on a 4-Kbyte boundary; the physical base address of the IOPM is specified in the IOPM_BASE_PA field in the VMCB and loaded into the processor by the VMRUN instruction. The VMRUN instruction ignores the lower 12 bits of the address specified in the VMCB. If the address of the last byte in the IOPM is greater than or equal to the maximum supported physical address, this is treated as illegal VMCB state and causes a #VMEXIT(VMEXIT_INVALID).

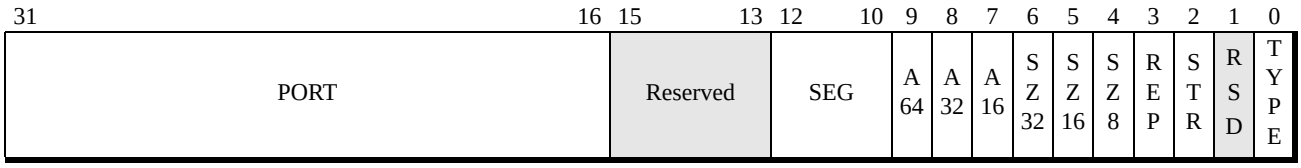
Each bit in the IOPM corresponds to an 8-bit I/O port. Bit 0 in the table corresponds to I/O port 0, bit 1 to I/O port 1 and so on. A bit set to 1 indicates that accesses to the corresponding port should be intercepted. The IOPM is accessed by physical address, and should reside in memory that is mapped as writeback (WB).

15.10.2 IN and OUT Behavior

If the IOIO_PROT intercept bit is set, the IOPM controls port access. For IN/OUT instructions that access more than a single byte, the permission bits for all bytes are checked; if any bit is set to 1, the I/O operation is intercepted.

Exceptions related to virtual x86 mode, IOPL, or the TSS-bitmap are checked *before* the SVM intercept check. All other exceptions are checked *after* the SVM intercept check.

I/O Intercept Information. When an IOIO intercept triggers, the following information (describing the intercepted operation in order to facilitate emulation) is saved in the VMCB's EXITINFO1 field:



Bits	Mnemonic	Description
31:16	PORT	Intercepted I/O port
15-13	—	Reserved
12:10	SEG	Effective segment number
9	A64	64-bit address
8	A32	32-bit address
7	A16	16-bit address
6	SZ32	32-bit operand size
5	SZ16	16-bit operand size
4	SZ8	8-bit operand size
3	REP	Repeated port access
2	STR	String based port access (INS, OUTS)
1	—	Reserved
0	TYPE	Access Type (0 = OUT instruction, 1 = IN instruction)

Figure 15-2. EXITINFO1 for IOIO Intercept

The rIP of the instruction *following* the IN/OUT is saved in EXITINFO2, so that the VMM can easily resume the guest after I/O emulation.

15.10.3 (REP) OUTS and INS

Bits 12:10 of the EXITINFO1 field provide the effective segment number (the default segment is DS). (For segment register encodings, see Table A-32, "16-Bit Register and Memory References" on page 478, in *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*.)

INS provides the effective segment (always ES, encoded as 0).

On intercepted SMI-on-I/O, bits 12:10 of EXITINFO1 encode the segment. For definitions of the remaining bits of this field, (section 15.13.3).

15.11 MSR Intercepts

The VMM can intercept RDMSR and WRMSR instructions by means of the *SVM MSR permissions map* (MSRPM) on a per-MSR basis.

MSR Permissions Map. The MSR permissions bitmap consists of four separate bit vectors of 16 Kbits (2 Kbytes) each. Each 16 Kbit vector controls guest access to a defined range of 8K MSRs. Each MSR is covered by two bits defining the guest read and write access permissions. The lsb of the two bits controls read access to the MSR and the msb controls write access. A value of 1 indicates that the operation is intercepted. The four separate bit vectors must be packed together and located in two contiguous physical pages of memory. If the MSR_PROT intercept is active, any attempt to read or write an MSR not covered by the MSRPM will automatically cause an intercept.

The following table defines the ranges of MSRs covered by the MSR permissions map. Note that the MSR ranges are not contiguous.

Table 15-8. MSR Ranges Covered by MSRPM

MSRPM Byte Offset	MSR Range
000h–7FFh	0000_0000h–0000_1FFFh
800h–FFFh	C000_0000h–C000_1FFFh
1000h–17FFh	C001_0000h–C001_1FFFh
1800h–1FFFh	Reserved

The MSRPM is accessed by physical address and should reside in memory that is mapped as writeback (WB). The MSRPM must be aligned on a 4KB boundary. The physical base address of the MSRPM is specified in MSRPM_BASE_PA field in the VMCB and is loaded into the processor by the VMRUN instruction. The VMRUN instruction ignores the lower 12 bits of the address specified in the VMCB, and if the address of the last byte in the table is greater than or equal to the maximum supported physical address, this is treated as illegal VMCB state and causes a #VMEXIT(VMEXIT_INVALID).

RDMSR and WRMSR Behavior. If the MSR_PROT bit in the VMCB's intercept vector is clear, RDMSR/WRMSR instructions are not intercepted.

RDMSR and WRMSR instructions check for exceptions and intercepts in the following order:

- Exceptions common to all MSRs (e.g., #GP if not at CPL 0)
- Check SVM intercepts in the MSR permission map, if the MSR_PROT intercept is requested.

- Exceptions specific to a given MSR (including password protection, unimplemented MSRs, reserved bits, etc.)

MSR Intercept Information. On #VMEXIT, the processor indicates in the VMCB's EXITINFO1 whether a RDMSR (EXITINFO1 = 0) or WRMSR (EXITINFO1 = 1) was intercepted.

15.12 Exception Intercepts

When intercepting exceptions that define an error code (normally pushed onto the exception stack), the SVM hardware delivers that error code in the VMCB's EXITINFO1 field; the exception vector number can be derived from the EXITCODE. The CS.SEL and rIP saved in the VMCB on an exception-intercept match those that would otherwise have been pushed onto the exception stack frame, except that when an interrupt-based instruction causes an intercept, the rIP of the instruction is stored in the VMCB, rather than the rIP of the next instruction. The interrupt-based instructions are INT3 (opcode CC), INTO, and BOUND.

Unless otherwise noted below, no special registers are written before an exception is intercepted. For details on guest state saved in the VMCB, see section 15.7.1.

External interrupts and software interrupts (INT*n* instruction) do not check the exception intercepts, even when they use a vector in the range 0 to 31.

Exceptions that occur during the handling of a prior exception are checked for intercepts *before* being combined with the prior exception (e.g., into a double-fault). If the result of combining exceptions is a double-fault or shutdown, the processor checks whether those are intercepted before attempting delivery.

Example: Assume that the VMM intercepts #GP and #DF exceptions, and the guest raises a (non-intercepted) #NP, during the delivery of which it also gets a #GP (e.g., due to an illegal IDT entry)—a situation that, according to x86 semantics, results in a #DF. In this case, #VMEXIT signals an intercepted #GP, *not* an intercepted #DF and fills EXITINTINFO with the #NP fault. On the other hand, if only the #DF intercept were active in this scenario, #VMEXIT would signal an intercepted #DF.

The following subsections detail the individual intercepts.

15.12.1 #DE (Divide By Zero)

The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.2 #DB (Debug)

The #DB exception can have fault-type (e.g., instruction breakpoint) or trap-type (e.g., data breakpoint) behavior; accordingly the intercept differs in what state is saved in the VMCB (see section 15.7.1). In either case, however, the value saved for DR6 and DR7 matches what would be visible to a #DB exception handler (i.e., both #DB faults and traps are permitted to write DR6 and DR7 before the intercept). The EXITINFO1 and EXITINFO2 fields are undefined.

Fault-type #DB exceptions, whether indicated in EXITCODE or EXITINTINFO, cause the CS:rIP saved in the VMCB to indicate the instruction that caused the #DB exception. Trap-type #DB exceptions cause the VMCB's CS:rIP to indicate the instruction following the instruction that caused the exception. A vector 1 exception generated by the single byte INT1 instruction (also known as ICEBP) does not trigger the #DB intercept. Software should use the dedicated ICEBP intercept to intercept ICEBP (see section 15.9).

15.12.3 Vector 2 (Reserved)

This intercept bit is not implemented; use the NMI intercept (section 15.13.2) instead. The effect of setting this bit is undefined.

15.12.4 #BP (Breakpoint)

This intercept applies to the trap raised by the single byte INT3 (opcode CCh) instruction. The EXITINFO1 and EXITINFO2 fields are undefined. The CS:rIP reported on #VMEXIT are those of the INT3 instruction.

15.12.5 #OF (Overflow)

This intercept applies to the trap raised by the INTO (opcode CEh) instruction. The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.6 #BR (Bound-Range)

This intercept applies to the fault raised by the BOUND instruction. The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.7 #UD (Invalid Opcode)

The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.8 #NM (Device-Not-Available)

The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.9 #DF (Double Fault)

The EXITINFO1 and EXITINFO2 fields are undefined. The rIP value saved in the VMCB is undefined (as is the case for the rIP value pushed on the stack for #DF exceptions). If a double fault is intercepted, the exceptions leading up to the double fault will have written any status registers normally written by those exceptions.

15.12.10 Vector 9 (Reserved)

This intercept is not implemented. The effect of setting this bit is undefined.

15.12.11 #TS (Invalid TSS)

The EXITINFO1 and EXITINFO2 fields are undefined. The rIP value saved in the VMCB may point to either the instruction causing the task switch, or to the first instruction of the incoming task. See section 15.14.1 for information on the EXITINFO1 and EXITINFO2 fields.

15.12.12 #NP (Segment Not Present)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #NP exception. The EXITINFO2 field is undefined.

15.12.13 #SS (Stack Fault)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #SS exception. The EXITINFO2 field is undefined.

15.12.14 #GP (General Protection)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #GP exception.

15.12.15 #PF (Page Fault)

This intercept is tested *before* CR2 is written by the exception. The error code saved in EXITINFO1 is the same as would be pushed onto the stack by a non-intercepted #PF exception in protected mode. The faulting address is saved in the EXITINFO2 field in the VMCB. Even when the guest is running in paged real mode, the processor will deliver the (protected-mode) page-fault error code in EXITINFO1, for the VMM to use in analyzing the intercepted #PF. The processor may provide additional instruction decode assist information. (See section 15.8.4.)

15.12.16 #MF (X87 Floating Point)

This intercept is tested *after* the floating point status word has been written, as is the case for a normal FP exception. The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.17 #AC (Alignment Check)

The EXITINFO1 field contains the error code that would be pushed on the stack by an #AC exception. The EXITINFO2 field is undefined.

15.12.18 #MC (Machine Check)

The SVM intercept is checked after all #MC-specific registers have been written, but before other guest state is modified. When #MC is being intercepted, a machine-check exits to the VMM, whenever possible, and shuts down the processor only when this is not a reasonable option. The EXITINFO1 and EXITINFO2 fields are undefined.

Note that in some processors, if the guest VM has disabled machine check handling (CR4.MCE=0) then all machine check errors that occur in the guest will result in a shutdown event. However in

processors where CPUID Fn8000_000A_EDX[HOST_MCE_OVERRIDE] (bit 23) = 1 the VMM may override this behavior by setting CR4.MCE=1 in the host. In this scenario, machine check errors that occur in the guest and which can be contained by the processor will always result in a #VMEXIT(MC).

15.12.19 #XF (SIMD Floating Point)

This intercept is tested after the SIMD status word (MXCSR) has been written, as is the case for a normal FP exception. The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.20 #SX (Security Exception)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #SX exception. The EXITINFO2 field is undefined.

15.12.21 #CP (Control Protection)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #CP exception. The EXITINFO2 field is undefined.

15.13 Interrupt Intercepts

External interrupts, when intercepted, cause a #VMEXIT; the interrupt is held pending so that the interrupt can eventually be taken in the VMM. Exception intercepts do not apply to external or software interrupts, so it is not possible to intercept an interrupt by means of the exception intercepts, even if the interrupt should happen to use a vector in the range from 0 to 31.

15.13.1 INTR Intercept

This intercept affects physical, as opposed to virtual, maskable interrupts. See “Virtual Interrupt Intercept” on page 519 for virtualization of maskable interrupts.

15.13.2 NMI Intercept

This intercept affects non-maskable interrupts. NMI interrupts (and SMIs) may be blocked for one instruction following an STI.

15.13.3 SMI Intercept

This intercept affects System Management Mode Interrupts (SMIs); see “SMM Support” on page 521 for details on SMI handling.

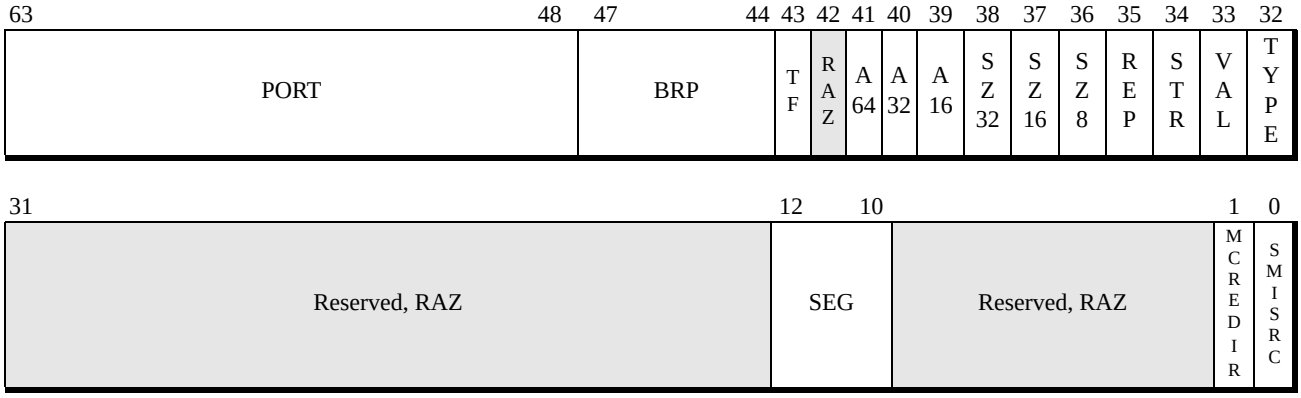
When this intercept triggers, bit 0 of the EXITINFO1 field distinguishes whether the SMI was caused internally by I/O Trapping (bit 0 = 0), or asserted externally (bit 0 = 1).

If the SMI was asserted while the guest was executing an I/O instruction, extra information (describing the I/O instruction) is saved in the upper 32 bits of EXITINFO1, and the rIP of the I/O instruction is

saved in EXITINFO2. EXITINFO1 indicates that SMI was asserted during an I/O instruction when the VALID bit is set.

If the SMI wasn't asserted during an I/O instruction, the extra EXITINFO1 and EXITINFO2 bits are undefined.

The SMI intercept is ignored when HWCR[SMMLOCK] is set.



Bits	Mnemonic	Description
63:48	PORT	Intercepted I/O port
47:44	BRP	I/O breakpoint matches
43	TF	EFLAGS TF value
42	—	Reserved, RAZ
41	A64	64-bit address
40	A32	32-bit address
39	A16	16-bit address
38	SZ32	32-bit operand size
37	SZ16	16-bit operand size
36	SZ8	8-bit operand size
35	REP	Repeated port access
34	STR	String based port access (INS, OUTS)
33	VAL	Valid (SMI was detected during an I/O instruction)
32	TYPE	Access Type (0 = OUT instruction, 1 = IN instruction)
31:13	—	Reserved, RAZ
12:10	SEG	Effective segment number (see section 15.9)
9:2	—	Reserved, RAZ
1	MCREDIR	SMI was due to a redirect machine check error (See “Interaction with SMI and #MC” on page 585)
0	SMISRC	SMI source (0 = internal, 1 = external)

Figure 15-3. EXITINFO1 for SMI Intercept

15.13.4 INIT Intercept

The INIT intercept allows the VMM to intercept the assertion of INIT while a guest is running. An intercepted INIT remains pending until the VMM sets GIF (see “Global Interrupt Flag, STGI and

CLGI Instructions” on page 514), at which point it either takes effect or is redirected. See section 15.21.8 for a discussion of the INIT redirection feature.

15.13.5 Virtual Interrupt Intercept

This intercept is taken just before a guest takes a virtual interrupt. When the intercept triggers, the virtual interrupt has not been taken, and remains pending in the guest's VMCB V_IRQ field. This intercept is not required for handling fixed local APIC interrupts, but may be used for emulating ExtINT interrupt delivery mode (which is not masked by the TPR), or legacy PICs in auto-EOI mode.

15.14 Miscellaneous Intercepts

The SVM architecture includes intercepts to handle task switches, processor freezes due to FERR, and shutdown operations.

15.14.1 Task Switch Intercept

Checked by—Any instruction or event that causes a task switch (e.g., JMP, CALL, exceptions, interrupts, software interrupts).

Priority—The intercept is checked before the task switch takes place but *after* the incoming TSS and task gate (if one was involved) have been checked for correctness.

Task switches can modify several resources that a VMM may want to protect (CR3, EFLAGS, LDT). However, instead of checking various intercepts (e.g., CR3 Write, LDTR Write) individually, task switches check only a single intercept bit.

On #VMEXIT, the following information is delivered in the VMCB:

- EXITINFO1[15:0] holds the segment selector identifying the incoming TSS.
- EXITINFO2[31:0] holds the error code to push in the new task, if applicable; otherwise, this field is undefined.
- EXITINFO2[63:32] holds auxiliary information for the VMM:
  - EXITINFO2[36]—Set to 1 if the task switch was caused by an IRET; else cleared to 0.
  - EXITINFO2[38]—Set to 1 if the task switch was caused by a far jump; else cleared to 0.
  - EXITINFO2[44]—Set to 1 if the task switch has an error code; else cleared to 0.
  - EXITINFO2[48]—The value of EFLAGS.RF that would be saved in the outgoing TSS if the task switch were not intercepted.

15.14.2 Ferr_Freeze Intercept

Checked when the processor freezes due to assertion of FERR (while IGNNE is deasserted, and legacy handling of FERR is selected in CR0.NE), i.e., while the processor is waiting to be unfrozen by an external interrupt.

15.14.3 Shutdown Intercept

When this intercept occurs, any condition that normally causes a shutdown causes a #VMEXIT to the VMM instead. After an intercepted shutdown, the state saved in the VMCB is undefined.

15.14.4 Pause Intercept Filtering

On processors that support Pause filtering (indicated by CPUID Fn8000_000A_EDX[PauseFilter] = 1), the VMCB provides a 16 bit PAUSE Filter Count value. On VMRUN this value is loaded into an internal counter. Each time a PAUSE instruction is executed, this counter is decremented until it reaches zero at which time a #VMEXIT is generated if PAUSE intercept is enabled. If the PAUSE Filter Count is set to zero and PAUSE Intercept is enabled, every PAUSE instruction will cause a #VMEXIT.

In addition, some processor families support Advanced Pause Filtering (indicated by CPUID Fn8000_000A_EDX[PauseFilterThreshold] = 1). In this mode, a 16-bit PAUSE Filter Threshold field is added in the VMCB. The threshold value is a cycle count that is used to reset the pause counter.

As with simple Pause filtering, VMRUN loads the PAUSE count VMCB value into an internal counter. Then, on each PAUSE instruction the hardware checks the elapsed number of cycles since the most recent PAUSE instruction against the PAUSE Filter Threshold. If the elapsed cycle count is greater than the PAUSE Filter Threshold, then the internal pause count is reloaded from the VMCB and execution continues. If the elapsed cycle count is less than the PAUSE Filter Threshold, then the internal pause count is decremented. If the count value is less than zero and PAUSE intercept is enabled, a #VMEXIT is triggered.

If Advanced Pause Filtering is supported and PAUSE Filter Threshold field is set to zero, the filter will operate in the simpler, count only mode.

See Section 3.3, “Processor Feature Identification,” on page 71 for more information on using the CPUID instruction.

15.15 VMCB State Caching

VMCB state caching allows the processor to cache certain guest register values in hardware between a #VMEXIT and subsequent VMRUN instructions and use the cached values to improve context-switch performance. Depending on the particular processor implementation, VMRUN loads each guest register value either from the VMCB or from the VMCB state cache, as specified by the value of the VMCB Clean field in the VMCB. Support for VMCB state caching is indicated by CPUID Fn8000_000A_EDX[VmcbClean] = 1.

The SVM architecture uses the physical address of the VMCB as a unique identifier for the guest virtual CPU for the purposes of deciding whether the cached copy belongs to the guest. For the purposes of VMCB state caching, the ASID is not a unique identifier for a guest virtual CPU.

15.15.1 VMCB Clean Bits

The VMCB Clean field (VMCB offset 0C0h, bits 31:0) controls which guest register values are loaded from the VMCB state cache on VMRUN. Each set bit in the VMCB Clean field allows the processor to load one guest register or group of registers from the hardware cache; each clear bit requires that the processor load the guest register from the VMCB. The clean bits are a hint, since any given processor implementation may ignore bits that are set to 1 on any given VMRUN, unconditionally loading the associated register value(s) from the VMCB. Clean bits that are set to zero are always honored.

This field is backward-compatible to CPUs that do not support VMCB state caching; older CPUs neither cache VMCB state nor read the VMCB Clean field.

Older hypervisors that are not aware of VMCB state caching and obey the SBZ property of undefined VMCB fields will not enable VMCB state caching.

15.15.2 Guidelines for Clearing VMCB Clean Bits

The hypervisor must clear specific bits in the VMCB Clean field every time it explicitly modifies the associated guest state in the VMCB. The guest's execution can cause cached state to be updated, but the hypervisor is not responsible for setting VMCB Clean bits corresponding to any state changes caused by guest execution.

The hypervisor must clear the entire VMCB field to 0 for a guest, under the following circumstances:

- This is the first time a particular guest is run.
- The hypervisor executes the guest on a different CPU core than one used the last time that guest was executed.
- The hypervisor has moved the guest's VMCB to a different physical page since the last time that guest was executed.

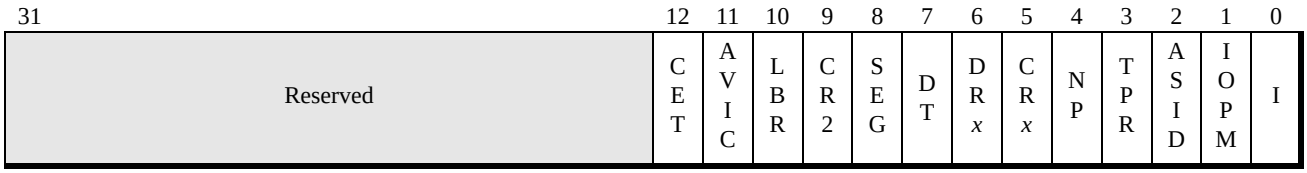
Failure to clear the VMCB Clean bits to zero in these cases may result in undefined behavior.

The CPU automatically treats the VMCB Clean field as zero on the current VMRUN when the hypervisor executes a guest that is not currently cached. The CPU compares the VMCB physical address against all cached VMCB physical addresses and treats the VMCB Clean field as zero, if no cached VMCB address matches.

SMM software (or any other agent external to the hypervisor that has access to VMCBs) that changes the contents of a VMCB needs to comprehend the clean bits and adjust them accordingly; otherwise the guest may not operate as intended.

15.15.3 VMCB Clean Field

The layout of the VMCB Clean field is illustrated in Figure 15-4 below.



Bits	Mnemonic	Description
31:13	—	Reserved
12	CET	S_CET, SSP, ISST_ADDR
11	AVIC	AVIC APIC_BAR; AVIC APIC_BACKING_PAGE, AVIC PHYSICAL_TABLE and AVIC LOGICAL_TABLE Pointers
10	LBR	DbgCtlMsr, br_from/to, lastint_from/to
9	CR2	CR2
8	SEG	CS/DS/SS/ES Sel/Base/Limit/Attr, CPL
7	DT	GDT/IDT Limit and Base
6	DR*x*	DR6, DR7
5	CR*x*	CR0, CR3, CR4, EFER
4	NP	Nested Paging: NCR3, G_PAT
3	TPR	V_TPR, V_IRQ, V_INTR_PRIO, V_IGN_TPR, V_INTR_MASKING, V_INTR_VECTOR (Offset 60h–67h)
2	ASID	ASID
1	IOPM	IOMSRPM: IOPM_BASE, MSRPM_BASE
0	I	Intercepts: all the intercept vectors, TSC offset, Pause Filter Count

Figure 15-4. Layout of VMCB Clean Field

Bits 31:12 are reserved for future implementations. For forward compatibility, if the hypervisor has not modified the VMCB, the hypervisor may write FFFF_FFFFh to the VMCB Clean Field to indicate that it has not changed any VMCB contents other than the fields described below as explicitly uncached. The hypervisor should write 0h to indicate that the VMCB is new or potentially inconsistent with the CPU's cached copy, as occurs when the hypervisor has allocated a new location for an existing VMCB from a list of free pages and does not track whether that page had recently been used as a VMCB for another guest. If any VMCB fields (excluding explicitly uncached fields) have been modified, all clean bits that are undefined (within the scope of the hypervisor) must be cleared to zero.

Bit 31 is a special bit reserved for the host that the CPU will never use for a clean bit.

The following are explicitly not cached and not represented by Clean bits:

- TLB_Control
- Interrupt shadow
- VMCB status fields (Exitcode, EXITINFO1, EXITINFO2, EXITINTINFO, Decode Assist, etc.)
- Event injection
- RFLAGS, RIP, RSP, RAX

15.16 TLB Control

TLB entries are tagged with *Address Space Identifier* (ASID) bits to distinguish different guest virtual address spaces when shadow page tables are used, or different guest physical address spaces when nested page tables are used. The VMM can choose a software strategy in which it keeps multiple shadow page tables, and/or multiple nested page tables in processors that support nested paging, up-to-date; the VMM can allocate a different ASID for each shadow or nested page table. This allows switching to a new process in a guest under shadow paging (changing CR3 contents), or to a new guest under nested paging (changing nCR3 contents), without flushing the TLBs. (See section 15.25 for a complete explanation of nested paging operation.)

With shadow paging, the VMM is responsible for setting up a shadow page table for each guest linear address space that maps it to system physical addresses. These are used as the active page tables in place of the guest OS's page tables. The VMM sets the CR3 field in the guest VMCB to point to the system physical address of the desired shadow page table. The VMM is responsible for updating the shadow page table when the guest changes its page table or paging control state, and the VMM updates the access and dirty bits of the guest page table.

The VMRUN instruction and #VMEXIT write the CR0, CR3, CR4 and EFER registers, but these writes do *not* flush the TLB. The VMM is responsible for explicitly invalidating any guest translations that may be affected by its actions. There are two mechanisms available for this described in the next two sections.

When running with SVM enabled, global page table entries (PTEs) are global only *within* an ASID, not across ASIDs.

Software Rule. When the VMM changes a guest’s paging mode by changing entries in the guest’s VMCB, the VMM must ensure that the guest’s TLB entries are flushed from the TLB. The relevant VMCB state includes:

- CR0—PG, WP, CD, NW.
- CR3—Any bit.
- CR4—PGE, PAE, PSE.
- EFER—NXE, LMA, LME.

15.16.1 TLB Flush

TLB flush operations function identically whether or not SVM is enabled (e.g., MOV-TO-CR3 flushes non-global mappings, whereas MOV-TO-CR4 flushes global and non-global mappings). TLB flush operations must not be assumed to affect all ASIDs. If a VMM sets the intercept bit for any guest action that would have flushed the TLB, the #VMEXIT intercept occurs and the TLB is not flushed; it is the VMM's responsibility to flush the TLB appropriately. In implementations that do not provide a way to selectively flush all translations of a single specified ASID, software may effectively flush the guest's TLB entries by allocating a new ASID for the guest and not reusing the old ASID until the entire TLB has been flushed at least once.

The TLB_CONTROL field in the VMCB provides the commands specified by the control byte encodings shown in Table 15-9. The first two commands are available on all processors that support SVM; support for the other commands is optional and is indicated by CPUID Fn8000_000A_EDX[FlushByAsid] = 1.

Table 15-9. TLB Control Byte Encodings

Encoding	Function Definition
00h	Do not flush
01h	Flush entire TLB (Should be used only on legacy hardware.)
03h	Flush this guest's TLB entries
07h	Flush this guest's non-global TLB entries
Note: All encodings not defined in this table are reserved.	

When the VMM sets the TLB_CONTROL field to 1, the VMRUN instruction flushes the TLB for all ASIDs, for both global and non-global pages. The VMRUN instruction reads, but does not change, the value of the TLB_CONTROL field.

A MOV-to-CR3, a task switch that changes CR3, or clearing or setting CR0.PG or bits PGE, PAE, PSE of CR4 affects only the TLB entries belonging to the current ASID, regardless of whether the operation occurred in host or guest mode. The current ASID is 0 when the CPU is not inside a guest context.

All TLB entries belonging to all ASIDs are flushed by SMI, RSM, MTRR modifications, IORR modifications, and access to other system MSRs that affect address translation.

If a hypervisor modifies a nested page table by decreasing permission levels, clearing present bits, or changing address translations and intends to return to the same ASID, it should use either TLB command 011b or 001b.

15.16.2 Invalidate Page, Alternate ASID

The INVLPGA instruction allows the VMM to selectively invalidate the TLB mapping for a given guest virtual page within a given ASID. The linear address is specified in the implicit register operand rAX; the ASID is specified in ECX. The input address is always interpreted as a guest virtual address, so INVLPGA is typically meaningful only when used with shadow page tables; it does not provide a means to invalidate a nested translation by guest physical address.

15.17 Global Interrupt Flag, STGI and CLGI Instructions

The global interrupt flag (GIF) is a bit that controls whether interrupts and other events can be taken by the processor. The STGI and CLGI instructions set and clear, respectively, the GIF. Table 15-10 shows how the value of the GIF affects how interrupts and exceptions are handled. Implementations may

provide hardware support for virtualizing the GIF in nested virtualization scenarios; see section 15.33, for details.

Table 15-10. Effect of the GIF on Interrupt Handling

Interrupt source	GIF==0	GIF ==1
Debug exception or trap, due to breakpoint register match	Ignored and discarded	Normal operation
Debug trace trap due to EFLAGS.TF	Normal operation	Normal operation
RESET	Normal operation	Normal operation
INIT	Held pending until GIF==1	Normal operation, see Table 15-12
NMI	Held pending until GIF==1	Normal operation, see Table 15-13
External SMI	Held pending until GIF==1	Normal operation, see Table 15-14
Internal SMI (I/O Trapping)	Ignored and discarded	Normal operation, see Table 15-14
INTR and vINTR	Held pending until GIF==1	Normal operation
#SX (Security Exception)	n/a[1]	Normal operation
Machine Check	If possible (implementation-dependent), held pending until GIF==1, otherwise shutdown.	Normal operation
DBREQ# (external debug request)	Normal operation	Normal operation
	(VM_CR.DPD always controls DBREQ)	
A20M	Normal operation	Normal operation
	(VM_CR.DIS_A20M controls A20 masking)	
Other implementation-specific but non-architecturally-visible interrupts (STPCLK, IGNNE toggle, ECC scrub)	Normal operation	Normal operation

Note:

1. #SX is caused only by an INIT signal that has been "redirected" (i.e., converted to an #SX; see section 15.28); the conversion only happens when GIF==1, as the INIT is simply held pending otherwise.

15.18 VMMCALL Instruction

This instruction is meant as a way for a guest to explicitly call the VMM. No CPL checks are performed, so the VMM can decide whether to make this instruction legal at the user-level or not.

If VMMCALL instruction is not intercepted, the instruction raises a #UD exception.

15.19 Paged Real Mode

To facilitate virtualization of real mode, the VMRUN instruction may legally load a guest CR0 value with PE = 0 but PG = 1. Likewise, the RSM instruction is permitted to return to paged real mode. This

processor mode behaves in every way like real mode, with the exception that paging is applied. The intent is that the VMM run the guest in paged-real mode at CPL0, and with page faults intercepted. The VMM is responsible for setting up a shadow page table that maps guest *physical* memory to the appropriate system physical addresses.

The behavior of running a guest in paged real mode without intercepting page faults to the VMM is undefined.

15.20 Event Injection

The VMM can inject exceptions or interrupts (collectively referred to as events) into the guest by setting bits in the VMCB's EVENTINJ field prior to executing the VMRUN instruction. The format of the field is shown in Table 15-5. The encoding matches that of the EXITINTINFO field. When an event is injected by means of this mechanism, the VMRUN instruction causes the guest to take the specified exception or interrupt unconditionally before executing the first guest instruction.

Injected events are treated in every way as though they had occurred normally in the guest (in particular, they are recorded in EXITINTINFO) with the following exceptions:

- Injected events are not subject to intercept checks. (Note, however, that if secondary exceptions occur during delivery of an injected event, those exceptions *are* subject to exception intercepts.)
- An injected NMI does not block delivery of further NMIs.
- If the VMM attempts to inject an event that is impossible for the guest mode (e.g., a #BR exception when the guest is in 64-bit mode), the event injection will fail and no guest state instructions will be executed; VMRUN will immediately exit with an error code of VMEXIT_INVALID.
- Injecting an exception (TYPE = 3) with vectors 3 or 4 behaves like a trap raised by INT3 and INTO instructions, respectively, in which case the processor checks the DPL of the IDT descriptor before dispatching to the handler.
- Software interrupts cannot be properly injected if the processor does not support the NextRIP field. Support is indicated by CPUID Fn8000_000A_EDX[NRIPS] = 1. Hypervisor software should emulate the event injection of software interrupts if NextRIP is not supported.
- Event injection does not support injection of intercepted #DB faults that are the result of a guest ICEBP instruction. ICEBP does not perform DPL checks, as does INT*n* injection. Hypervisor software should emulate the injection of ICEBP.

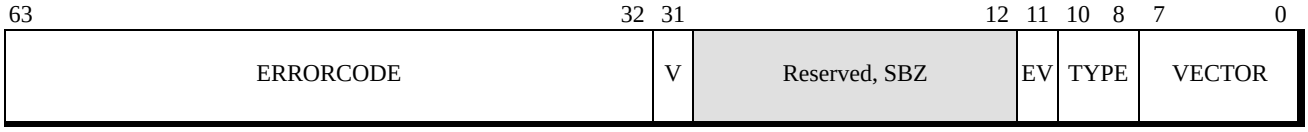


Figure 15-5. EVENTINJ Field in the VMCB

The fields in EVENTINJ are as follows:

- *VECTOR*—Bits 7:0. The 8-bit IDT vector of the interrupt or exception. If TYPE is 2 (NMI), the VECTOR field is ignored.
- *TYPE*—Bits 10:8. Qualifies the guest exception or interrupt to generate. Table 15-11 shows possible values and their corresponding interrupt or exception types. Values not indicated are unused and reserved.

Table 15-11. Guest Exception or Interrupt Types

Value	Type
0	External or virtual interrupt (INTR)
2	NMI
3	Exception (fault or trap)
4	Software interrupt (INT*n* instruction)

- *EV (Error Code Valid)*—Bit 11. Set to 1 if the exception should push an error code onto the stack; clear to 0 otherwise.
- *V (Valid)*—Bit 31. Set to 1 if an event is to be injected into the guest; clear to 0 otherwise.
- *ERRORCODE*—Bits 63:32. If EV is set to 1, the error code to be pushed onto the stack, ignored otherwise.

VMRUN exits with VMEXIT_INVALID error code if either:

- Reserved values of TYPE have been specified, or
- TYPE = 3 (exception) has been specified with a vector that does not correspond to an exception (this includes vector 2, which is an NMI, not an exception).

15.21 Interrupt and Local APIC Support

SVM hardware support is designed to ensure efficient virtualization of interrupts.

15.21.1 Physical (INTR) Interrupt Masking in EFLAGS

To prevent the guest from blocking maskable interrupts (INTR), SVM provides a VMCB control bit, V_INTR_MASKING, which changes the operation of EFLAGS.IF and accesses to the TPR by means of the CR8 register. While running a guest with V_INTR_MASKING cleared to zero:

- EFLAGS.IF controls both virtual and physical interrupts.

While running a guest with V_INTR_MASKING set to 1:

- The host EFLAGS.IF at the time of the VMRUN is saved and controls physical interrupts while the guest is running.
- The guest value of EFLAGS.IF controls virtual interrupts only.

15.21.2 Virtualizing APIC.TPR

SVM provides a virtual TPR register, V_TPR, for use by the guest; its value is loaded from the VMCB by VMRUN and written back to the VMCB by #VMEXIT. The APIC's TPR always controls the task priority for physical interrupts, and the V_TPR always controls virtual interrupts.

While running a guest with V_INTR_MASKING cleared to 0:

- Writes to CR8 affect both the APIC's TPR and the V_TPR register.
- Reads from CR8 operate as they would without SVM.

While running a guest with V_INTR_MASKING set to 1:

- Writes to CR8 affect only the V_TPR register.
- Reads from CR8 return V_TPR.

15.21.3 TPR Access in 32-Bit Mode

The mechanism for TPR virtualization described in section 15.21.2 applies only to accesses that are performed using the CR8 register. However, in 32-bit mode, the TPR is traditionally accessible only by using a memory-mapped register. Typically, a VMM virtualizes such TPR accesses by not mapping the APIC page addresses in the guest. A guest access to that region then causes a #PF intercept to the VMM, which inspects the guest page tables to determine the physical address and, after recognizing the physical address as belonging to the APIC, finally invokes software emulation code.

To improve the efficiency of TPR accesses in 32-bit mode, SVM makes CR8 available to 32-bit code by means of an alternate encoding of MOV TO/FROM CR8 (namely, MOV TO/FROM CR0 with a LOCK prefix). To achieve better performance, 32-bit guests should be modified to use this access method, instead of the memory-mapped TPR. (For details, see “MOV CRn” on page 377 of the *AMD64 Programmer’s Reference Volume 3: General Purpose and System Instructions*, order# 24594.)

The alternate encodings of the MOV TO/FROM CR8 instructions are available even if SVM is disabled in EFER.SVME. They are available in both 64-bit and 32-bit mode.

15.21.4 Injecting Virtual (INTR) Interrupts

Virtual Interrupts allow the host to pass an interrupt (#INTR) to a guest. While inside a guest, the virtual interrupt follows the same rules that a real interrupt follows (virtual #INTR is not taken until EFLAGS.IF is 1, the guest's TPR has enabled interrupts at the same priority as that of the pending virtual interrupt).

SVM provides an efficient mechanism by which the VMM can inject virtual interrupts into a guest:

- As described in Section 15.13.1, the VMM can intercept physical interrupts that arrive while a guest is running, by activating the INTR intercept in the VMCB.
- As described in Section 15.21.4, the VMM can virtualize the interrupt masking logic by setting the V_INTR_MASKING bit in the VMCB.

- The three VMCB fields V_IRQ, V_INTR_PRIO, and V_INTR_VECTOR indicate whether there is a virtual interrupt pending, and, if so, what its vector number and priority are. The VMRUN instruction loads this information into corresponding on-chip registers.
- The processor takes a virtual INTR interrupt if
  - V_IRQ and V_INTR_PRIO indicate that there is a virtual interrupt pending whose priority is greater than the value in V_TPR,
  - interrupts are enabled in EFLAGS.IF,
  - interrupts are enabled using GIF, and
  - the processor is not in an interrupt shadow (see Section 15.21.5).

  The only other difference between virtual INTR handling and normal interrupt handling is that, in the latter case, the interrupt vector is obtained from the V_INTR_VECTOR register (as opposed to running an INTACK cycle to the local APIC).
- The V_IGN_TPR field in the VMCB can be set to indicate that the currently pending virtual interrupt is not subject to masking by TPR. The priority comparison against V_TPR is omitted in this case. This mechanism can be used to inject ExtINT-type interrupts into the guest.
- When the processor dispatches a virtual interrupt (through the IDT), V_IRQ is cleared after checking for intercepts of virtual interrupts and before the IDT is accessed.
- On #VMEXIT, V_IRQ is written back to the VMCB, allowing the VMM to track whether a virtual interrupt has been taken.
- Physical interrupts take priority over virtual interrupts, whether they are taken directly or through a #VMEXIT.
- On #VMEXIT, the processor clears its internal copies of V_IRQ and V_INTR_MASKING, so virtual interrupts do not remain pending in the VMM, and interrupt control reverts to normal.

15.21.5 Interrupt Shadows

The x86 architecture defines the notion of an *interrupt shadow*—a single-instruction window during which interrupts are not recognized. For example, the instruction after an STI instruction that sets EFLAGS.IF (from zero to one) does not recognize interrupts or certain debug traps. The VMCB INTERRUPT_SHADOW field indicates whether the guest is currently in an interrupt shadow. This information is saved on #VMEXIT and loaded on VMRUN.

15.21.6 Virtual Interrupt Intercept

When virtualizing interrupt handling, a VMM typically needs only gain control when new interrupts for a guest arrive or are generated, and when the guest issues an EOI (end-of-interrupt). In some circumstances, it may also be necessary for the VMM to gain control at the moment interrupts become enabled in the guest (i.e., just before the guest takes a virtual interrupt). The VMM can do so by enabling the VINTR intercept.

15.21.7 Interrupt Masking in Local APIC

When guests have direct access to devices, interrupts arriving at the local APIC can usually be dismissed only by the guest that owns the device causing the interrupt. To prevent one guest from blocking other guests' interrupts (by never processing their own), the VMM can mask pending interrupts in the local APIC, so they do not participate in the prioritization of other interrupts.

SVM introduces the following APIC features:

- A 256-bit IER (interrupt enable) register is added to the local APIC. This register resets to all ones (enabling all 256 vectors). Software can read and write the IER by means of the memory-mapped APIC page.
- Only vectors that are enabled in the IER participate in the APIC computation of the highest-priority pending interrupt.
- The VMM can issue specific end-of-interrupt (EOI) commands to the local APIC, allowing the VMM to clear pending interrupts in any order, rather than always targeting the interrupt with highest-priority.

15.21.8 INIT Support

The INIT signal interrupts the processor at the next instruction boundary and causes an unconditional control transfer. INIT reinitializes the control registers, segment registers and GP registers in a manner similar to RESET, but does not alter the contents of most MSRs, caches or numeric coprocessor (x87 or SSE) state, and then transfers control to the same instruction address as RESET (physical address FFFFFFF0h). Unlike RESET, INIT is not expected to be visible to the memory controller, and hence will not trigger automatic clearing of trusted memory pages by memory controller hardware.

To maintain the security of such pages, the VMM can request that INITs be redirected and turned into #SX exceptions by setting the R_INIT bit in the VM_CR MSR (see Section 15.30.1). This allows the VMM to gain control when an INIT is requested and scrub any sensitive context. The VMM may then disable the redirection of INIT and cause the platform to reassert INIT (see the relevant BKDG or PPR documentation for details), at which point the processor will respond in the normal manner. The actions initiated by the INIT pin may also be initiated by an incoming APIC INIT interrupt; the mechanisms described here apply in either case. Table 15-12 summarizes the handling of INITs.

Table 15-12. INIT Handling in Different Operating Modes

GIF	INIT Intercept	INIT Redirect	Processor Response to INIT
0	X	X	Hold pending until GIF = 1.
1	1	X	#VMEXIT(INIT), INIT is still pending.
	0	0	Taken normally.
		1	#SX, INIT is no longer pending.

If redirection is enabled without the INIT intercept being enabled, an INIT that asserts during guest execution will result in #SX being asserted within the guest, with the INIT being cleared. The VMM may intercept the assertion of #SX as described in “Intercept Operation” on page 493. Note that when a VMM has intercepted an INIT assertion, it may modify R_INIT any time before setting GIF to control the behavior when GIF is ultimately set, or the VMM may instead return to a guest VM with the intercept disabled and redirection enabled to effectively hand off the INIT to the guest as a #SX exception.

15.21.9 NMI Support

The VMM can intercept non-maskable interrupts (NMI) using a VMCB control bit (see Table 15-13). When intercepted, NMIs cause an exit from the guest and are held pending.

Table 15-13. NMI Handling in Different Operating Modes

GIF	NMI Intercept	Processor Response to NMI
0	X	Hold pending until GIF=1.
1	1	#VMEXIT(NMI), NMI is still pending.
	0	Taken normally.

15.22 SMM Support

This section describes SVM support for virtualization of System Management Mode (SMM).

15.22.1 Sources of SMI

Various events can cause an assertion of a system management interrupt (SMI); these are classified into three categories

- Internal, synchronous (also known as I/O Trapping)—implementation-specific IOIO or config space trapping in the CPU itself; always synchronous in response to an IN or OUT instruction. I/O Trapping is set up by means of MSRs and can be brought under the control of the VMM by intercepting guest access to those MSRs.
- External, synchronous—IOIO trapping in response to (and synchronous with) IN or OUT instructions, but generated by an external agent (typically the Southbridge).
- External, asynchronous—generated externally in response to an external, physical event, e.g., closing a laptop lid, temperature sensor triggering, etc.

15.22.2 Response to SMI

How hardware responds to SMIs is a function of whether SMM interrupts are being intercepted and whether interrupts are enabled globally, as shown in Table 15-14.

Table 15-14. SMI Handling in Different Operating Modes

GIF	Intercept SMI	Internal SMI	External SMI
0	x	Lost.	Hold pending until GIF=1.
1	1	Exit guest, code #VMEXIT(SMI), SMI is not pending.	#VMEXIT(SMI), SMI is still pending.
	0	Taken normally.	Taken normally.

By intercepting SMIs, the VMM can gain control before the processor enters SMM.

15.22.3 Containerizing Platform SMM

In some usage scenarios, the VMM may not trust the existing platform SMM code, or may otherwise want to ensure that the SMM does not operate in the context of certain guests or the hypervisor. To address these cases, SVM provides the ability to *containerize* SMM code, i.e., run it inside a guest, with the full protection mechanisms of the VMM in place. In other scenarios, the VMM may not want to exert control over SMM.

There are three solutions for the VMM to control SMM handlers:

- The simplest solution is to not intercept SMI signals. SMIs encountered while in a guest context are taken from within the guest context. In this case, the SMM handler is not subject to any intercepts set up by the VMM and consequently runs outside of the virtualization controls. The state saved in the SMM State-Save area as seen by the SMM handler reflects the state of the guest that had been running at the time the SMI was encountered. When the SMM handler executes the RSM instruction, the processor returns to executing in the guest context, and any modifications to the SMM State-Save area made by the SMM handler are reflected in the guest state.
- A hypervisor may want to emulate all SMI-based I/O interceptions for a guest and to take SMI signals only in the hypervisor context. The hypervisor should set all IOIO intercept bits and the SMI intercept bit for the guest to ensure that there is no possibility of encountering synchronous (internal or external) SMI signals while running the guest. Any #VMEXIT(SMI) encountered is then known to be due to an external, asynchronous SMI. The hypervisor may respond to the #VMEXIT(SMI) by executing the STGI instruction, which causes the pending SMI to be taken immediately. When an SMI due to an I/O instruction is pending, the effect of executing STGI in the hypervisor is undefined. To handle a pending SMI due to an I/O instruction, the hypervisor must either containerize SMM or not intercept SMI.
- The most involved solution is to containerize SMM by placing it in a guest. Containerizing gives the VMM full control over the state that the SMM handler can access.

Containerizing Platform SMM. A VMM can containerize SMM by creating its own trusted SMM hypervisor and use that handler to run the platform SMM code in a container. The SMM hypervisor may be the same code as the VMM itself, or may be an entirely different set of code. The trusted SMM hypervisor sets up a guest context to run the platform SMM as a guest. The guest context consists of a VMCB and related state and the guest's (real or virtual) SMM save area. The SMM hypervisor

emulates SMM entry, including setup of the SMM save area, and emulates RSM at the end of SMM operation. The guest executes the platform SMM code in paged real mode with appropriate SVM intercepts in place, thus ensuring security.

For this approach to work, the VMM may need to write the SMM_BASE MSR, as well as related SMM control registers. As part of the emulation of SMM entry and RSM, the VMM needs to access the SMM_CTL MSR (see Section 15.30.3). However, these actions conflict with any platform firmware that locks SMM control registers.

A VMM can determine if it is running with a compatible firmware setup by checking the SMMLOCK bit in the HWCR MSR (described in the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product). If the bit is 1, firmware has locked the SMM control registers and the VMM is unable to move them or insert its own SMM hypervisor.

As the processor physically enters SMM, the SMRAM regions are remapped. The VMM design must ensure that none of its code or data disappears when the SMRAM areas are mapped or unmapped. Also note that the ASEG region of the SMRAM overlaps with a portion of video memory, so the SMM hypervisor should not attempt to write diagnostic messages to the screen. Any attempt by guests to relocate any of the SMRAM areas (by means of certain MSR writes) must also be intercepted to prevent malicious SMM code from interfering with VMM operation.

Writes to the SMM_CTL MSR cause a #GP if firmware has locked the SMM control registers.

15.23 Last Branch Record Virtualization

The debug control MSR (DebugCtl) provides control of control-transfer recording and other debug facilities. (See Chapter 13, "Software Debug and Performance Resources," on page 385, for more information on using the debug control MSR.) Software sets the last-branch record (DebugCtl[LBR]) bit to 1 to cause the processor to record the source and target addresses of the last control transfer taken before a debug exception. These control transfers include branch instructions, interrupts, and exceptions. Recorded information is stored in four MSRs:

- LastBranchFromIP
- LastBranchToIP
- LastIntFromIP
- LastIntToIP

Under SVM, to virtualize the function of these MSRs, the VMM must save the contents of the control-transfer recording MSRs on #VMEXIT and restore them prior to the VMRUN for each guest. If control-transfer recording is to be used in host state as well the values of these registers must be exchanged between values tracked by host and guest.

15.23.1 Hardware Acceleration for LBR Virtualization

Processors optionally support hardware acceleration for LBR virtualization. The following fields are allocated in the VMCB state save area to hold the contents of the DebugCTL and control-transfer recording MSRs:

- DBGCTL—Holds the guest value of the DebugCTL MSR.
- BR_FROM—Holds the guest value of the LastBranchFromIP MSR.
- BR_TO—Holds the guest value of the LastBranchToIP MSR.
- LASTEXCPFROM—Holds the guest value of the LastIntFromIP MSR.
- LASTEXCPTO—Holds the guest value of the LastIntToIPLastIntToIP MSR.

When VMCB.LBR_VIRTUALIZATION_ENABLE is set, VMRUN saves all five host control-transfer MSRs in the host save area, and then loads the same five MSRs for the guest from the VMCB save area. Similarly, #VMEXIT saves the guest's MSRs and loads the host's MSRs to and from their respective save areas.

15.23.2 LBR Virtualization CPUID Feature Detection

CPUID Fn8000_000A_EDX[LbrVirt] = 1 indicates support for the LBR virtualization acceleration feature on AMD64 processors. See Section 3.3, “Processor Feature Identification,” on page 71 for more information on using the CPUID instruction.

15.24 External Access Protection

By securing the virtual address translation mechanism, the VMM can restrict guest CPU accesses to memory. However, should the guest have direct access to DMA-capable devices, an additional protection mechanism is required. SVM provides multiple protection domains which can restrict device access to physical memory on a per-page basis. This is accomplished via control logic in the northbridge’s host bridge which governs any external access port (e.g., PCI or HyperTransport™ technology interfaces).

15.24.1 Device IDs and Protection Domains

The northbridge’s host bridge provides a number of protection domains. Each protection domain has associated with it a device exclusion vector (DEV) that specifies the per-page access rights of devices in that domain. Devices are identified by a HyperTransport™ bus/unitID (device ID) and the host bridge contains a lookup table of fixed size that maps device IDs to a protection domain.

15.24.2 Device Exclusion Vector (DEV)

A DEV is a contiguous array of bits in physical memory; each bit in the DEV (in little-endian order) corresponds to one 4-Kbyte page in physical memory.

The physical address of the base of a DEV must be 4-Kbyte-aligned and stored in one of the DEVBASE registers, which are accessed through an indirection mechanism in the DEVCTL PCI

Configuration Space function block in the host bridge (see “DEV Control and Status Registers” on page 528). The DEV protection hardware is not operational until enabled by setting a control bit in the DEV Control Register, also in the DEVCTL function block.

The DEV may have to cover part of MMIO space beyond the DRAM. Especially in 64-bit systems, the operating system should map MMIO space starting immediately after the DRAM area and building up, as opposed to starting down from the maximum physical address.

Host Bridge and Processor DEV Caching. For improved performance, the host bridge may cache portions of the DEV. Any such cached information can be invalidated by setting the DEV_FLUSH flag in the DEV control register to 1. Software must set this flag after modifying DEV contents to ensure that the protection logic uses the updated values. The host bridge automatically clears this flag when the flush operation completes. After setting this flag, software should monitor it until it has cleared, in order to synchronize DEV updates with subsequent activity.

By default, the host bridge probes the processor caches for the latest data when it accesses the DEV in DRAM. However, it is possible to disable probing by means of the DEV_CR register (“DEV_CR Register” on page 528); this is recommended in the case of unified memory architecture (UMA) graphics systems. If cache probing is disabled, host bridge reads of the DEV will not check processor caches for more recent copies. This requires software on the CPU to map the memory containing the DEV as uncacheable (UC) or write-through (WT). Alternatively, software must perform a CLFLUSH before it can expect a change to the DEV to be visible by the northbridge (and before software flushes the DEV cache in the host controller).

Multiprocessor Issues. Device-originated memory requests are checked against the DEV at the point of entry to the system—the northbridge to which the device is physically attached. Each northbridge can have its own set of domains, device-to-domain mappings, and DEV tables (e.g., domain #2 on one node can encompass different devices, and can have different access rights than domain #2 on another node). Thus, the number of protection domains available to software can scale with the number of northbridges in the system.

15.24.3 Access Checking

Memory Space Accesses. When a memory-space read or write request is received on an external host bridge port, the host bridge maps the HyperTransport bus device ID to a protection domain number, which in turn selects the DEV defining the access permissions for the device (see Figure 15-6). The host bridge then checks the memory address against the DEV contents by indexing into the DEV with the PFN portion of the address (bits 39:12). The PFN is used as a bit index within the DEV. If the bit read from the DEV is set to 1, the host bridge inhibits the access by returning all ones for the data for a read request, or suppressing the store operation on a write request. A Master Abort error response will be returned to the requesting device.

Peer-to-peer memory accesses routed up to the host bridge are also subjected to checks against the DEV. Peer-to-peer transfers that may be occurring behind bridges are not checked.

DEV checks are applied before addresses are translated by the GART. The DEV table is never consulted by accesses originating in the CPU.

I/O Space Accesses. The host bridge can be configured to reject all I/O space accesses from devices, by setting the IOSPE bit in the DEV_CR control register (see “DEV_CR Register” on page 528). I/O space peer-to-peer transfers behind bridges are not checked.

Config Space Accesses. Major aspects of host bridge functionality are configured by means of control registers that are accessed through PCI configuration space. Because this is potentially accessible by means of device peer-to-peer transfers, the host bridge always blocks access to this space from anything other than the CPU.

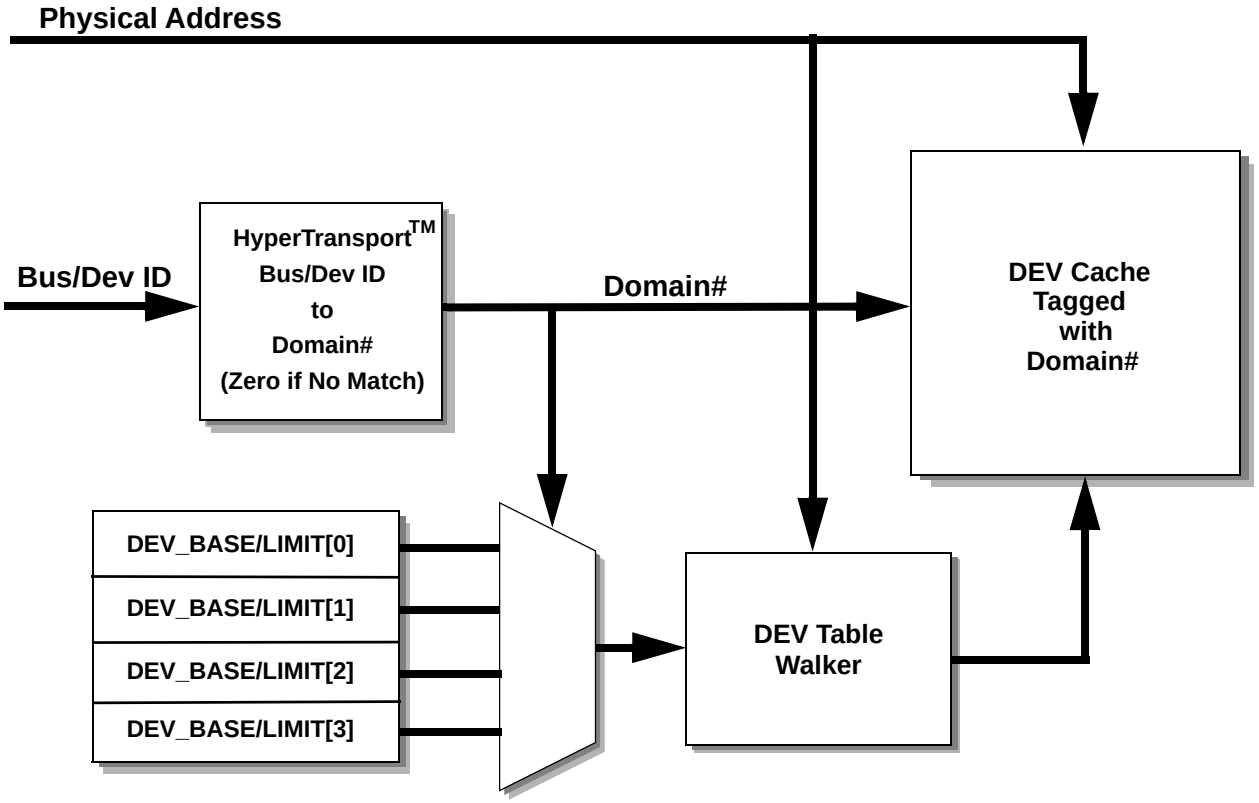


Figure 15-6. Host Bridge DMA Checking

15.24.4 DEV Capability Block

The presence of DEV support is indicated through a new PCI capability block. The capability block also provides access to the registers that control operation of the DEV feature.

The DEV capability block in PCI space contains three 32-bit words: the capability header (DEV_HDR), and two registers (DEV_OP and DEV_DATA) which serve as an indirection mechanism for accessing the actual DEV control and status registers.

Table 15-15. DEV Capability Block, Overall Layout

Byte Offset	Register	Comments
0	DEV_HDR	Capability block header
4	DEV_OP	Selects control/status register to access
8	DEV_DATA	Read/write to access register selected in DEV_OP

DEV Capability Header. The DEV capability header (DEV_HDR) is defined in Table 15-16.

Table 15-16. DEV Capability Header (DEV_HDR) (in PCI Config Space)

Bit(s)	Definition
31:22	Reserved, MBZ
21	Interrupt Reporting Capability
20	Machine Check Exception Reporting Capability
19	Reserved, MBZ
18:16	DEV Capability Block Type; hardwired to 000b.
15:8	PCI Capability pointer; points to next capability in list
7:0	PCI Capability ID; hardwired to 0x0F

15.24.5 DEV Register Access Mechanism

The northbridge's DEV control and status registers are accessed through an indirection mechanism: writing the DEV_OP register selects which internal register is to be accessed, and the DEV_DATA register can be read or written to access the selected register.

Figure 15-7 shows the format of the DEV_OP register. The DEV_DATA register reflects the format of the DEV register selected in DEV_OP.

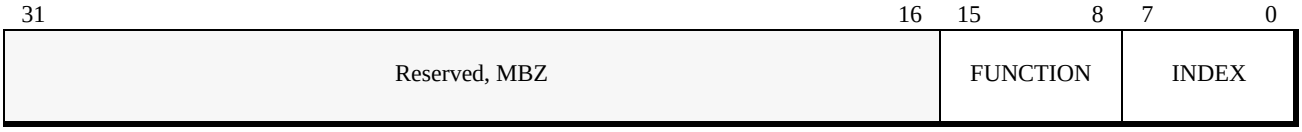


Figure 15-7. Format of DEV_OP Register (in PCI Config Space)

The FUNCTION field in the DEV_OP register selects the function/register to read or write according to the encoding in Table 15-17; for blocks of registers that have multiple instances (e.g., multiple DEV_BASE_HI/LO registers), the INDEX field selects the instance; otherwise it is ignored.

Table 15-17. Encoding of Function Field in DEV_OP Register

Function Code	RegisterType	Number of Instances
0	DEV_BASE_LO	multiple
1	DEV_BASE_HI	multiple
2	DEV_MAP	multiple
3	DEV_CAP	single
4	DEV_CR	single
5	DEV_ERR_STATUS	single
6	DEV_ERR_ADDR_LO	single
7	DEV_ERR_ADDR_HI	single

For example, to write the DEV_BASE_HI register for protection domain number 2, software sets DEV_OP.FUNCTION to 1, and DEV_OP.INDEX to 2, and then writes the desired 32-bit value into DEV_DATA. As the DEV_OP and DEV_DATA registers are accessed through PCI config space (ports 0CF8h–0CFFh), they may be secured from unauthorized access by software executing on the processor by appropriate settings in the SVM I/O protection bitmap. These registers are also protected by the host bridge from external access as described in “Config Space Accesses” on page 526.

15.24.6 DEV Control and Status Registers

The DEV control and status registers are accessible by means of the indirection mechanism; these registers are *not* directly visible in PCI config space.

DEV_CAP Register. Read-only register; holds implementation specific information: the number of protection domains supported, the number of DEV_MAP registers (which map device/unit IDs to domain numbers), and the revision ID.

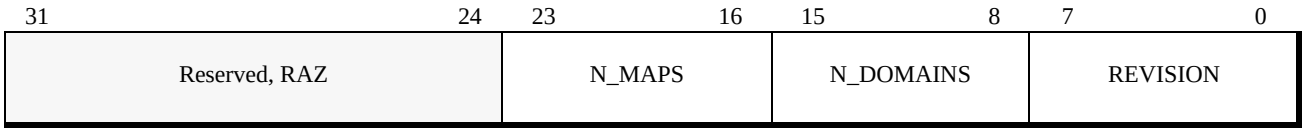


Figure 15-8. Format of DEV_CAP Register (in PCI Config Space)

The initial implementation provide four domains and three map registers.

DEV_CR Register. This is the main control register for the DEV mechanism; it is cleared to zero by RESET.

Table 15-18. DEV_CR Control Register

Bit(s)	Definition
31:7	Reserved, MBZ
6	DEV table walk probe disable. 0 = Use probe on DEV walk; 1 = Do not use probe
5	SL_DEV_EN. Enable bit for limited memory protection, see Section 15.24.8 on page 530. Set to “1” by SKINIT instruction, can be cleared by software.
4	Invalidate DEV cache. Software must set this bit to 1 to invalidate the DEV cache; cleared by hardware when invalidation is complete.
3	Enable MCE reporting. 0 = Do not generate MCE; 1 = Generate MCE on errors.
2	I/O space protection enable (IOSPEN) 0 = Allow upstream I/O cycles; 1 = Block.
1	Memory clear disable. If non-zero, memory-clearing on reset is disabled. This bit is not writable until the memory is enabled.
0	DEV global enable bit. If zero, DEV protection is turned off.

DEV_BASE Address/Limit Registers. The DEV base address registers (one set per domain) each point to the physical address of a DEV table corresponding to a protection domain. The address and size are encoded in a pair (high/low) of 32-bit registers. The N_DOMAINS field in DEV_CAP indicates how many (pairs of) DEV_BASE registers are implemented. The register format is as shown in Figures 15-9 and 15-10.

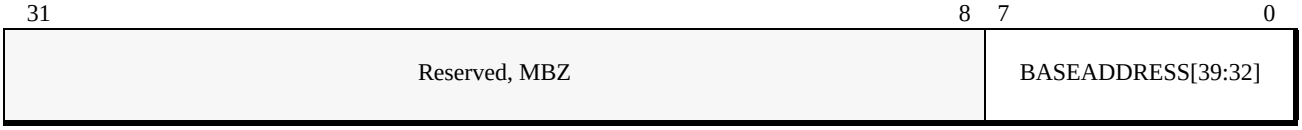


Figure 15-9. Format of DEV_BASE_HI[n] Registers

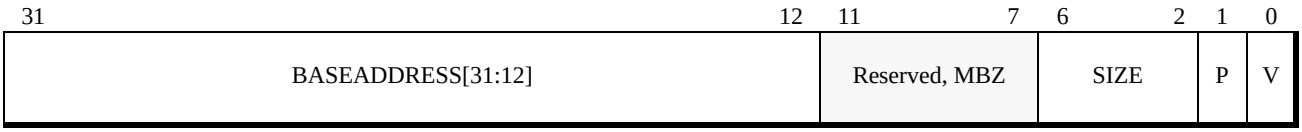


Figure 15-10. Format of DEV_BASE_LO[n] Registers

Fields of the DEV_BASE_HI and DEV_BASE_LO registers are defined as follows:

- *Valid (V)*—Bit 0. Indicates whether a DEV table has been defined for the given protection domain; if this bit is clear, software can leave the other fields undefined, and no protection checks are performed for memory references in this domain.

- *Protect (P)*—Bit 1. Indicates whether accesses to addresses beyond the address range covered by the DEV are legal (P=0) or illegal (P=1).
- *SIZE*—Bits 6:2. Specifies how much memory the DEV covers, expressed increments of 4GB * 2^{size}. In other words, a DEV table covers a minimum of 4GB, and can expand by powers of two.

DEV_MAP Registers. The DEV_MAP registers assign protection domain numbers to device-originated requests by matching the device ID (HT bus and unit number) associated with the request against bus and unit numbers in the registers. If no match is found in any of the registers, a domain number of zero is returned. The number of DEV_MAP registers implemented by the chip is indicated by the N_MAPS field in DEV_CAP.

The format of the DEV_MAP registers is shown in Figure 15-11.

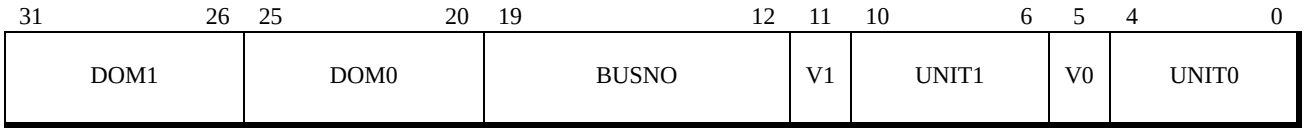


Figure 15-11. Format of DEV_MAP[n] Registers

The fields of the DEV_MAP[*n*] registers are defined as follows:

- UNIT0—Bits 4:0. Specifies the first of two HyperTransport link unit numbers on the bus number specified by the BUSNO field.
- V0—Bit 5. Indicates whether UNIT0 is valid (no matches occur on invalid entries).
- UNIT1—Bits 10:6. Specifies the second of two HyperTransport link unit numbers on the bus number specified by the BUSNO field.
- V1—Bit 11. Indicates whether UNIT1 is valid (no matches occur on invalid entries).
- BUSNO—Bits 19:12. Specifies a HyperTransport link bus number.
- DOM0—Bits 25:20. Specifies the protection domain for the first HyperTransport link unit.
- DOM1—Bits 31:26. Specifies the protection domain for the second HyperTransport link unit.

15.24.7 Unauthorized Access Logging

Any attempted unauthorized access by devices to DEV-protected memory is logged by the host bridge in the DEV_Error_Status and DEV_Error_Address registers for possible inspection by the VMM.

15.24.8 Secure Initialization Support

The host bridge contains additional logic that operates in conjunction with the SKINIT instruction to provide a limited form of memory protection during the secure startup protocol. This provides protection for a Secure Loader image in memory, allowing it to, among other things, set up full DEV protection. (See Section 15.27 for detailed operation of SKINIT.)

The host bridge logic includes a hidden (not accessible to software) SL_DEV_BASE address register. SL_DEV_BASE points to a 64KB-aligned 64KB region of physical memory. When SL_DEV_EN is 1, the 64KB region defined by SL_DEV_BASE is protected from external access (as if it were protected by the DEV), as well as from any access (both CPU and external accesses) via GART-translated addresses. Additionally, the SL_DEV mechanism, when enabled, blocks all device accesses to PCI Configuration space.

15.25 Nested Paging

The optional SVM nested paging feature provides for two levels of address translation, thus eliminating the need for the VMM to maintain shadow page tables.

15.25.1 Traditional Paging versus Nested Paging

Figure 15-12 shows how a page in the linear address space is mapped to a page in the physical address space in traditional (single-level) address translation. Control register CR3 contains the physical address of the base of the page tables (PT, represented by the shaded box in the figure), which governs the address translation.

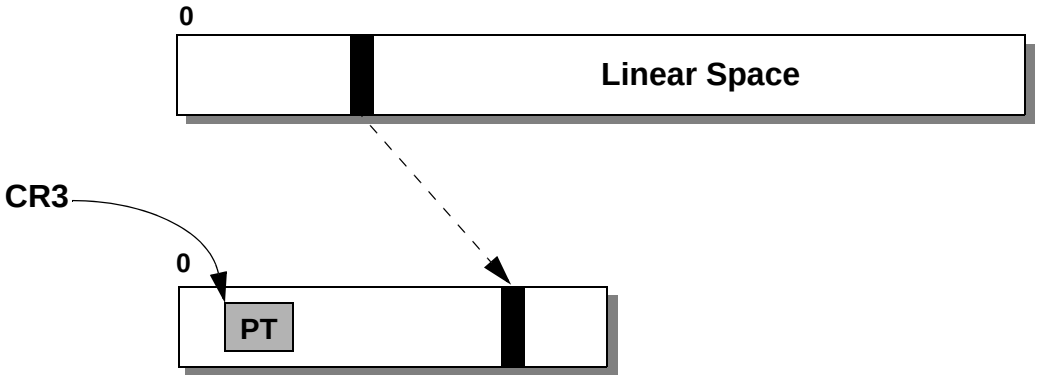


Figure 15-12. Address Translation with Traditional Paging

With nested paging enabled, *two* levels of address translation are applied; refer to Figure 15-13 below.

- Both guest and host levels have their own copy of CR3, referred to as gCR3 and nCR3, respectively.
- Guest page tables (gPT) map guest linear addresses to guest physical addresses. The guest page tables are in guest physical memory, and are pointed to by gCR3.
- Nested page tables (nPT) map guest physical addresses to system physical addresses. The nested page tables are in system physical memory, and are pointed to by nCR3.
- The most-recently used translations from guest linear to system physical address are cached in the TLB and used on subsequent guest accesses.

It is important to note that gCR3 and the guest page table entries contain guest physical addresses, not system physical addresses. Hence, before accessing a guest page table entry, the table walker first translates that entry's guest physical address into a system physical address.

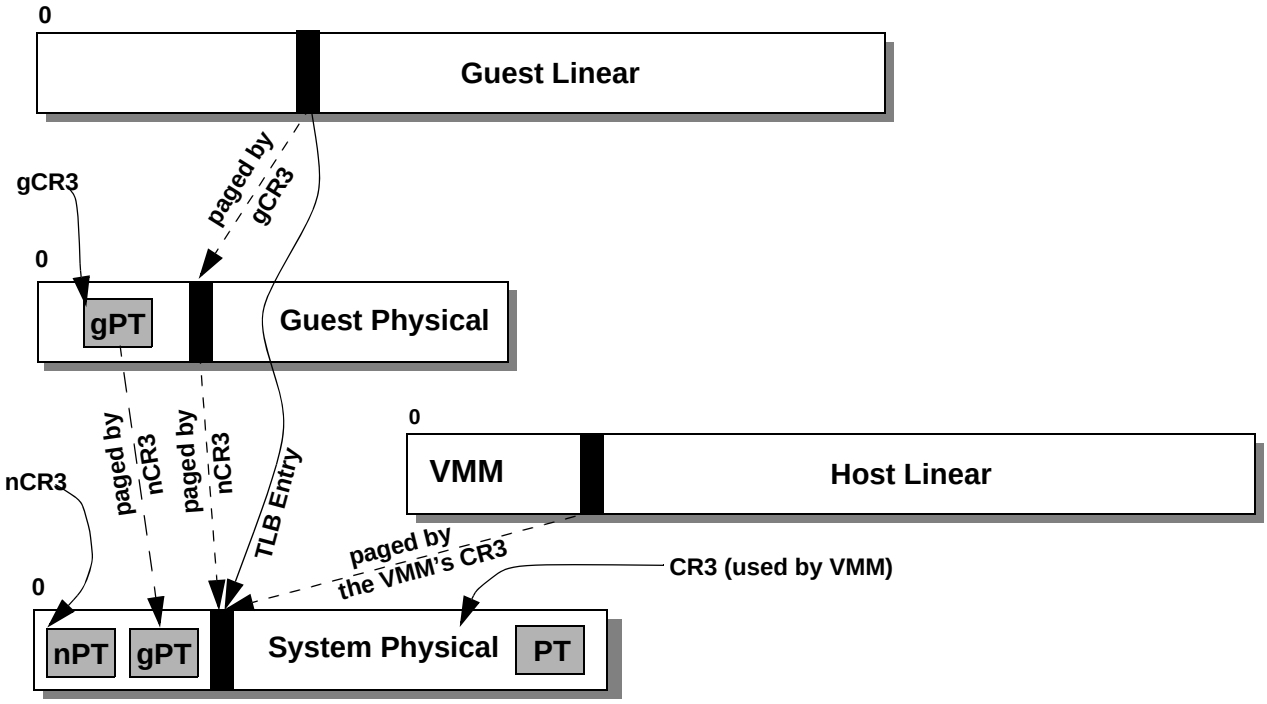


Figure 15-13. Address Translation with Nested Paging

The VMM can give each guest a different ASID, so that TLB entries from different guests can coexist in the TLB. The ASID value of zero is reserved for the host; if the VMM attempts to execute VMRUN with a guest ASID of zero, the result is #VMEXIT(VMEXIT_INVALID). Note that because an ASID is associated with the guest's physical address space, it is common across all of the guest's virtual address spaces within a processor. This differs from shadow page tables where ASIDs tag individual guest virtual address spaces. Note also that the same ASID may or may not be associated with the same address space across all processors in a multiprocessor system, for either nested tables or shadow tables; this depends on how the VMM manages ASID assignment.

15.25.2 Replicated State

Most processor state affecting paging is replicated for host and guest. This includes the paging registers CR0, CR3, CR4, EFER and PAT. CR2 is not replicated but is loaded by VMRUN. The MTRRs are not replicated.

While nested paging is enabled, all (guest) references to the state of the paging registers by x86 code (MOV to/from CR*n*, etc.) read and write the guest copy of the registers; the VMM's versions of the

registers are untouched and continue to control the second level translations from guest physical to system physical addresses. In contrast, when nested paging is disabled, the VMM's paging control registers are stored in the host state save area and the paging control registers from the guest VMCB are the only active versions of those registers.

15.25.3 Enabling Nested Paging

The VMRUN instruction enables nested paging when the NP_ENABLE bit in the VMCB is set to 1. The VMCB contains the hCR3 value for the page tables for the extra translation. The extra translation uses the same paging mode as the VMM used when it executed the most recent VMRUN.

Nested paging is automatically disabled by #VMEXIT.

Nested paging is allowed only if the host has paging enabled. Support for nested paging is indicated by CPUID Fn8000_000A_EDX[NP] = 1. If VMRUN is executed with hCR0.PG cleared to zero and NP_ENABLE set to 1, VMRUN terminates with #VMEXIT(VMEXIT_INVALID). See Section 3.3, “Processor Feature Identification,” on page 71 for more information on using the CPUID instruction.

15.25.4 Nested Paging and VMRUN/#VMEXIT

When VMRUN is executed with nested paging enabled (NP_ENABLE = 1), the paging registers are affected as follows:

- VMRUN saves the VMM’s CR3 in the host save area.
- VMRUN loads the guest paging state from the guest VMCB into the guest registers (i.e., VMRUN loads CR3 with the VMCB CR3 field, etc.). The guest PAT register is loaded from G_PAT field in the VMCB.
- VMRUN loads nCR3, the version of CR3 to be used while the nested-paging guest is running, from the N_CR3 field in the VMCB. The other host paging-control bits (hCR4.PAE, etc.) remain the same as they were in the VMM at the time VMRUN was executed.

When VMRUN is executed with nested paging enabled (NP_ENABLE = 1), the following conditions are considered illegal state combinations, in addition to those mentioned in “Canonicalization and Consistency Checks” on page 489:

- Any MBZ bit of nCR3 is set.
- Any G_PAT.PA field has an unsupported type encoding or any reserved field in G_PAT has a non-zero value. (See Section 7.8.1, “PAT Register,” on page 218.)

When #VMEXIT occurs with nested paging enabled:

- #VMEXIT writes the guest paging state (gCR3, gCR0, etc.) back into the VMCB. nCR3 is not saved back into the VMCB.
- #VMEXIT need not reload any host paging state other than CR3 from the host save area, though an implementation is free to do so.

15.25.5 Nested Table Walk

When the guest is running with nested paging enabled, a TLB miss causes several nested table walks:

- Guest Page Tables—the gCR3 register specifies a guest physical address, as do the entries in the guest's page tables. These guest physical addresses must be translated to system physical addresses using the nested page tables. Nested page table level faults can occur on these accesses, including write faults due to setting of accessed and dirty bits in the guest page table.
- Final Guest-Physical Page—once a guest linear to guest physical mapping is known, guest permissions can be checked. If the guest page tables allow the access, the guest physical address is walked in the nested page tables to find the system physical address.

Table walks for guest page tables are always treated as user writes at the nested page table level. For this reason,

- the page must be writable by user at the nested page table level, or else a #VMEXIT(NPF) is raised, and
- the dirty and accessed bits are always set in the nested page table entries that were touched during nested page table walks for guest page table entries.

A table walk for the guest page itself is always treated as a user access at the nested page table level, but is treated as a data read, data write, or code read, depending on the guest access.

If the guest has paging disabled (gCR0.PG = 0), there are no guest page table entries to be translated in the nested page tables. In this case, the final guest-physical address is equal to the guest-linear address, and is still translated in the nested page tables.

15.25.6 Nested versus Guest Page Faults, Fault Ordering

In nested paging, page faults can be raised at either the guest or nested page table level. Nested walks proceed in the following order; faults are generated in the same order:

1. Walk the guest page table entries in the nested page table. Dirty/Accessed bits are set as needed in the nested page table. Any nested page table faults result in #VMEXIT(NPF).
2. As the guest page table walk proceeds from the top of the page table to the last entry, any not-present entries or reserved bits in the guest page table entries at each level of the guest walk cause #PF in the guest. Guest dirty and accessed bits are set as needed in the guest page tables during the walk. Steps 1 and 2 are repeated for each level of the guest page table that is traversed.
3. Once the guest physical address for the guest access has been determined, check the guest permissions; any fault at this point causes a #PF in the guest.
4. Perform the final translation from guest physical to system physical using the nested page table; any fault during this translation results in a #VMEXIT(NPF).

Nested page faults are entirely a function of the nested page table and VMM processor mode. Nested faults cause a #VMEXIT(NPF) to the VMM. The faulting guest physical address is saved in the VMCB's EXITINFO2 field; EXITINFO1 delivers an error code similar to a #PF error code:

- Bit 0 (P)—cleared to 0 if the nested page was not present, 1 otherwise
- Bit 1 (RW)—set to 1 if the nested page table level access was a write. Note that host table walks for guest page tables are always treated as data writes.
- Bit 2 (US)—set to 1 if the nested page table level access was a user access. Note that nested page table accesses performed by the MMU are treated as user accesses unless there are features enabled that override this.
- Bit 3 (RSV)—set to 1 if reserved bits were set in the corresponding nested page table entry
- Bit 4 (ID)—set to 1 if the nested page table level access was a code read. Note that nested table walks for guest page tables are always treated as data writes, even if the access itself is a code read
- Bit 6 (SS) - set to 1 if the fault was caused by a shadow stack access.

In addition, the VMCB contents for nested page faults indicate whether the page fault was encountered during the nested page table walk for a guest page TLB entry, or for the final nested walk for the guest physical address, as indicated by EXITINFO1[33:32]:

- Bit 32—set to 1 if nested page fault occurred while translating the guest's final physical address
- Bit 33—set to 1 if nested page fault occurred while translating the guest page tables
- Bit 37—set to 1 if the page was marked as a supervisor shadow stack page in the leaf node of the nested page table and the shadow stack check feature is enabled in VMCB offset 90h.

Guest faults are entirely a function of the guest page tables and processor mode; they are delivered to the guest as normal #PF exceptions without any VMM intervention, unless the VMM is intercepting guest #PF exceptions. Bits 32 and 33 of EXITINFO1 are written during nested page faults to indicate whether the page fault was encountered during the nested page table walk for a guest page table's table entries, or if the fault was encountered during the nested page table walk for the translation of the final guest physical address.

The processor may provide additional instruction decode assist information. (See section 15.10.)

15.25.7 Combining Nested and Guest Attributes

Any access to guest physical memory is subjected to a permission check by examining the mapping of the guest physical address in the nested page table.

A page is considered writable by the guest only if it is marked writable at both the guest and nested page table levels. Note that the guest's gCR0.WP affects only the interpretation of the guest page table entry; setting gCR0.WP cannot make a page writable at any CPL in the guest, if the page is marked read-only in the nested page table. The host hCR0.WP bit is ignored under nested paging.

A page is considered executable by the guest only if it is marked executable at both the guest and nested page table levels. If the EFER.NXE bit is cleared for the guest, all guest pages are executable at the guest level. Similarly, if the EFER.NXE bit is cleared for the host, all nested page table mappings are executable at the underlying nested level.

Some attributes are taken from the guest page tables and operating modes only. A page is considered global within the guest only if is marked global in the guest page tables; the nested page table entry and host hCR4.PGE are irrelevant. Global pages are only global within their ASID.

A page is considered user in the guest only if it is marked as user at the guest level. The page must be marked user in the nested page table to allow any guest access at all.

15.25.8 Combining Memory Types, MTRRs

When nested paging is disabled, the processor behaves as though there is no gPAT register.

The host PAT MSR determines memory type attributes for the current VM, and guest writes to the PAT MSR that aren't intercepted by the VMM will alter the host PAT MSR. The hypervisor is responsible for context-switching the PAT MSR contents on world switches between VM's.

When nested paging is enabled, the processor combines guest and nested page table memory types. Registers that affect memory types include:

- The PCD/PWT/PAT*i* bits in the nested and guest page table entries.
- The PCD/PWT bits in the nested CR3 and guest CR3 registers.
- The guest PAT type (obtained by appropriately indexing the gPAT register).
- The host PAT type (obtained by appropriately indexing the host’s PAT register).
- The MTRRs (which are referenced based only on system physical address).
- gCR0.CD and hCR0.CD.

Note that there is no hardware support for guest MTRRs; the VMM can simulate their effect by altering the memory types in the nested page tables. Note that the MTRRs are only applied to system physical addresses.

The rules for combining memory types when constructing a guest TLB entry are:

- Nested and guest PAT types are combined according to Table 15-19, producing a “combined PAT type.”
- Combined PAT type is further combined with the MTRR type according to Table 15-20, where the relevant MTRRs are determined by the system physical address.
- Either gCR0.CD or hCR0.CD can disable caching.

Memory Consistency Issues. Because the guest uses extra fields to determine the memory type, the VMM may use a different memory type to access a given piece of memory than does the guest. If one access is cacheable and the other is not, the VMM and guest could observe different memory images, which is undesirable. (MP systems are particularly sensitive to this problem when the VMM desires to migrate a virtual processor from one physical processor to another.)

To address this issue, the following mechanisms are provided:

- VMRUN and #VMEXIT flush the write combiners. This ensures that all writes to WC memory by the guest are visible to the host (or vice-versa) regardless of memory type. (It does not ensure that cacheable writes by one agent are properly observed by WC reads or writes by the other agent.)
- A new memory type *WC+* is introduced. WC+ is an uncacheable memory type, and combines writes in write-combining buffers like WC. Unlike WC (but like the CD memory type), accesses to WC+ memory also snoop the caches on all processors (including self-snooping the caches of the processor issuing the request) to maintain coherency. This ensures that cacheable writes are observed by WC+ accesses.
- When combining nested and guest memory types that are incompatible with respect to caching, the WC+ memory type is used instead of WC (and Table 15-20 ensures that the snooping behavior is retained regardless of the host MTRR settings). Refer to Table 15-19 or details.

Table 15-19 shows how guest and host PAT types are combined into an effective PAT type. When interpreting this table, recall (a) that guest and host PAT types are not combined when nested paging is disabled and (b) that the intent is for the VMM to use its PAT type to simulate guest MTRRs.

Table 15-19. Combining Guest and Host PAT Types

		Host PAT Type					
		UC	**UC–**	**WC**	**WP**	**WT**	**WB**
Guest PAT Type	**UC**	UC	UC	UC	UC	UC	UC
	UC–	UC	UC–	WC	UC	UC	UC
	WC	WC	WC	WC	WC+	WC+	WC+
	WP	UC	UC	UC	WP	UC	WP
	WT	UC	UC	UC	UC	WT	WT
	WB	UC	UC	WC	WP	WT	WB

The existing AMD64 table that defines how PAT types are combined with the physical MTRRs is extended to handle CD and WC+ PAT types as shown in Table 15-20.

Table 15-20. Combining PAT and MTRR Types

		MTRR Type				
		UC	**WC**	**WP**	**WT**	**WB**
Effective PAT Type	**UC**	UC	CD	CD	CD	CD
	UC–	UC	WC	CD	CD	CD
	WC	WC	WC	WC	WC	WC
	WC+	WC	WC	WC+	WC+	WC+
	WP	UC	CD	WP	CD	WP
	WT	UC	CD	CD	WT	WT
	WB	UC	WC	WP	WT	WB

15.25.9 Page Splintering

When an address is mapped by guest and nested page table entries with different page sizes, the TLB entry that is created matches the size of the smaller page.

15.25.10 Legacy PAE Mode

The behavior of PAE mode in a nested-paging guest differs slightly from the behavior of (host-only) legacy PAE mode, in that the guest's four PDPEs are not loaded into the processor at the time CR3 is written. Instead, the PDPEs are accessed on demand as part of a table walk. This has the side-effect that illegal bit combinations in the PDPEs are not signaled at the time that CR3 is written, but instead when the faulty PDPE is accessed as part of a table walk.

This means that an operating system cannot rely on the behavior when the in-memory PDPEs are different than the in-processor copy.

15.25.11 A20 Masking

There is no provision for applying A20 masking to guest physical addresses; the VMM can emulate A20 masking by changing the nested page mappings accordingly.

15.25.12 Detecting Nested Paging Support

Nested Paging is an optional feature of SVM and is not available in all implementations of SVM-capable processors. The CPUID instruction should be used to detect nested paging support on a particular processor. See Section 3.3, "Processor Feature Identification," on page 71 for more information on using the CPUID instruction.

15.25.13 Guest Mode Execute Trap Extension

The Guest Mode Execute Trap (GMET) extension allows a hypervisor to cause nested page faults on attempts by a guest to execute code at CPL0, 1 or 2 from pages designated by the hypervisor. The presence of the GMET extension is indicated by CPUID Fn8000_000A EDX[17]=1. The GMET mode is selected for a targeted guest by setting bit 3 of VMCB offset 090h to 1. For processors that don't support GMET this bit is ignored.

On GMET capable processors, when this bit is set to 1 on a VMRUN, the processor changes how the U/S bit in the nested page table is interpreted. The NX bit still prohibits execution of code at any privilege level when set to 1. However, with GMET enabled and the effective NX bit =0, if the effective U/S bit =1 and the page is being accessed for execution at CPL0, 1 or 2, a nested page fault #VMEXIT(NPF) is generated. If the effective NX bit =0 and the effective U/S bit =0 then the

translation is allowed for the code page. The following table summarizes the behavior when GMET is enabled.

Table 15-21. GMET Page Configuration

nPT NX Bit	nPT U/S Bit	Guest User-Mode Code	Guest Supervisor-Mode Code
1	X	No Execute	No Execute
0	1	Execute	No Execute
0	0	Execute	Execute

The EXITINFO1 field for the nested page fault contains the page fault error code describing attributes of the attempted translation that caused the fault. A GMET violation is not explicitly indicated with a separate bit. It is up to software to determine if it was NX based or GMET based by inspecting this error code along with the faulting page's effective NX and U/S settings.[1]

15.25.14 Supervisor Shadow Stacks

The Supervisor Shadow Stack (SSS) feature is an extension to nested paging which allows a hypervisor to restrict which guest physical addresses may be used for a guest supervisor shadow stack. Supervisor shadow stack accesses made by the guest to pages not designated as SSS pages in the nested page tables result in a #VMEXIT(NPF).

Determining Support for SSS. Support for the SSS feature is indicated CPUID Fn8008_000A_EDX[19](SupervisorShadowStack)=1.

Enabling SSS. The SSS feature is enabled by setting bit 4 in VMCB offset 90h (See Table B-1. VMCB Layout, Control Area). The SSS feature can be enabled only if nested paging is enabled in the VMCB and the PAE and No Execute paging modes (EFER.NXE=1) are enabled in the host.

- Attempting to execute a VMRUN with SSS enabled and nested paging disable result in a VMEXIT(INVALID).
- If the host is not legacy non-PAE mode or EFER.NXE=0, attempts to enable the SSS feature are silently ignored.

The SSS feature can be enabled regardless of whether the guest has enabled shadow stacks or not.

Designating SSS Pages. When the SSS feature is enabled, the hypervisor indicates a page may be used for a supervisor shadow stack using following combination of nested page table bits:

- NX=1 and U/S=0 in the final nested page table entry used to translate the address.
- R/W=1 in all other nested non-leaf page table entries leading to the final nested page table entry.

Although is not enforced by the SSS feature, R/W should be 0 in the final nested page table entry in order to achieve the desired security functionality.

[1] The guest user/supervisor indication is normally provided in ExitInfo1, however on some implementations a GMET erratum may require CPL to be read from the guest VMCB.

SSS Access Checking. When the SSS feature is enabled, guest supervisor shadow stack accesses are allowed only to physical pages designated as SSS pages in the nested page tables. Note that supervisor shadow stack writes to SSS pages are allowed to complete even though R/W=0 in the final nested page table entry.

The following accesses to SSS pages are not allowed:

- Supervisor shadow accesses made to non-SSS pages. These result in a #VMEXIT(NPF) with the SS bit set in the EXITINFO1 error code.
- Attempting to execute code from an SSS page. This results in a VMEXIT(#NPF) the same as any page with NX=1.

See Chapter 15.25.6, “Nested versus Guest Page Faults, Fault Ordering,” on page 534 for more information on EXITINFO1 error codes for nested page faults.

15.26 Security

SVM provides additional hardware support that is designed to facilitate the construction of trusted software systems. While the security features described in this section are orthogonal to SVM’s virtualization support (and are not required for processor virtualization), the two form building blocks for trusted systems.

SKINIT Instruction. The SKINIT instruction and associated system support (the Trusted Platform Module or TPM) are designed to allow for verifiable startup of trusted software (such as a VMM), based on secure hash comparison.

Security Exception. A security exception (#SX) is used to signal certain security-critical events.

15.27 Secure Startup with SKINIT

The SKINIT instruction is one of the keys to creating a “root of trust” starting with an initially untrusted operating mode. SKINIT reinitializes the processor to establish a secure execution environment for a software component called the secure loader (SL) and starts execution of the SL in a way that cannot be tampered with. SKINIT also copies the secure loader executable image to an external device, such as a Trusted Platform Module (TPM) for verification using unique bus transactions that preclude SKINIT operation from being emulated by software in a way that the TPM could not readily detect. (Detailed operation is described in Section 15.27.4.)

15.27.1 Secure Loader

A secure loader (SL) typically initializes SVM hardware mechanisms and related data structures, and initiates execution of a trusted piece of software such as a VMM (referred to as a Security Kernel, or SK, in this document), after first having validated the identity of that software.

SKINIT allows SVM protections to be reliably enabled after the system is already up and running in a non-trusted mode — there is no requirement to change the typical x86 platform boot process.

Exact details of the handoff from the SL to an SK are dependent on characteristics of the SL, SK and the initial untrusted operating environment. However, there are specific requirements for the SL image, as described in Section 15.27.2.

15.27.2 Secure Loader Image

The secure loader (SL) image contains all code and initialized data sections of a secure loader. This code and initial data are used to initialize and start a security kernel in a completely safe manner, including setting up DEV protection for memory allocated for use by SL and SK. The SL image is loaded into a region of memory called the secure loader block (SLB) and can be no larger than 64Kbyte (see Section 15.27.3). The SL image is defined to start at byte offset 0 in the SLB.

The first word (16 bits) of the SL image must specify the SL entry point as an unsigned offset into the SL image. The second word must contain the length of the image in bytes; the maximum length allowed is 65535 bytes. These two values are used by the SKINIT instruction. The layout of the rest of the image is determined by software conventions. The image typically includes a digital signature for validation purposes. The digital signature hash must include the entry point and length fields. SKINIT transfers the SL image to the TPM for validation prior to starting SL execution (see Section 15.27.6 for further details of this transfer). The SL image for which the hash is computed must be ready to execute without prior manipulation.

15.27.3 Secure Loader Block

The secure loader block is a 64Kbyte range of physical memory which may be located at any 64Kbyte-aligned address below 4Gbyte. The SL image must have been loaded into the SLB starting at offset 0 before executing SKINIT. The physical address of the SLB is provided as an input operand (in the EAX register) to SKINIT, which sets up special protection for the SLB against device accesses (i.e., the DEV need not be activated yet).

The SL must be written to execute initially in flat 32-bit protected mode with paging disabled. A base address can be derived from the value in EAX to access data areas within the SL image using base+displacement addressing, to make the SL code position-independent.

Memory between the end of the SL image and the end of the SLB may be used immediately upon entry by the SL as secure scratch space, such as for an initial stack, before DEV protections are set up for the rest of memory. The amount of space required for this will limit the maximum size of the SL image, and will depend on SL implementation. SKINIT sets the ESP register to the appropriate top-of-stack value (EAX + 10000h).

Figure 15-14 illustrates the layout of the SLB, showing where EAX and ESP point after SKINIT execution. Labels in italics indicate suggested uses; other labels reflect required items.

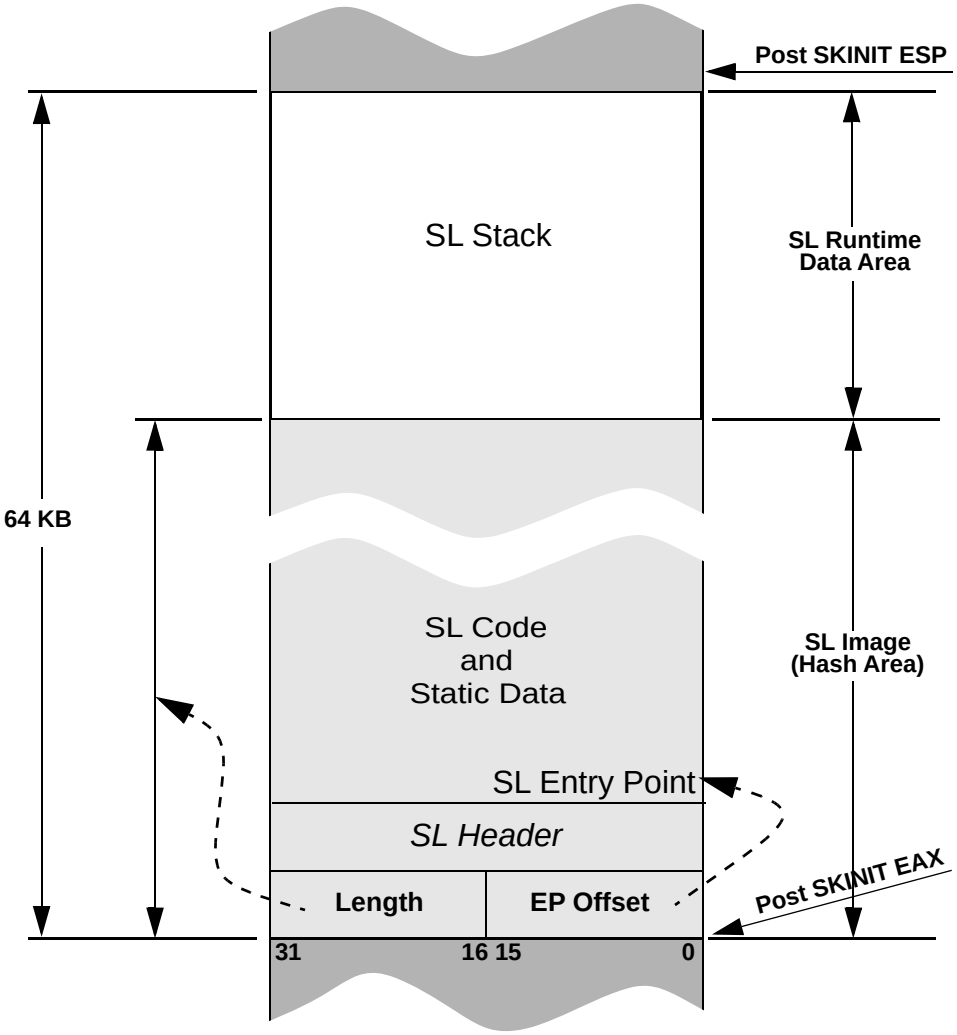


Figure 15-14. SLB Example Layout

15.27.4 Trusted Platform Module

The trusted platform module, or TPM, is an essential part of full trusted system initialization. This device is attached to an LPC link off the system I/O hub. It recognizes special SKINIT transactions, receives the SL image sent by SKINIT and verifies the signature. Based on the outcome, the device decides whether or not to cooperate with the SL or subsequent SK. The TPM typically contains sealed storage containing cryptographic keys and other high-security information that may be specific to the platform.

15.27.5 System Interface, Memory Controller and I/O Hub Logic

SKINIT uses special support logic in the processor's system interface unit, the internal controller and the I/O hub to which the TPM is attached. SKINIT uses special transactions that are unique to SKINIT, along with this support logic, designed to securely transmit the SL Image to the TPM for validation.

The use of this special protocol is intended to allow the TPM to detect true execution, as opposed to emulation, of a trusted Secure Loader, which in turn provides a means for verifying the subsequent loading and startup of a trusted Security Kernel.

15.27.6 SKINIT Operation

The SKINIT instruction is intended to be used primarily in normal mode prior to the VMM taking control.

SKINIT takes the physical base address of the SLB as its only input operand in EAX, and performs the following steps:

1. Reinitialize processor state in the same manner as for the INIT signal, then enter flat 32-bit protected mode with paging off. The CS selector is set to 8h and CS is read only. The SS selector is set to 10h and SS is read/write and expand-up. The CS and SS bases are cleared to 0 and limits are set to 4G. DS, ES, FS and GS are left as 16-bit real mode segments and the SL must reload these with protected mode selectors having appropriate GDT entries before using them. Initialized data in the SLB may be referenced using the SS segment override prefix until DS is reloaded. The general purpose registers are cleared except for EAX, which points to the start of the secure loader, EDX, which contains model, family and stepping information, and ESP, which contains the initial stack pointer for the secure loader. Cache contents remain intact, as do the x87 and SSE control registers. Most MSRs also retain their values, except those which might compromise SVM protections. The EFER MSR, however, is cleared. The DPD, R_INIT and DIS_A20M flags in the VM_CR register are unconditionally set to 1.
2. Form the SLB base address by clearing bits 15:0 of EAX (EAX is updated), and enable the SL_DEV protection mechanism (see Section 15.24.8) to protect the 64-Kbyte region of physical memory starting at the SLB base address from any device access.
3. In multiprocessor operation, perform an interprocessor handshake as described in section 15.27.8.
4. Read the SL image from memory and transmit it to the TPM in a manner that cannot be emulated by software.
5. Signal the TPM to complete the hash and verify the signature. If any failures have occurred along the way, the TPM will conclude that no valid SL was started.
6. Clear the Global Interrupt Flag. This disables all interrupts, including NMI, SMI and INIT and ensures that the subsequent code can execute atomically. If the processor enters the shutdown state (due to a triple fault for instance) while GIF is clear, it can only be restarted by means of a RESET.
7. Update the ESP register to point to the first byte beyond the end of the SLB (SLB base + 65536), so that the first item pushed onto the stack by the SL will be at the top of the SLB.

8. Add the unsigned 16-bit entry point offset value from the SLB to the SLB base address to form the SL entry point address, and jump to it.

The validation of the SL image by the TPM is a one-way transaction as far as SKINIT is concerned. It does not depend on any response from the TPM after transferring the SL image before jumping to the SL entry point, and initiates execution of the Secure Loader unconditionally. Because of the processor initialization performed, SKINIT does not honor instruction or data breakpoint traps, or trace traps due to EFLAGS.TF.

Pending interrupts. Device interrupts that may be pending prior to SKINIT execution due to EFLAGS.IF being clear, or that assert during the execution of SKINIT, will be held pending until software subsequently sets GIF to 1. Similarly, SMI, INIT and NMI interrupts that assert after the start of SKINIT execution will also be held pending until GIF is set to 1.

Debug Considerations. SKINIT automatically disables various implementation-specific hardware debug features. A debug version of the SL can reenable those features by clearing the VM_CR.DPD flag immediately upon entry.

15.27.7 SL Abort

If the SL determines that it cannot properly initialize a valid SK, it must cause GIF to be set to 1 and clear the VM_CR MSR to re-enable normal processor operation.

15.27.8 Secure Multiprocessor Initialization

The following standard APIC features are used for secure MP initialization:

- The concept of a single Bootstrap Processor (BSP) and multiple Application Processors (APs).
- The INIT interprocessor interrupt (IPI), which puts the target processors into a halted state (INIT state) which is responsive only to a subsequent Startup IPI.
- The Startup IPI causes target processors to begin execution at a location in memory that is specified by the Boot Processor and conveyed along with the Startup IPI. The operation of the processor in response to a Startup IPI is slightly modified to support secure initialization, as described below.

A Startup IPI normally causes an AP to start execution at a location provided by the IPI. To support secure MP startup, each AP responds to a startup IPI by additionally clearing its GIF and setting the DPD, R_INIT and DIS_A20M flags in the VM_CR register if, and only if, the BSP has indicated that it has executed an SKINIT. All other aspects of Startup IPI behavior remain unchanged.

Software Requirements for Secure MP initialization. The driver that starts the SL must execute on the BSP. Prior to executing the SKINIT instruction, the driver must save any processor-specific system register contents to memory for restoration after reinitialization of the APs. The driver should also put all APs in an idle state. The driver must first confirmed that all APs are idle and then it must issue an INIT IPI to all APs and wait for its local APIC busy indication to clear. This places the APs into a halted state which is responsive only to a subsequent Startup IPI. APs will still respond to snoops for cache coherency. The driver may execute SKINIT at any time after this point. Depending on processor

implementation, a fixed delay of no more than 1000 processor cycles may be necessary before executing SKINIT to ensure reliable sensing of APIC INIT state by the SKINIT.

AP Startup Sequence. While the SL starts executing on the BSP, the APs remain halted in APIC INIT state. Either the SL or the SK may issue the Startup IPI for the APs at whatever point is deemed appropriate. The Startup IPI conveys an 8-bit vector specified by the software that issues the IPI to the APs. This vector provides the upper 8 bits of a 20-bit physical address. Therefore, the AP startup code must reside in the lower 1Mbyte of physical memory—with the entry point at offset 0 on that particular page.

In response to the Startup IPI, the APs start executing at the specified location in 16-bit real mode. This AP startup code must set up protections on each processor as determined by the SL or SK. It must also set GIF to re-enable interrupts, and restore the pre-SKINIT system context (as directed by the SL or SK executing on the BSP), before resuming normal system operation.

The SL must ensure the integrity of the AP startup sequence, for example by including the startup code in the hashed SL image and setting up DEV protection for it before copying it to the desired area. The AP startup code does not need to (and should not) execute SKINIT. Care must also be taken to avoid issuing another INIT IPI from any processor after the BSP executes SKINIT and before all APs have received a Startup IPI, as this could compromise the integrity of AP initialization.

Pending interrupts. Device interrupts that may be pending on an AP prior to the APIC INIT IPI due to EFLAGS.IF being clear, or that assert any time after the processor has accepted the INIT IPI, will be held pending through the subsequent Startup IPI, and remain pending until software sets GIF to 1 on that AP. Similarly, SMI, INIT, and NMI interrupts that assert after the processor has accepted the INIT IPI will also be held pending until GIF is set to 1.

Aborting MP initialization. In the event that the SL or SK on the BSP decides to abort SVM system initialization for any reason, the following clean-up actions must be performed by SL code executing on each processor before returning control to the original operating environment:

- The BSP and all APs that responded to the Startup IPI must restore GIF and clear VM_CR on each processor for normal operation.
- For each processor that has a distinct memory controller associated with it, the SL_DEV_EN flag in the DEV control register must be cleared in order to restore normal device accessibility to the 64KB SL memory range.

Any secure context created by the SL that should not be exposed to untrusted code should be cleaned up as appropriate before these steps are taken.

15.28 Security Exception (#SX)

The Security Exception fault signals security-sensitive events that occur while executing the VMM, in the form of an exception so that the VMM may take appropriate action. (A VMM would typically intercept comparable sensitive events in the guest.) Currently, the only use of the #SX is to redirect external INITs into an exception so that the VMM may — among other possibilities — destroy

sensitive information before re-issuing the INIT, this time without redirection. The INIT redirection is controlled by the VM_CR.R_INIT bit. (See “INIT Support” on page 520 for more details on INIT and #SX behavior). Note that INIT is gated by the Global Interrupt Flag (GIF), and so will be held pending if asserted while GIF is 0. When GIF transitions to 1, INIT will either take effect or be redirected to #SX, depending on the state of R_INIT.

The #SX exception dispatches to vector 30, and behaves like other fault-class exceptions such as General Protection Fault (#GP). The #SX exception pushes an error code. The only error code currently defined is 1, and indicates redirection of INIT has occurred.

The #SX exception is a contributory fault.

15.29 Advanced Virtual Interrupt Controller

The AMD Advanced Virtual Interrupt Controller (AVIC) is an important enhancement to AMD Virtualization™ Technology (AMD-V). In a virtualized environment, AVIC presents to each guest a virtual interrupt controller that is compliant with the local Advanced Programmable Interrupt Controller (APIC) architecture. See Chapter 16, “Advanced Programmable Interrupt Controller (APIC),” on page 601 for a detailed description of APIC.

15.29.1 Introduction

In a virtualized computer system, each guest operating system needs access to an interrupt controller to send and receive device and interprocessor interrupts. When there is no hardware acceleration, it falls to the virtual machine monitor (VMM) to intercept guest-initiated attempts to access the interrupt controller registers and provide direct emulation of the controller system programming interface allowing the guest to initiate and process interrupts. The VMM uses the underlying physical and virtual interrupt delivery mechanisms of the system to deliver interrupts from I/O devices and virtual processors to the target guest virtual processor and to handle any required end of interrupt processing.

Given the high rate of device and interprocessor interrupt generation in certain scenarios, in particular on server-class systems, the emulation of a local APIC can be a significant burden for the VMM. The AVIC architecture addresses the overhead of guest interrupt processing in a virtualized environment by applying hardware acceleration to the following components of interrupt processing:

- Providing a guest operating system access to performance-critical interrupt controller registers
- Initiating intra- and inter-processor interrupts (IPIs) in and between virtual processors in a guest

Software-initiated Interrupts. Modern operating systems use software interrupts (self-IPIs) to implement software event signalling, inter-process communication and the scheduling of deferred processing. System software sets up and initiates these interrupts by writing to control registers of the local APIC. AVIC hardware reduces VMM overhead by providing hardware assist for many of these operations.

Inter-processor Interrupts. Inter-processor interrupts (IPIs) are used extensively by modern operating systems to handle communication between processor cores within a machine (or, in a

virtualized environment, between virtual processors within a virtual machine). IPIs are also employed to provide signaling and synchronization for operations such as cross-processor TLB invalidations (also known as TLB shootdowns). AVIC provides hardware mechanisms that deliver the interrupt to the virtual interrupt controller of the target virtual processor without VMM intervention.

Device Interrupts. Acceleration of the delivery of virtual interrupts from I/O devices to virtual processors is not addressed directly by AVIC hardware. This acceleration would be provided by an I/O memory management unit (IOMMU). The AVIC architecture is compatible with the AMD I/O Memory Management Unit (IOMMU). For more information on the IOMMU architecture, see *AMD I/O Virtualization Technology (IOMMU) Specification* (order #48882). See “Device Interrupts” on page 561 for further details of device interrupt handling under the AVIC extension.

The following subsections describe the AVIC architecture in detail.

15.29.2 Local APIC Register Virtualization

The system programming interface for the local APIC comprises a set of memory-mapped registers. In a non-virtualized environment, system software directly reads and writes these registers to configure the interrupt controller and initiate and process interrupts. In a virtualized environment, each guest operating system still requires access to this system programming interface but does not own the underlying interrupt processing hardware. To provide this facility to the guest operating system, VMM-level software emulates the local APIC for each guest virtual processor.

The AVIC architecture provides an image of the local APIC called the guest virtual APIC (guest vAPIC) in the guest physical address (GPA) space of each virtual processor when the virtual machine for the guest is instantiated. This image is backed by a page in the system physical address (SPA) space called a vAPIC backing page. The backing page remains pinned in system memory as long as the virtual machine persists, even when the specific virtual processor associated with the backing page is not running. Accesses to the memory-mapped register set by the guest are redirected by AVIC hardware to this backing page.

The VMM reads configuration, control, and command information written by the guest from the backing page and writes status information to this page for the guest to read. The guest is allowed to read most registers directly without the need for VMM intervention. Most writes are intercepted allowing the VMM to process and act on the configuration, control, and command data from the guest. However, for certain frequently used command and control operations, specific hardware support allows the guest to directly initiate interrupts and complete end of interrupt processing, eliminating the need for VMM intervention in the execution of performance-critical operations.

15.29.3 AVIC Backing Page

AVIC hardware detects attempted accesses by the guest to its local APIC register set and redirects these accesses to the vAPIC backing page. This is illustrated in the figure below.

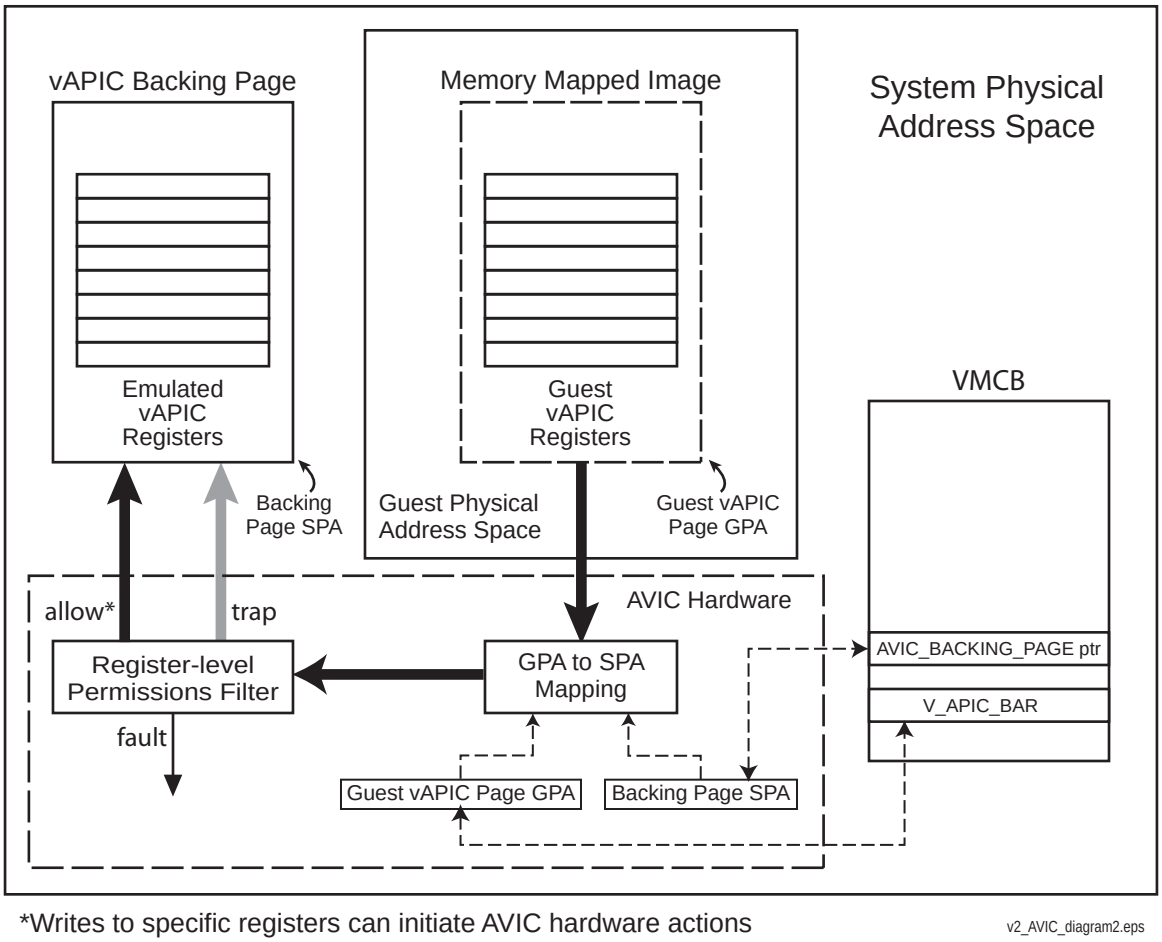


Figure 15-15. vAPIC Backing Page Access

To correctly redirect guest accesses of the guest vAPIC registers to the vAPIC backing page, the hardware needs two addresses. These are:

- vAPIC backing page address in the SPA space
- Guest vAPIC base address (APIC BAR) in the GPA space

System software is responsible for setting up a translation in the nested page table granting guest read and write permissions for accesses to the vAPIC Backing Page in SPA space. AVIC hardware walks the nested page table to check permissions, but does not use the SPA address specified in the leaf page table entry. Instead, AVIC hardware finds this address in the AVIC_BACKING_PAGE pointer field of the VMCB.

The VMM initializes the backing page with appropriate default APIC register values including items such as APIC version number. The vAPIC backing page address and the guest vAPIC base address are stored in the VMCB fields AVIC_BACKING_PAGE pointer and V_APIC_BAR respectively.

System firmware initializes the value of guest vAPIC base address (and VMCB.V_APIC_BAR) to FEE0_0000h. This is the address where the guest operating system expects to find the local APIC register set when it boots. If the guest attempts to relocate the local APIC register base address in GPA space by writing to the APIC Base Address Register (MSR 0000_001Bh), the VMM should intercept the write to update the V_APIC_BAR field of the guest's VMCB(s) and the GPA part of translation in the virtual machine's nested page tables.

The vAPIC backing page must be present in system physical memory for the life of the guest VM because some fields are updated even when the guest is not running.

15.29.3.1 Virtual APIC Register Accesses

AVIC hardware detects attempted guest accesses to the vAPIC registers in the backing page. These attempted accesses are handled by the register-level permissions filter in one of three ways:

- Allow—The access to the backing page is allowed to complete. Writes update the backing page value, while reads return the current value. In certain cases, a write results in specific hardware-based acceleration actions (summarized in Table 15-22 and described below).
- Fault—The processor performs an SVM intercept before the access. Causes a #VMEXIT.
- Trap— The processor performs an SVM intercept immediately after the access completes. Causes a #VMEXIT.

The details of this behavior for each of these registers are summarized in the following table:

Table 15-22. Guest vAPIC Register Access Behavior

Offset	Register Name	Result
20h	APIC ID Register	Read: Allowed Write: #VMEXIT (trap)
30h	APIC Version Register	Read: Allowed Write: #VMEXIT (fault)
80h	Task Priority Register (TPR)	Read: Allowed Write: Accelerated by AVIC
90h	Arbitration Priority Register (APR)	Read: #VMEXIT (fault) Write: #VMEXIT (fault)
A0h	Processor Priority Register (PPR)	Read: Allowed Write: #VMEXIT (fault)
B0h	End of Interrupt Register (EOI)	Read: Allowed Write: Accelerated by AVIC for edge-triggered interrupts or #VMEXIT (trap) for level triggered interrupts
C0h	Remote Read Register	Read: Allowed Write: #VMEXIT (trap)
D0h	Logical Destination Register	Read: Allowed Write: #VMEXIT (trap)
E0h	Destination Format Register	Read: Allowed Write: #VMEXIT (trap)

Table 15-22. Guest vAPIC Register Access Behavior (continued)

Offset	Register Name	Result
F0h	Spurious Interrupt Vector Register	Read: Allowed Write: #VMEXIT (trap)
100h–170h	In-Service Register (ISR)	Read: Allowed Write: #VMEXIT (fault)
180h–1F0h	Trigger Mode Register (TMR)	Read: Allowed Write: #VMEXIT (fault)
200h–270h	Interrupt Request Register (IRR)	Read: Allowed Write: #VMEXIT (fault)
280h	Error Status Register (ESR)	Read: Allowed Write: #VMEXIT (trap)
300h	Interrupt Command Register Low (bits 31:0)	Read: Allowed Write: Accelerated by AVIC or #VMEXIT (trap) for advanced functions.
310h	Interrupt Command Register High (bits 63:32)	Read: Allowed Write: Allowed
320h	Timer Local Vector Table Entry	Read: Allowed Write: #VMEXIT (trap)
330h	Thermal Local Vector Table Entry	Read: Allowed Write: #VMEXIT (trap)
340h	Performance Counter Local Vector Table Entry	Read: Allowed Write: #VMEXIT (trap)
350h	Local Interrupt 0 Vector Table Entry	Read: Allowed Write: #VMEXIT (trap)
360h	Local Interrupt 1 Vector Table Entry	Read: Allowed Write: #VMEXIT (trap)
370h	Error Vector Table Entry	Read: Allowed Write: #VMEXIT (trap)
380h	Timer Initial Count Register	Read: Allowed Write: #VMEXIT (trap)
390h	Timer Current Count Register	Read: #VMEXIT (fault) Write: #VMEXIT (fault)
3E0h	Timer Divide Configuration Register	Read: Allowed Write: #VMEXIT (trap)
400h–FFFh	Extended Registers	Read: #VMEXIT (fault) Write: #VMEXIT (fault)

Accesses to any other register locations not explicitly defined in this table are allowed to read and write the backing page.

All vAPIC registers are 32-bits wide and are located at 16-byte aligned offsets. The results of an attempted read or write of any bytes in the range [register_offset + 4:register_offset + 15] are undefined.

Guest writes to the Task Priority Register (TPR) and specific usage cases of writes to the End of Interrupt (EOI) Register and the Interrupt Command Register Low (ICRL) cause specific hardware actions. AVIC hardware allows guest writes to the Interrupt Command Register High (ICRH) since the writing of this register has no immediate hardware side-effect. AVIC hardware maintains and uses the value in the Processor Priority Register (PPR) to control the delivery of interrupts to guest virtual processors. The following sections discuss the handling of accesses by the guest to these registers in the vAPIC backing page.

Task Priority Register (TPR). When the guest operating system writes to the TPR, the value is updated in the backing page and the upper 4 bits of the value are automatically copied by the hardware to the V_TPR value in the VMCB. All reads from the TPR location return the value from the vAPIC backing page. Also, any TPR accesses using the MOV CR8 semantics update the backing page and V_TPR values.

The priority value stored in CR8 and V_TPR are not the same format as the APIC TPR register. Only the Task Priority bits of are maintained in the lower 4 bits of CR8 and V_TPR. The Task Priority Sub-class value is not stored. Writes to the memory-mapped TPR register update bits 3:0 of CR8 and V_TPR and writes to CR8 update the TPR backing page value bits 7:4 while bits 3:0 are set to zero.

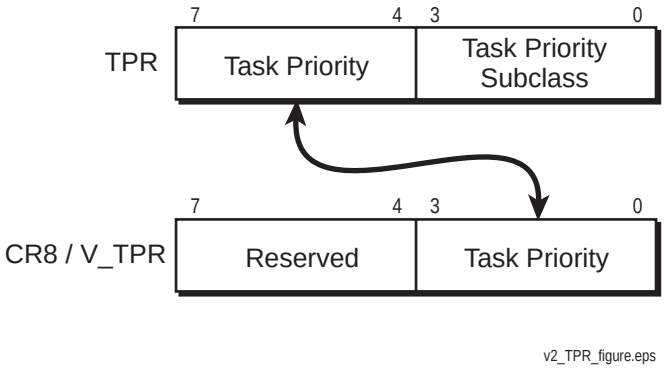


Figure 15-16. Virtual APIC Task Priority Register Synchronization

The synchronization between the Task Priority field of the TPR and the Task Priority field of CR8 is normal local APIC behavior which is emulated by AVIC. For more information on the APIC, see Chapter 16, “Advanced Programmable Interrupt Controller (APIC),” on page 601.

Processor Priority Register (PPR). Writes to the processor priority register by the guest cause a #VMEXIT without updating the value in the backing page. AVIC hardware maintains the PPR value in the backing page. AVIC hardware updates the PPR value in the backing page when either the TPR value or the highest in-service interrupt changes. This value is used to control the delivery of virtual interrupts to the guest. PPR reads by the guest are allowed.

End of Interrupt (EOI) Register. When the guest writes to the EOI register address, AVIC hardware clears the highest priority in-service interrupt (ISR) bit in the backing page and re-evaluates the interrupt state to determine if another pending interrupt should be delivered. If the highest priority in-service interrupt is set to level mode (in the corresponding TMR bit), the EOI write causes a #VMEXIT to allow the VMM to emulate the level-triggered behavior.

Interrupt Command Register Low (ICRL). Writes to the ICRL register have the side-effect of initiating the generation of an interprocessor interrupt (IPI) based on the values written to the fields in both the ICRL and ICRH registers. AVIC hardware handles the generation of IPIs when the specified Message Type is Fixed (also known as fixed delivery mode) and the Trigger Mode is edge-triggered. The hardware also supports self and broadcast delivery modes specified via the Destination Shorthand (DSH) field of the ICRL. Logical and physical APIC ID formats are supported. All other IPI types cause a #VMEXIT. For more information on AVIC's handling of IPI commands, see "Inter-processor Interrupts" on page 546.

15.29.4 VMCB Changes in Support of AVIC

The following paragraphs provide an overview of new VMCB fields defined as part of the AVIC architecture.

15.29.4.1 VMCB Control Word

AVIC adds the AVIC Enable bit to the VMCB control word at offset 60h:

Table 15-23. Virtual Interrupt Control (VMCB offset 60h)

VMCB offset	Bit(s)	Field Name	Description
060h	7:0	V_TPR	Virtual TPR for the guest [1] [2]
	8	V_IRQ	If nonzero; virtual INTR is pending [2] [3]
	15:9	—	Reserved, SBZ
	19:16	V_INTR_PRIO	Priority for virtual interrupt[3]
	20	V_IGN_TPR	If nonzero, the current virtual interrupt ignores the virtual TPR[3]
	23:21	—	Reserved, SBZ
	24	V_INTR_MASKING	Virtualized masking of INTR interrupts
	25	—	Reserved, SBZ
	31	AVIC Enable	If set, enables AVIC
	39:32	V_INTR_VECTOR	Vector to use for this interrupt[3]
	63:40	—	Reserved, SBZ

Note(s):
1. *Bits 3:0 are used for the 4-bit virtual TPR value; bits 7:4 are Reserved, SBZ.*
2. *This value is written back to the VMCB at #VMEXIT.*
3. *This field is ignored on VMRUN when AVIC is enabled.*

AVIC Enable—Virtual Interrupt Control, Bit 31. The AVIC hardware support may be enabled on a per virtual processor basis. This bit determines whether or not AVIC is enabled for a particular virtual processor. Any guest configured to use AVIC must also enable RVI (nested paging). Enabling AVIC implicitly disables the V_IRQ, V_INTR_PRIO, V_IGN_TPR, and V_INTR_VECTOR fields in the VMCB Control Word. Enabling AVIC also affects CR8 behavior independent of V_INTR_MASKING enable (bit 24): writes to CR8 affect the V_TPR and update the backing page and reads from CR8 return V_TPR.

15.29.4.2 Newly Defined VMCB Fields

AVIC utilizes a number of formerly reserved locations in the VMCB. Table 15-24 below lists the new fields defined by the architecture:

Table 15-24. New VMCB Fields Defined by AVIC

VMCB Offset	Bit(s)	Field Name	Description
098h	63:52	Reserved, SBZ	—
	51:12	V_APIC_BAR	Bits 51:12 of the GPA of the guest vAPIC register bank
	11:0	Reserved, SBZ	—
0E0h	63:52	Reserved, SBZ	—
	51:12	AVIC_BACKING_PAGE Pointer	Bits 51:12 of HPA of the vAPIC backing page
	11:0	Reserved, SBZ	—
0F0h	63:52	Reserved, SBZ	—
	51:12	AVIC_LOGICAL_TABLE Pointer	Bits 51:12 of HPA of the Logical APIC Table
	11:0	Reserved, SBZ	—
0F8h	63:52	Reserved, SBZ	—
	51:12	AVIC_PHYSICAL_TABLE Pointer	Bits 51:12 of HPA for the Physical APIC Table
	11:8	Reserved, SBZ	—
	7:0	AVIC_PHYSICAL_MAX_INDEX	Index of the last guest physical core ID for this guest

These fields are discussed further in the following paragraphs:

V_APIC_BAR—VMCB, Offset 098h. This entry is used to hold a copy of guest physical base address of its local APIC register block. The guest can change the GPA of its local APIC register block by writing to the guest version of the APIC Base Address Register (MSR 0000_001Bh). Writes to this MSR are intercepted by the VMM and the value is used to update the GPA in the nested page table entry for the vAPIC backing page and the value to be saved in this field of the VMCB.

APIC_BACKING_Page Pointer—VMCB, Offset 0E0h. This is a 52-bit HPA pointer to the vAPIC backing page for this virtual processor. The vAPIC backing page is described in more detail in the following section.

Logical APIC Table Pointer—VMCB, Offset 0F0h. This is a 52-bit HPA pointer to the Logical APIC ID Table for the virtual machine containing this virtual processor. This table is described in more detail in the following section.

Physical APIC Table Pointer—VMCB, Offset 0F8h. This is a 52-bit HPA pointer to the Physical APIC ID Table for the virtual machine containing this virtual processor. This table is described in more detail in the following section.

AVIC_PHYSICAL_MAX_INDEX—VMCB, Offset 0F8h. Bits [7:0]. This 8-bit value provides the index of the last guest physical core ID for this guest.

15.29.4.3 Restrictions on Physical Address Pointers

All of the physical addresses in the previous sections must point to legal, implementation-supported physical address ranges. These pointers are evaluated on VMRUN and cause a #VMEXIT if they are outside of the legal range. These memory ranges must be mapped as write-back cacheable memory type.

All the addresses point to 4-Kbyte aligned data structures. Bits 11:0 are reserved (except for offset 0F8h) and should be set to zero. The lower 8 bits of offset 0F8h are used for the field AVIC_PHYSICAL_MAX_INDEX.

Multiprocessor VM requirements. When running a VM which has multiple virtual CPUs, and the VMM runs a virtual CPU on a core which had last run a different virtual CPU from the same VM, regardless of the respective ASID values, care must be taken to flush the TLB on the VMRUN using a TLB_CONTROL value of 3h. Failure to do so may result in stale mappings misdirecting virtual APIC accesses to the previous virtual CPU's APIC backing page.

15.29.5 AVIC Memory Data Structures

The AVIC architecture defines three new memory-resident data structures. Each of these structures is defined to fit exactly in one 4-Kbyte page. Future implementations may expand the size.

15.29.5.1 Virtual APIC Backing Page

Each virtual processor in the system is assigned a virtual APIC backing page (vAPIC backing page). Accesses by the guest to the local APIC register block in the guest physical address space are redirected to the vAPIC backing page in system memory. The vAPIC backing page is used by AVIC hardware and the VMM to emulate the local APIC. See “Virtual APIC Register Accesses” on page 549 for a detailed description.

15.29.5.2 Physical APIC ID Table

The physical APIC ID table is set up and maintained by the VMM and is used by the hardware to locate the proper vAPIC backing page to be used to deliver interrupts based on the guest physical APIC ID. One physical APIC ID table must be provided per virtual machine.

The guest physical APIC ID is used as an index into this table. Each entry contains a pointer to the virtual processor's vAPIC backing page, a bit to indicate whether the virtual processor is currently scheduled on a physical core, and if so, the physical APIC ID of that core.

The length of this table is fixed at 4 Kbytes allowing a maximum of 512 virtual processors per virtual machine. However, in this version of the architecture the maximum number of virtual processors per guest is limited to 256. The physical ID table can be populated in a sparse manner using the valid bit to indicate assigned IDs. The index of the last valid entry is stored in the VMCB AVIC_PHYSICAL_MAX_INDEX field.

A pointer to this table is maintained in the VMCB. Because there is a single Physical APIC ID Table per virtual machine, the value of this pointer is the same for every virtual processor within the virtual machine.

Each entry in the table has the following format:

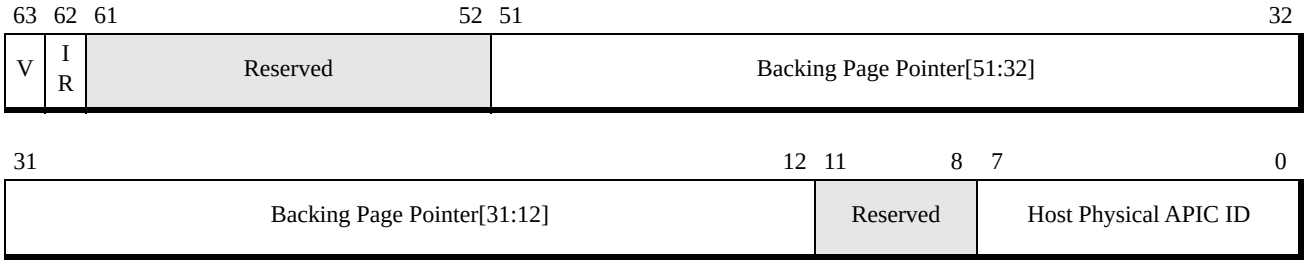


Figure 15-17. Physical APIC ID Table Entry

Table 15-25. Physical APIC ID Table Entry Fields

Bit(s)	Field Name	Description
63	V	Valid bit. When set, indicates that this entry contains a valid vAPIC backing page pointer. If cleared, this table entry contains no information.
62	IR	IsRunning. This bit indicates that the corresponding guest virtual processor is currently scheduled by the VMM to run on a physical core.
61:52	—	Reserved, SBZ. Should always be set to zero.
51:12	Backing Page Pointer	4-Kbyte aligned HPA of the vAPIC backing page for this virtual processor.
11:8	—	Reserved/SBZ for legacy APIC; extension of Host Physical APIC ID when x2APIC is enabled.
7:0	Host Physical APIC ID	Physical APIC ID of the physical core allocated by the VMM to host the guest virtual processor. This field is not valid unless the IsRunning bit is set.

Note that the IR bit, when set, indicates that the VMM has assigned a physical core to host this virtual processor. The bit does not differentiate between a physical processor running in guest mode (actively executing guest software) or in hypervisor mode (having suspended the execution of guest software).

The Physical APIC ID Table occupies the lower half of a single 4-Kbyte memory page, formatted as follows:

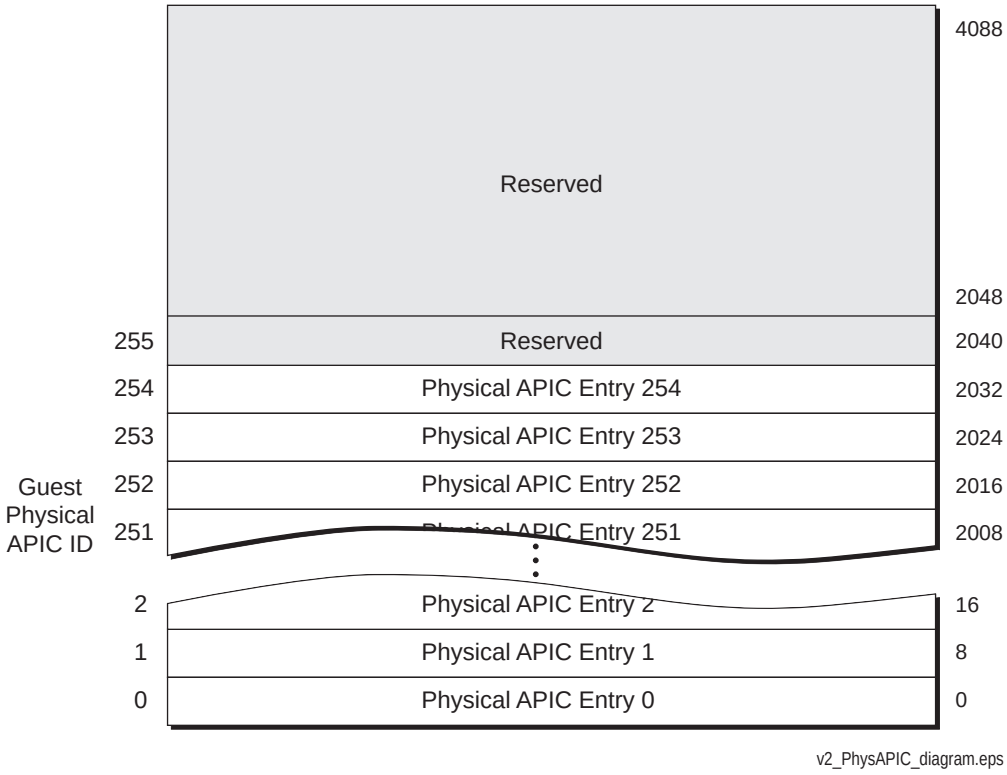


Figure 15-18. Physical APIC Table in Memory.

Since a destination of FFh is used to specify a broadcast, physical APIC ID FFh is reserved. The upper 2048 bytes of the table are reserved and should be set to zero.

15.29.5.3 Logical APIC ID Table

In addition to the Physical APIC ID Table, each guest VM is assigned a Logical APIC ID Table. This table is used to lookup the guest physical APIC ID for logically addressed interrupt requests. Each entry of this table provides the guest physical APIC ID corresponding to a single logically addressed APIC. Note that this implies that the logical ID of each vAPIC must be unique. The entries of this table are selected using the logical ID and interpreted differently depending upon logical APIC addressing mode of the guest. logical destination modes are supported: flat clustered.

If the guest attempts to change the logical ID of its APIC, the VMM must reflect this change in the Logical APIC ID Table. AVIC hardware supports the fixed interrupt message type targeting one or more logical destinations. The hardware also supports self and broadcast delivery modes specified via the Destination Shorthand (DSH) field of the ICRL. Any other message types must be supported through emulation by the VMM.

A pointer to this table is maintained in the VMCB. Because there is a single Logical APIC ID Table per virtual machine, the value of this pointer is the same for every virtual processor within the virtual machine.

For all logical destination modes, the table entries have the following format:

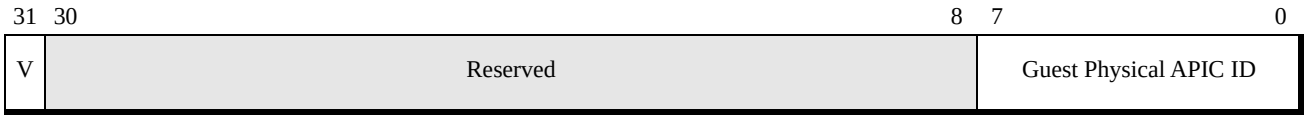


Figure 15-19. Logical APIC ID Table Entry

Table 15-26. Logical APIC ID Table Entry Fields

Bit(s)	Field Name	Description
31	V	Valid Bit. When set, indicates that this table entry contains a valid physical APIC ID. If cleared, this table entry contains no information.
30:8	—	Reserved, SBZ. Should always be set to zero.
7:0	Guest Physical APIC ID	Guest physical APIC ID corresponding to the local APIC selected when logically addressed.

Logical APIC ID Table Format for Flat Mode. When running in flat mode, AVIC expects the logical APIC ID table to be formatted as shown in Figure 15-20 below. This mode uses only the first 8 entries of the table. Although the logical APIC ID is an eight bit value, supported encodings must be of the form 2^i, where i = 0 to 7. In the figure the value i is used and represents the index into the table. The actual byte offset into the table for a given logical APIC ID *l_apic_id* is 4 * $\log_2$(l_apic_id).

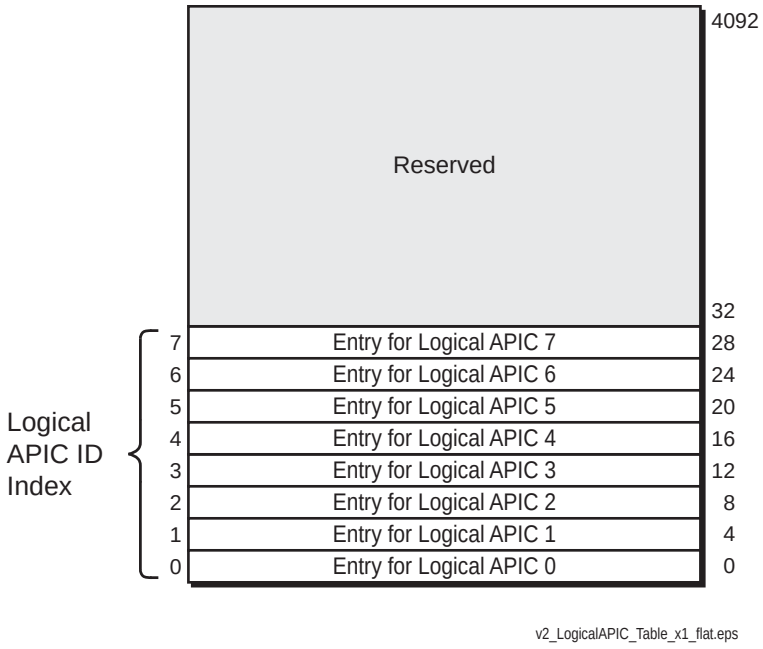


Figure 15-20. Logical APIC ID Table Format, Flat Mode.

Logical APIC ID Table Format for Cluster Mode. In cluster mode, bits [7:4] of the logical APIC ID represent the cluster number and bits [3:0] represent the APIC index (bit encoded). The cluster number Fh (15) is reserved. Since the APIC index field is four bits, four encodings are supported for the APIC index value.

The actual byte offset into the table for a given cluster *c* and an APIC index *apic_ix* is (16 * c) + 4 * $\log_2$(apic_ix)

When running in cluster mode, AVIC expects the logical APIC ID table to be formatted as shown in Figure 15-21 below.

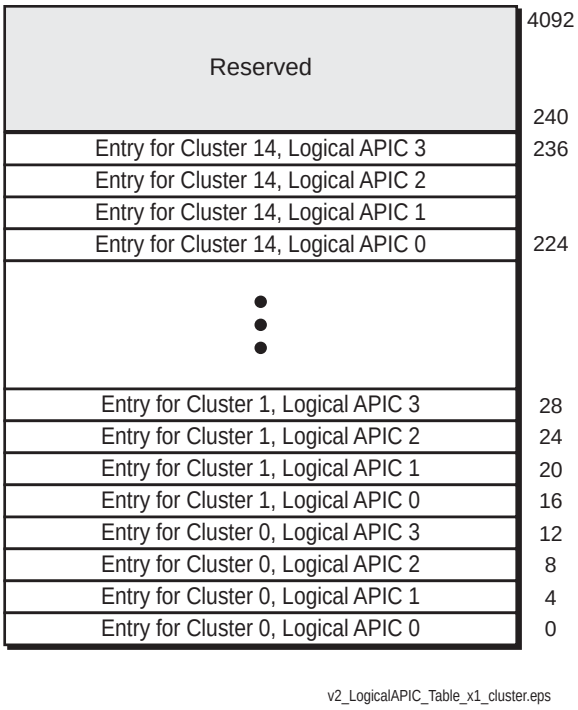


Figure 15-21. Logical APIC ID Table Format, Cluster Mode.

15.29.6 Interrupt Delivery

There are two fundamental types of virtual interrupts—interprocessor interrupts (IPIs) and I/O device interrupts (device interrupts). An IPI is initiated when guest system software writes the ICRL register. A device interrupt is initiated by a I/O device that has been programmed by guest system software (usually a device driver) to send a message signaling an event to a specific guest physical processor. This message usually includes an interrupt vector number indicating the nature of the event.

The following sections discuss the actions taken by AVIC hardware when a virtual processor signals an IPI and the actions taken by I/O virtualization hardware when a device signals a virtual interrupt.

15.29.6.1 Interprocessor Interrupts

To process an IPI, AVIC hardware executes the following steps:

1. If the destination-shorthand coded in the command is 01b (i.e. self), update the IRR in the backing page, signal doorbell to self and skip remaining steps.
2. If destination-shorthand is non-zero, or if the destination field is FFh (i.e. broadcast), jump to step 4.
3. If the destination(s) is (are) logically addressed, lookup the guest physical APIC IDs for each logical ID using the Logical APIC ID table.

   If the entry is not valid (V bit is cleared), cause a #VMEXIT.

If the entry is valid, but contains an invalid backing page pointer, cause a #VMEXIT.

4. Lookup the vAPIC backing page address in the Physical APIC table using the guest physical APIC ID as an index into the table.

   For directed interrupts, if the selected table entry is not valid, cause a #VMEXIT. For broadcast IPIs, invalid entries are ignored.

5. For every valid destination:
   - Atomically set the appropriate IRR bit in each of the destinations' vAPIC backing page.
   - Check the IsRunning status of each destination.
   - If the destination IsRunning bit is set, send a doorbell message using the host physical core number from the Physical APIC ID table.

6. If any destinations are identified as not currently scheduled on a physical core (i.e., the IsRunning bit for that virtual processor is not set), cause a #VMEXIT.

Refer to Section 15.29.9.1, "AVIC IPI Delivery Not Completed," on page 563 for new exitcodes associated with the #VMEXIT exceptions listed above.

15.29.6.2 Device Interrupts

The delivery of a I/O device interrupt to a virtual processor is handled by an IOMMU with virtual interrupt capability. To deliver a virtual interrupt, I/O virtualization hardware executes the following steps:

1. An interrupt message arrives from the I/O device identifying the source device and interrupt vector number.
2. I/O virtualization hardware uses the device ID to determine the guest physical APIC ID of the core that is the target of the device interrupt.
3. I/O virtualization hardware uses the guest physical APID ID to index into the Physical APIC ID Table to find the SPA of the vAPIC backing page. If the I/O virtualization hardware accesses an entry in the Physical APIC ID Table that is not valid (V bit is cleared), the I/O virtualization hardware aborts the virtual interrupt delivery and logs an error.
4. I/O virtualization hardware performs any required vector number translation.
5. I/O virtualization hardware atomically sets the bit in the IRR in the vAPIC backing page that corresponds to the vector.
6. If the virtual processor that is the target of the interrupt is not currently running on its assigned physical core, the virtual interrupt will be presented when the virtual processor is made active again. I/O virtualization hardware may provide additional information to the VMM about the device interrupt to aid in virtual processor scheduling decisions.

   If the virtual processor that is the target of the interrupt is scheduled on a physical processor (indicated by the IsRunning bit of the Physical APIC ID table entry being set), I/O virtualization hardware uses the host physical APIC ID in the table entry to send a doorbell signal to the corresponding processor core to signal that an interrupt needs to be processed.

15.29.7 CPUID Feature Bits for AVIC

A CPUID feature bit is used indicate support for AVIC on a specific hardware implementation. CPUID Fn8000_000A_EDX[AVIC] is designated for this purpose and is returned in bit 13 of EDX. If EDX[13] is set, the AVIC architecture is supported on that hardware.

See Section 3.3, “Processor Feature Identification,” on page 71 for more information on using the CPUID instruction.

15.29.8 New Processor Mechanisms

In order to support the direct injection of interrupts into the guest and to accelerate critical vAPIC functions, new hardware mechanisms are implemented in the processor.

15.29.8.1 Special Trap/Fault Handling for vAPIC Accesses

To virtualize the local APIC utilized by the guest to generate and process interrupts, all read and write accesses by the guest virtual processor to its local APIC registers are redirected to the vAPIC backing page. Most reads and many writes to this guest physical address range read or write the contents of memory locations within the vAPIC backing page at the corresponding offset.

To support proper handling and emulation of the guest local APIC, the processor provides permissions filtering hardware (Refer to Figure 15-15.) that detects and intercepts accesses to specific offsets (representing APIC registers) within the vAPIC backing page. This hardware either allows the access, blocks the access and causes a #VMEXIT (fault behavior), or allows the access and then causes a #VMEXIT (trap behavior).

Hardware directly handles the side effects of guest writes to the TPR and EOI registers. Writes to the ICRL register with simple functional side effects such as the generation of a directed IPI or a self-IPI request are handled directly. Values written to the ICRL defined to initiate more complex behavior cause a #VMEXIT to allow the VMM to emulate the function. A guest write to the ICRH register has no immediate hardware side effect and is allowed.

Most other write access attempts within the vAPIC register bank address range cause a #VMEXIT with trap or fault behavior allowing the VMM to emulate the function of that register. See Table 15-22 for more detail.

Reads and writes to locations within the vAPIC backing page, but outside the offset range of defined vAPIC registers are allowed to complete.

15.29.8.2 Doorbell Mechanism

Each core provides a doorbell mechanism that is used by other cores (for IPIs) and the IOMMU (for device interrupts) to signal to the VMM of the target physical core that a virtual interrupt requires processing. The exact mechanism is implementation-specific, but must be protected from access from non-privileged software running on other cores and from direct access by an external device.

When the doorbell is received in guest mode, hardware on the receiving core evaluates the vAPIC state in the vAPIC backing page for the currently running virtual processor and injects the interrupt into the guest as appropriate.

Doorbell Register. The system programming interface to the doorbell mechanism is provided via an MSR. Sending a doorbell signal to a another core is initiated by writing the physical APIC ID corresponding to that core to the Doorbell Register (MSR C001_011Bh). The format of this register is shown in Figure 15-22 below.

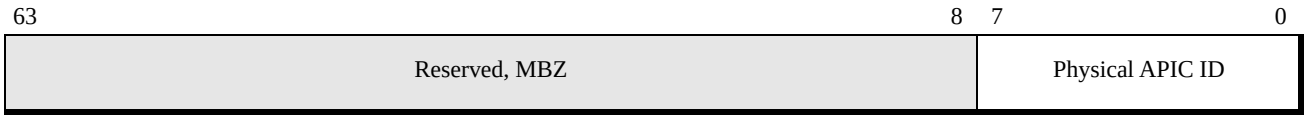


Figure 15-22. Doorbell Register, MSR C001_011Bh

Writing to this register causes a doorbell signal to be sent to the specified physical core. The serializing semantics of WRMSR are relaxed when writing to the Doorbell Register. Any attempt to read from this register results in a #GP.

Processing of Doorbell Signals. A doorbell signal delivered to a running guest is recognized by the hardware regardless of whether it can be immediately injected into the guest as a virtual interrupt. On the next VMRUN, the virtual interrupt delivery mechanism evaluates the state of the IRR register of the guest's vAPIC backing page to find the highest priority pending interrupt and injects it if interrupt masking and priority allow.

15.29.8.3 Additional VMRUN Handling

In addition to the normal VMRUN operations, the core re-evaluates the APIC state in the vAPIC backing page upon entry into the guest and processes pending interrupts as necessary. Specifically:

- On VMRUN the interrupt state is evaluated and the highest priority pending interrupt indicated in the IRR is delivered if interrupt masking and priority allow
- Any doorbell signals received during VMRUN processing are recognized immediately after entering the guest
- When AVIC mode is enabled for a virtual processor, the V_IRQ, V_INTR_PRIO, V_INTR_VECTOR, and V_IGN_TPR fields in the VMCB are ignored.

15.29.9 New Exit Codes for AVIC

The AVIC architecture defines two new AVIC-related #VMEXIT events. These cases are described in the following sections. Assigned EXITCODE values are given in Table C-1 on page 693.

15.29.9.1 AVIC IPI Delivery Not Completed

An IPI could not be delivered to all targeted guest virtual processors because at least one guest virtual processor was not allocated to a physical core at the time. This results in a #VMEXIT with an exit code

of AVIC_INCOMPLETE_IPI. Additional data associated with this #VMEXIT event is returned in the EXITINFO1 and EXITINFO2 fields.

EXITINFO1. This field contains the values written to the vAPIC ICRH and ICRL registers.

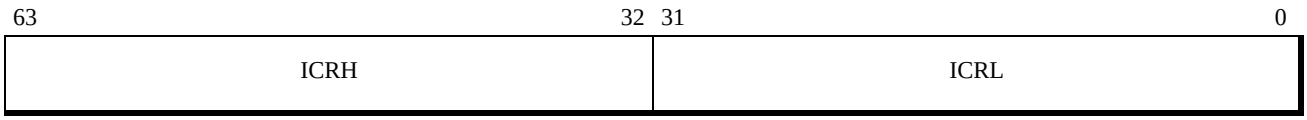


Figure 15-23. EXITINFO1

Table 15-27. EXTINFO1 Fields

Bit(s)	Field Name	Description
63:32	ICRH	Value written to the vAPIC ICRH register.
31:0	ICRL	Value written to the vAPIC ICRL register.

EXITINFO2. This field contains information describing the specific reason for the IPI delivery failure.

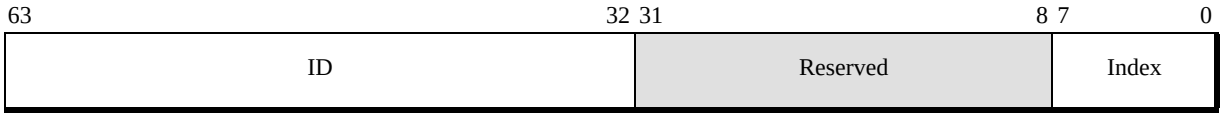


Figure 15-24. EXITINFO2

Table 15-28. EXTINFO2 Fields

Bit(s)	Field Name	Description
63:32	ID	Specific reason for the delivery failure. See Table 15-29 for defined values.
31:8	—	Reserved
7:0	Index	For ID = 1 – 3, this field provides the index of a logical or physical table entry. Reserved for all other ID values.

The ID field identifies the reason for the IPI delivery failure:

Table 15-29. ID Field—IPI Delivery Failure Cause

ID	Cause	Description	Index
0	Invalid Interrupt type	The trigger mode for the specified IPI was set to level or the destination type is unsupported.	Reserved.
1	IPI Target Not Running	IsRunning bit of the target for a Singlecast/Broadcast/Multicast IPI is not set in the physical APIC ID table.	Index of the physical or logical APIC ID table entry for the target virtual processor that was not scheduled on a physical core.
2	Invalid Target in IPI	Target ID invalid. This is due to one the following reasons: • In logical mode: cluster > max_cluster (64) • In physical mode: target > max_physical (512) • address is not present in the physical or logical ID tables	Index of the physical or logical table entry for the invalid target.
3	Invalid Backing Page Pointer	The vAPIC Backing Page Pointer field of the Physical APIC ID Table contained an invalid physical address.	For shorthand or broadcast delivery modes, index of the physical APIC ID Table containing the invalid address. For directed IPIs, index of the logical or physical APIC ID table depending on the destination mode.
> 3	Reserved	—	Reserved

15.29.9.2 AVIC Access to un-accelerated vAPIC register

A guest access to an APIC register that is not accelerated by AVIC results in a #VMEXIT with the exit code of AVIC_NOACCEL. This fault is also generated if an EOI is attempted when the highest priority in-service interrupt is set for level-triggered mode. Additional data associated with this #VMEXIT event is returned in the EXITINFO1 and EXITINFO2 fields.

EXITINFO1. This field contains the offset of the un-accelerated virtual APIC register and a bit indicating whether a read or write operation was attempted.

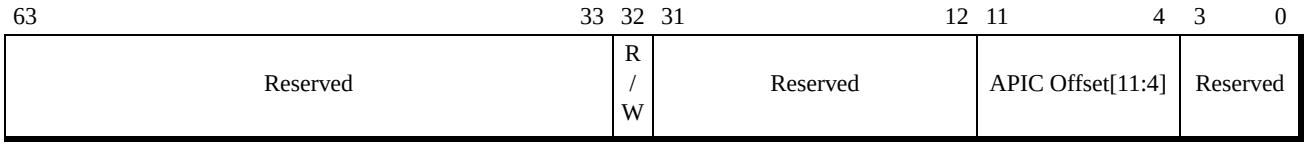


Table 15-30. EXTINFO1 Fields

Bit(s)	Field Name	Description
63:33	—	Reserved.
32	R/W	If set, write was attempted. If clear, read was attempted.

Table 15-30. EXTINFO1 Fields (continued)

Bit(s)	Field Name	Description
31:12	—	Reserved.
11:4	APIC_Offset[11:4]	Offset within virtual vAPIC backing page at which read or write was attempted. APIC_Offset[3:0] = 0, since all registers are aligned on 16-byte boundaries.
3:0	—	Reserved.

EXITINFO2. This field contains extra information for the un-accelerated operation. If the EXITINFO1 fields indicate a write to the vAPIC EOI register (offset = B0h), bits 7:0 of this value contain the number of the highest in-service vector found in the virtual APIC ISR.

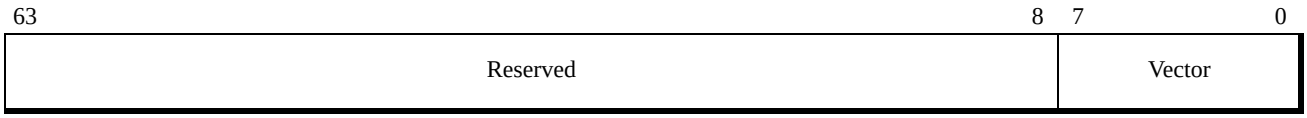


Table 15-31. EXTINFO2 Fields

Bit(s)	Field Name	Description
63:8	—	Reserved.
31:0	Vector	Vector for attempted EOI; otherwise undefined.

15.30 SVM Related MSRs

SVM uses the following MSRs for various control purposes. These MSRs are available regardless of whether SVM is enabled in EFER.SVME. For details on implementation-specific features, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.

15.30.1 VM_CR MSR (C001_0114h)

The VM_CR MSR controls certain global aspects of SVM. The layout of the MSR is shown in Figure 15-25.

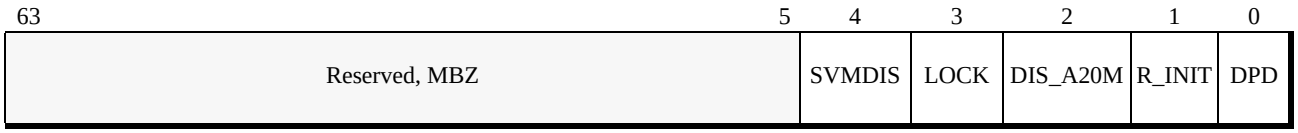


Figure 15-25. Layout of VM_CR MSR (C001_0114h)

The individual fields are as follows:

- DPD—Bit 0. If set, disables the external hardware debug port and certain internal debug features.
- R_INIT—Bit 1. If set, non-intercepted INIT signals are converted into an #SX exception.

- DIS_A20M—Bit 2. If set, disables A20 masking.
- LOCK—Bit 3. When this bit is set, writes to LOCK and SVMDIS are silently ignored. When this bit is clear, VM_CR bits 3 and 4 can be written. Once set, LOCK can only be cleared using the SVM_KEY MSR (See Section 15.31.) This bit is not affected by INIT or SKINIT.
- SVMDIS—Bit 4. When this bit is set, writes to EFER treat the SVME bit as MBZ. When this bit is clear, EFER.SVME can be written normally. This bit does not prevent CPUID from reporting that SVM is available. Setting SVMDIS while EFER.SVME is 1 generates a #GP fault, regardless of the current state of VM_CR.LOCK. This bit is not affected by SKINIT. It is cleared by INIT when LOCK is cleared to 0; otherwise, it is not affected.

15.30.2 IGNNE MSR (C001_0115h)

The read/write IGNNE MSR is used to set the state of the processor-internal IGNNE signal directly. This is only useful if IGNNE emulation has been enabled in the HW_CR MSR (and thus the external signal is being ignored). Bit 0 specifies the current value of IGNNE; all other bits are MBZ.

15.30.3 SMM_CTL MSR (C001_0116h)

The write-only SMM_CTL MSR provides software control over SMM signals. SMM_CTL MSR is not supported when CPUID Fn8000_0021[NoSmmCtlMSR] is set.

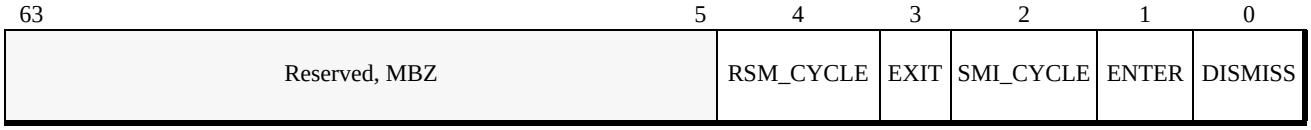


Figure 15-26. Layout of SMM_CTL MSR (C001_0116h)

Writing individual bits causes the following actions:

- DISMISS—Bit 0. Clear the processor-internal “SMI pending” flag.
- ENTER—Bit 1. Enter SMM: map the SMRAM memory areas, record whether NMI was currently blocked and block further NMI and SMI interrupts.
- SMI_CYCLE—Bit 2. Send SMI special cycle.
- EXIT—Bit 3. Exit SMM: unmap the SMRAM memory areas, restore the previous masking status of NMI and unconditionally reenable SMI.
- RSM_CYCLE—Bit 4. Send RSM special cycle.

Writes to the SMM_CTL MSR cause a #GP if platform firmware has locked the SMM control registers by setting HWCR[SMMLOCK].

Conceptually, the bits are processed in the order of ENTER, SMI_CYCLE, DISMISS, RSM_CYCLE, EXIT, though only the following bit combinations may be set together in a single write (for all other combinations of more than one bit, behavior is undefined):

- ENTER + SMI_CYCLE

- DISMISS + ENTER
- DISMISS + ENTER + SMI_CYCLE
- EXIT + RSM_CYCLE

The VMM must ensure that ENTER and EXIT operations are properly matched, and *not* nested, otherwise processor behavior is undefined. Also undefined are ENTER when the processor is already in SMM, and EXIT when the processor is not in SMM.

15.30.4 VM_HSAVE_PA MSR (C001_0117h)

The 64-bit read/write VM_HSAVE_PA MSR holds the physical address of a 4KB block of memory where VMRUN saves host state, and from which #VMEXIT reloads host state. The VMM software is expected to set up this register before issuing the first VMRUN instruction. Software must not attempt to read or write the host save-state area directly.

Writing this MSR causes a #GP if:

- any of the low 12 bits of the address written are nonzero, or
- the address written is greater than or equal to the maximum supported physical address for this implementation.

15.30.5 TSC Ratio MSR (C000_0104h)

Writing to the TSC Ratio MSR allows the hypervisor to control the guest's view of the Time Stamp Counter. The contents of TSC Ratio MSR sets the value of the TSCRatio. This constant scales the timestamp value returned when the TSC is read by a guest via the RDTSC or RDTSCP instructions or when the TSC, MPERF, or MPerfReadOnly MSRs are read via the RDMSR instruction by a guest running under virtualization.

This facility allows the hypervisor to provide a consistent TSC, MPERF, and MPerfReadOnly rate for a guest process when moving that process between cores that have a differing P0 rate. The TSCRatio does not affect the value read from the TSC, MPERF, and MPerfReadOnly MSRs when in host mode or when virtualization is disabled. System Management Mode (SMM) code sees unscaled TSC, MPERF and MPerfReadOnly values unless the SMM code is executed within a guest container. The TSCRatio value does not affect the rate of the underlying TSC, MPERF, and MPerfReadOnly counters, nor the value that gets written to the TSC, MPERF, and MPerfReadOnly MSRs counters on a write by either the host or the guest.

The TSC Ratio MSR specifies the TSCRatio value as a fixed-point binary number in 8.32 format, which is composed of 8 bits of integer and 32 bits of fraction. This number is the ratio of the desired P0 frequency to be presented to the guest relative to the P0 frequency of the core (See Section 17.1, “P-State Control,” on page 639). The reset value of the TSCRatio is 1.0, which sets the guest P0 frequency to match the core P0 frequency.

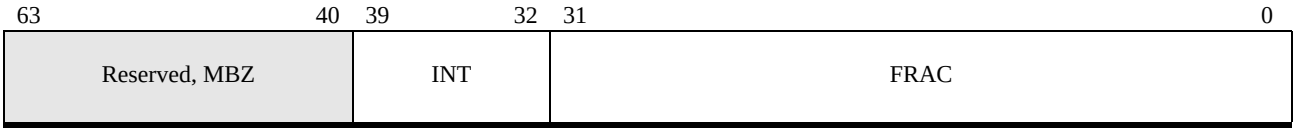
Note that:

TSCFreq = Core P0 frequency * TSCRatio, so TSCRatio = (Desired TSCFreq) / Core P0 frequency.

The TSC value read by the guest is computed using the TSC Ratio MSR along with the TSC_OFFSET field from the VMCB so that the actual value returned is:

TSC Value (in guest) = (P0 frequency * TSCRatio * t) + VMCB.TSC_OFFSET + (Last Value Written to TSC) * TSCRatio
Where **t** is time since the TSC was last written via the TSC MSR (or since reset if not written)

The layout of the TSC Ratio MSR is illustrated in figure below.



Bits	Mnemonic	Description	Access Type
63:40	—	Reserved	Reserved, MBZ
39:32	INT	Integer Part	R/W
31:0	FRAC	Fractional Part	R/W

Figure 15-27. TSC Ratio MSR (C000_0104h)

INT. Integer Part. Bits 39:32. Integer part of TSCRatio.

FRAC. Fractional Part. Bits 39:32. Fractional part of TSCRatio.

$TSCRatio = INT + FRAC \times 2^{-32}$

CPUID Fn8000_000A_EDX[TscRateMsr] =1 indicates support for the TSC Ratio MSR. See Section 3.3, “Processor Feature Identification,” on page 71 for more information on using the CPUID instruction.

15.31 SVM-Lock

The SVM-Lock feature allows software to prevent EFER.SVME from being set, either unconditionally or with a 64-bit key to re-enable SVM functionality.

Support for SVM-Lock is indicated by CPUID Fn8000_000A_EDX[SVML] = 1. On processors that support the SVM-Lock feature, SKINIT and STGI can be executed even if EFER.SVME=0. See descriptions of LOCK and SVMDIS bits in Section 15.30.1. When the SVM-Lock feature is not available, hypervisors can use the read-only VM_CR.SVMDIS bit to detect SVM (see Section 15.4).

15.31.1 SVM_KEY MSR (C001_0118h)

The write-only SVM_KEY MSR is used to create a password-protected mechanism to clear VM_CR.LOCK.

When VM_CR.LOCK is zero, writes to SVM_KEY MSR set the 64-bit SVM Key value.

When VM_CR.LOCK is one, writes to SVM_KEY MSR compare the written value to the SVM Key value; if the values match and are non-zero, the VM_CR.LOCK bit is cleared. If the values mismatch or the SVM Key value is zero, the write to SVM_KEY is ignored, and VM_CR.LOCK is unmodified. Software should read VM_CR.LOCK after writing SVM_KEY to determine whether the unlock succeeded.

If SVM Key is zero when VM_CR.LOCK is one, VM_CR.LOCK can only be cleared by a processor reset.

To preserve the security of the SVM key, reading the SVM_KEY MSR always returns zero.

15.32 SMM-Lock

The SMM-Lock feature allows platform firmware to prevent System Management Interrupts (SMI) from being intercepted in SVM. The SmmLock bit is located in the HWCR MSR register.

15.32.1 SmmLock Bit — HWCR[0]

The SmmLock bit (bit 0) is located in the HWCR MSR (C001_0015h). When SmmLock is clear, it can be set to one. Once set, the bit cannot be cleared by software and writes to it are ignored. SmmLock can only be cleared using the SMM_KEY MSR (see Section 15.32.2), or by a processor reset. This bit is not affected by INIT or SKINIT. When SmmLock is set, other SMM configuration registers cannot be written. For complete information on the HWCR register, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.

15.32.2 SMM_KEY MSR (C001_0119h)

The write-only SMM_KEY MSR is used to create a password-protected mechanism to clear SmmLock.

When SmmLock is zero, writes to SMM_KEY MSR set the 64-bit SMM Key value.

When SmmLock is one, writes to SMM_KEY MSR compare the written value to the SMM Key value; if the values match and are non-zero, the SmmLock bit is cleared. If the values mismatch or the SMM Key value is zero, the write to SMM_KEY is ignored, and SmmLock is unmodified. Software should read SmmLock after writing SMM_KEY to determine whether the unlock succeeded.

If SMM_Key MSR is equal to zero when SmmLock is one, SmmLock can only be cleared by a processor reset.

To preserve the security of the SMM key, reading SMM_KEY MSR always returns zero.

15.33 Nested Virtualization

Hardware support for improved performance of nested virtualization, which is the act of running a hypervisor as a guest under a higher-level hypervisor, is provided through the features described here.

These relieve the top-level hypervisor from performing certain common, high-overhead operations that can occur with nested virtualization.

15.33.1 VMSAVE and VMLOAD Virtualization

This feature allows the VMSAVE and VMLOAD instructions to execute in guest mode without causing a #VMEXIT. The VMCB address in RAX is treated as a guest physical address and is translated to a host physical address. Any page fault in attempting that translation will result in a normal #VMEXIT with a nested page fault exit code. If the translation is successful, the register state transfer to or from the VMCB will then be performed.

Support for virtualized VMSAVE and VMLOAD is indicated by CPUID Fn8000_000A_EDX[15]=1. When this feature is available, it must be explicitly enabled by setting bit 1 of VMCB offset 0B8h to 1. This enable bit is only recognized when the hypervisor is in 64 bit mode, nested paging is enabled and Secure Encrypted Virtualization is disabled, otherwise attempted execution of a VMLOAD or VMSAVE in the guest will result in a #VMEXIT with a VMSAVE/VMLOAD exit code.

15.33.2 Virtual GIF (VGIF)

This feature allows STGI and CLGI to execute in guest mode and control virtual interrupts in guest mode while still allowing physical interrupts to be intercepted by the hypervisor. The presence of the VGIF feature is indicated by CPUID Fn8000_000A_EDX[16]=1.

In order to provide this ability, two new bits are added to the VMCB field at offset 60h:

Offset	Bit	Description
60h	9	VGIF value (0 – Virtual interrupts are masked, 1 – Virtual Interrupts are unmasked)
60h	25	Virtual GIF enable for this guest (0 - Disabled, 1 - Enabled)

When a VMRUN is executed and VGIF is enabled, the processor uses bit 9 as the starting value of the virtual GIF. It then provides masking capability for when virtual interrupts are taken. STGI executed in the guest sets bit 9 of VMCB offset 60h and allows a virtual interrupt to be taken. CLGI executed in the guest clears bit 9 of VMCB offset 60h and causes the virtual interrupt to be masked. Bit 9 in the VMCB is also writeable by the hypervisor, and loaded on VMRUN and is saved on #VMEXIT.

The hypervisor can still use the STGI and CLGI intercept controls in the VMCB to intercept execution of these in the guest regardless of VGIF enablement.

15.34 Secure Encrypted Virtualization

Secure Encrypted Virtualization (SEV) is available when the CPU is running in guest mode utilizing AMD-V virtualization features. SEV enables running encrypted virtual machines (VMs) in which the code and data of the virtual machine are secured so that the decrypted version is available only within the VM itself. Each virtual machine may be associated with a unique encryption key so if data is

accessed by a different entity using a different key, the SEV encrypted VM's data will be decrypted with an incorrect key, leading to unintelligible data.

It is important to note that SEV mode therefore *represents a departure from the standard x86 virtualization security model*, as the hypervisor is no longer able to inspect or alter all guest code or data. The guest page tables, managed by the guest, may mark data memory pages as either private or shared, thus allowing selected pages to be shared outside the guest. Private memory is encrypted using a guest-specific key, while shared memory is accessible to the hypervisor.

15.34.1 Determining Support for SEV

Support for memory encryption features is reported in CPUID 8000_001F[EAX] as described in Section 7.10.1, “Determining Support for Secure Memory Encryption,” on page 228. Bit 1 indicates support for Secure Encrypted Virtualization.

When memory encryption features are present, CPUID 8000_001F[EBX] and 8000_001F[ECX] supply additional information regarding the use of memory encryption, such as the number of keys supported simultaneously and which page table bit is used to mark pages as encrypted. Additionally, in some implementations, the physical address size of the processor may be reduced when memory encryption features are enabled, for example from 48 to 43 bits. In this example, physical address bits 47:43 would be treated as reserved except where otherwise indicated. When memory encryption is supported in an implementation, CPUID 8000_001F[EBX] reports any physical address size reduction present. Bits reserved in this mode are treated the same as other page table reserved bits, and will generate a page fault if found to be non-zero when used for address translation.

Full CPUID details for memory encryption features may be found in Volume 3, section E.4.17.

15.34.2 Key Management

Under the memory encryption extensions defined here, each SEV-enabled guest virtual machine is associated with a memory encryption key, and the SME mode (if used, see Section 7.10 on page 228) is associated with a separate key. Key management for the SEV feature is not handled by the CPU but rather by a separate processor known as the AMD Secure Processor (AMD-SP) which is present on AMD SOCs. A detailed discussion of AMD-SP operation is beyond the scope of this manual.

CPU software is not aware of the values of these keys but the hypervisor should coordinate the loading of virtual machine keys through the AMD-SP driver. This coordination will also determine which ASID the hypervisor should use for a particular guest. Under SEV, the ASID is used as the key index that identifies which encryption key is used to encrypt/decrypt memory traffic associated with that SEV-enabled guest. Encryption keys themselves are never visible to CPU software and are never stored off-chip in the clear.

15.34.3 Enabling SEV

Prior to starting an encrypted VM, software must enable MemEncryptionModEn through MSR C001_0010 (SYSCFG) as described in Section 7.10.2, “Enabling Memory Encryption Extensions,” on page 229. SEV may then be enabled on a specific virtual machine during the VMRUN instruction if the hypervisor sets the SEV enable bit in VMCB offset 090h.

Byte Offset	Bit[s]	Description
090h	0	Enable nested paging
	1	Enable Secure Encrypted Virtualization
	2	Enable Encrypted State for Secure Encrypted Virtualization
	63-3	Reserved, SBZ

When SEV is enabled in a guest, the following additional consistency checks are performed during VMRUN:

- Nested paging (VMCB offset 090h, bit 0) must be enabled
- MSR C001_0015 (HWCR) [SmmLock] must be set
- ASID (VMCB offset 058h) must be within the allowed range for SEV

The allowed ASIDs for SEV operation may be a subset of the overall number of hardware supported ASIDs. In this scenario, SEV-enabled guests must use ASIDs in the defined subset, while non-SEV enabled guests can use the remaining ASID range. The range of ASIDs allowed for SEV-enabled guests is from 1 to a maximum value defined via CPUID 8000_001F[ECX].

Note that on systems where CPUID Fn8000_001F_EAX[11] is set to 1, the hypervisor must be in 64-bit mode in order to execute a VMRUN to an SEV-enabled guest. If not the VMRUN fails with a VMEXIT_INVALID error code.

If any of the above consistency checks fail when SEV is enabled on a guest, the VMRUN instruction will terminate with a VMEXIT_INVALID error code. If MemEncryptionModEn is 0, SEV cannot be enabled and the VMCB control bit for SEV is ignored.

15.34.4 Supported Operating Modes

Secure Encrypted Virtualization may be enabled on guests running in any operating mode. However the guest is only able to control memory encryption when operating in long mode or legacy PAE mode. In all other modes, all guest memory accesses are unconditionally considered private and are encrypted with the guest-specific key.

15.34.5 SEV Encryption Behavior

When a guest is executed with SEV enabled, the guest page tables are used to determine the C-bit for a memory page and hence the encryption status of that memory page. This allows a guest to determine which pages are private or shared, but this control is available only for data pages. Memory accesses on behalf of instruction fetches and guest page table walks are always treated as private, regardless of

the software value of the C-bit. This behavior ensures non-guest entities (such as the hypervisor) cannot inject their own code or data into an SEV-enabled guest. If a guest does wish to make data in instruction pages or page tables accessible to code outside of the guest, this data must be explicitly copied into a shared data page.

Note that while the guest may choose to set the C-bit explicitly on instruction pages and page table addresses, the value of this bit is a don't-care in such situations as hardware always performs these as private accesses.

15.34.6 Page Table Support

An SEV-enabled guest controls encryption in its own guest page tables using the C-bit defined by CPUID 8000_001F[EBX]. This location is the same C-bit location as defined under SME (Section 7.10, “Secure Memory Encryption,” on page 228) in non-virtualized mode. If the C-bit is an address bit, this bit is masked from the guest physical address when it is translated through the nested page tables. Consequently, the hypervisor does not need to be aware of which pages the guest has chosen to mark private.

For example if the C-bit is address bit 47, when a guest accesses virtual address 0x54321, it might be translated to guest physical address 0x8000_00AB_C321, indicating the page should be encrypted with the private guest key. When this guest physical address is translated through the nested page tables, host virtual address 0xAB_C321 is used for translation. The C-bit value from the guest physical address is saved and used on the final system physical address after the nested table translation as shown in Figure 15-28.

Note that because guest physical addresses are always translated through the nested page tables, the size of the guest physical address space is not impacted by any physical address space reduction indicated in CPUID 8000_001F[EBX]. If the C-bit is a physical address bit however, the guest physical address space is effectively reduced by 1 bit.

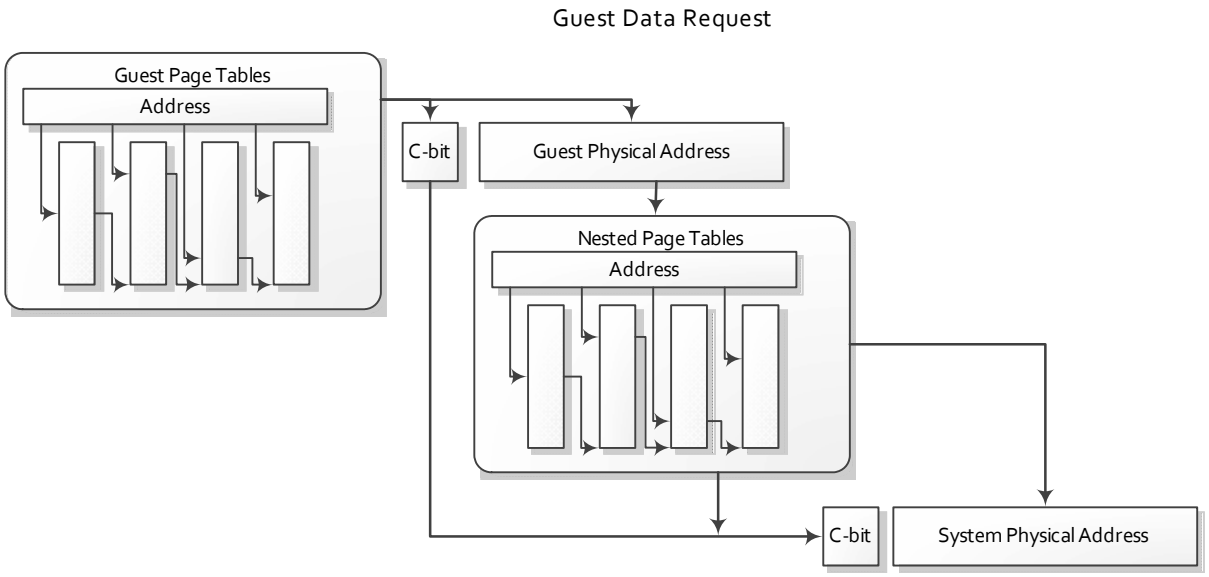


Figure 15-28. Guest Data Request

15.34.7 Restrictions

As with SME, some hardware implementations may not enforce coherency between mappings of the same physical page with different encryption enablement or keys. In such a system, when the encryption enablement or key for a particular memory page is to be changed, software must first ensure the page is flushed from all CPU caches. Certain conventional cache flushing techniques may not work however; see Section 15.34.9 for further details.

Note that if the hardware implementation enforces coherency across encryption domains as indicated by CPUID Fn8000_001F_EAX[10] then this flush is not required.

15.34.8 SEV Interaction with SME

SEV may be used in conjunction with SME mode. In this scenario, the guest page tables control encryption for guest memory, and the host (nested) page tables control encryption for shared memory. This behavior is summarized in Table 15-32. SEV is considered active when the CPU is in guest mode and the guest has SEV enabled in the VMCB.

Table 15-32. Encryption Control

Type of Access	MemEncryptionModEn	Guest Mode	SEV Mode Active	Encrypted	Encryption Key	Notes
Legacy Mode (memory encryption disabled)						
All	0	X	X	No	N/A	
Secure Memory Encryption Mode						
All	1	0	X	Optional	Host Key	Determined by page tables (CR3)
All	1	1	0	Optional	Host Key	Determined by nested page tables (hCR3)
Secure Encrypted Virtualization Mode						
Instruction Fetch	1	1	1	Yes	Guest Key	
Guest Page Table Access	1	1	1	Yes	Guest Key	
Nested Page Table Access	1	1	1	Optional	Host Key	Determined by nested page tables (hCR3)
Data Access	1	1	1	Optional[1]	See Table 15-33: SEV/SME Interaction	Determined by guest page tables (gCR3) and nested page tables (hCR3)

Note:

1. Encryption is guest-controlled in long mode and legacy PAE mode only. In all other modes, these accesses are always considered private and are encrypted with the guest key

Note that during a nested page table walk, it is possible for both the guest page tables to be encrypted and the nested page tables to be encrypted. In this scenario, the guest page tables are decrypted using the guest private encryption key, and the nested page tables are decrypted using the host (SME) encryption key.

Guest data accesses that are marked shared (C=0) by the guest may still be optionally encrypted using the host (SME) key if the pages are marked encrypted in the nested tables. If a page is marked encrypted in both the guest and nested tables, the guest tables have priority and the page will be encrypted using the guest key. This behavior is summarized in Table 15-33.

Table 15-33. SEV/SME Interaction

		Nested Page Table	
		C=0	C=1
Guest Page Table	C=0	Unencrypted	Encrypted with host key
	C=1	Encrypted with guest key	Encrypted with guest key

15.34.9 Page Flush MSR

If coherency across encryption domains is not supported (see “Restrictions” on page 575), and the hypervisor wishes to read an encrypted page, it must first flush the guest view of that page from all CPU caches to ensure it is able to view the most recent copy of that data. This may be accomplished by issuing a WBINVD instruction on all cores on which the guest has run, or by using the VMPAGE_FLUSH MSR (C001_011E). Support for the VMPAGE_FLUSH MSR is indicated in CPUID 8000_001F[EAX] bit 2.

The VMPAGE_FLUSH MSR is a write-only register that may be used to flush 4KB of data on behalf of a guest. The hypervisor writes the host linear address of the page and guest ASID to the MSR, and hardware will then perform a write-back invalidation of the page causing any dirty data present in any CPU caches throughout the system to be encrypted and written to DRAM. Note that the VMPAGE_FLUSH MSR uses the standard host page tables to perform the page translation. The Page Flush MSR operation will hit on and evict guest-cached instances of the memory, whereas CLFLUSH instructions using this same translation will not.

Bit[s]	Description
63:12	**VirtualAddr:** Write-only. Host virtual address of page to flush
11:0	**ASID:** Write-only. Guest ASID to use for the flush

The VMPAGE_FLUSH MSR will only flush memory pages marked private by the guest. If the hypervisor does not know if the memory page was marked private but wishes to evict the page from the cache, it should perform a standard CLFLUSH in addition to using the VMPAGE_FLUSH MSR.

Attempts to flush a host virtual address that is not mapped into a physical address or use of an ASID=0 will cause a #GP(0) fault.

15.34.10 SEV_STATUS MSR

Guests can determine what SEV features are currently active by reading the SEV_STATUS MSR (C001_0131). This MSR indicates which SEV features (if any) were enabled by the hypervisor in the last VMRUN for that guest as shown in Table 15-34. The SEV_STATUS MSR is read-only and accesses to the SEV_STATUS MSR cannot be intercepted by the hypervisor. The SEV_STATUS MSR is available on all platforms that support SEV. Bits 9:2 of SEV_STATUS reflect the enablement of various SEV-SNP security features as described in section 15.36.

Table 15-34. SEV_STATUS MSR Fields

Bit[s]	Description
63:17	**Reserved**
16	**VmsaRegProt_Enabled:** The guest was run with the VMSA Register Protection feature enabled in SEV_FEATURES[14]
15:12	**Reserved**
11	**SecureTsc_Enabled:** The guest was run with the Secure TSC feature enabled in SEV_FEATURES[9]
10	**Reserved**
9	**SNPBTBIsolation_Enabled:** The guest was run with the BTB isolation feature enabled in SEV_FEATURES[7]
8	**PreventHostIBS_Enabled:** This guest was run with the PreventHostIBS feature enabled in SEV_FEATURES[6].
7	**DebugSwap_Enabled:** This guest was run with debug register swapping enabled in SEV_FEATURES[5].
6	**AlternateInjection_Enabled:** The guest was run with the Alternate Injection feature enabled in SEV_FEATURES[4]
5	**RestrictedInjection_Enabled:** The guest was run with the Restricted Injection feature enabled in SEV_FEATURES[3]
4	**ReflectVC_Enabled:** The guest was run with the ReflectVC feature enabled in SEV_FEATURES[2]
3	**vTOM_Enabled:** The guest was run with the Virtual TOM feature enabled in SEV_FEATURES[1]
2	**SNP_Active:** The guest was run in SNP-Active mode as selected by SEV_FEATURES[0]
1	**SEV_ES_Enabled:** The guest was run with the SEV-ES feature enabled in VMCB offset 90h
0	**SEV_Enabled:** The guest was run with the SEV feature enabled in VMCB offset 90h

15.34.11 Virtual Transparent Encryption (VTE)

The Virtual Transparent Encryption feature can be enabled to force all memory accesses within an SEV guest to be encrypted with the guest's key. Support for this feature is indicated in CPUID Fn8000_001F[EAX] bit 16.

To enable this feature, the hypervisor must set VMCB offset 90h bit 5. Bit 5 is only observed when SEV (bit 1) is also set to 1 and SEV-ES (bit 2) is cleared to 0. In all other configurations of these bits (namely SEV disabled or SEV-ES enabled), bit 5 is ignored by hardware.

When this feature is enabled, CPU hardware treats the guest C-bit as 1 for all guest memory references. The actual C-bit in the guest page tables is ignored by hardware.

Guest address translation is unchanged, so the guest physical address (without the C-bit) is used for translation in the nested page tables.

15.35 Encrypted State (SEV-ES)

Encrypted VMs that use the SEV feature described in Section 15.34 may additionally use the SEV-ES feature to protect guest register state from the hypervisor. An SEV-ES VM's CPU register state is

encrypted during world switches and cannot be directly accessed or modified by the hypervisor. This is designed to protect against attacks such as exfiltration (unauthorized reading of VM state) and control flow attacks (modifying VM state) including rollback attacks (restoring an earlier VM register state).

SEV-ES includes architectural support for notifying a VM's operating system when certain types of world switches are about to occur, allowing the VM to selectively share information with the hypervisor when needed for functionality.

15.35.1 Determining Support for SEV-ES

SEV-ES support can be determined by reading CPUID Fn8000_001F[EAX] as described in Section 15.34.1. Bit 3 of EAX indicates support for SEV-ES.

15.35.2 Enabling SEV-ES

SEV-ES may be enabled on a per-VM basis by setting bit 2 in offset 90h of the VMCB. When enabling SEV-ES, the hypervisor must also enable SEV (offset 90h bit 1) and LBR virtualization (offset B8h bit 0). Additionally, all other programming requirements related to enabling SEV (see Section 15.34.3) must be satisfied when running an SEV-ES guest.

On some systems, there is a limitation on which ASID values can be used on SEV guests that are run with SEV-ES disabled. While SEV-ES may be enabled on any valid SEV ASID (as defined by CPUID Fn8000_001F[ECX]), there are restrictions on which ASIDs may be used for SEV guests with SEV-ES disabled. CPUID Fn8000_001F[EDX] indicates the minimum ASID value that must be used for an SEV-enabled, SEV-ES-disabled guest. For example, if CPUID Fn8000_001F[EDX] returns the value 5, then any VMs which use ASIDs 1-4 and which enable SEV must also enable SEV-ES.

Note that prior to running an SEV-ES VM for the first time, the hypervisor must coordinate with the AMD Secure Processor to create the initial encrypted state image for the guest VM.

15.35.3 SEV-ES Overview

The SEV-ES architecture is designed to protect guest VM register state by default, and only allow the guest VM itself to grant selective access as required. This additional security protection functionality is accomplished in two ways. First, all VM register state is saved and encrypted when a VM exit event (#VMEXIT) occurs. This state is decrypted and restored on a VMRUN only. Second, certain types of #VMEXIT events cause a new exception to be taken within the guest VM. This new exception (#VC, see Section 15.35.5) indicates that the guest VM performed some action which requires hypervisor involvement, an example of which would be an IO access by the VM. The guest #VC handler is responsible for determining what register state is necessary to expose to the hypervisor for the purpose of emulating this operation. The #VC handler also inspects the returned values from the hypervisor and updates the guest state if the output is deemed acceptable.

Register state that needs to be exposed utilizes a new structure called the Guest-Hypervisor Communication Block (GHCB). The GHCB location is chosen by the guest who maps the page as a shared memory page, thus allowing direct hypervisor access. Only state located in the GHCB can be

read by the hypervisor as all state stored in the traditional VMCB save state structure is encrypted using the guest memory encryption key and integrity protected.

In the #VC handler, the guest may utilize a new instruction (Section 15.35.6) to perform a world switch and invoke the hypervisor. In response to this, the hypervisor can inspect the GHCB and determine the services requested by the guest.

15.35.4 Types of Exits

When SEV-ES is enabled, all #VMEXIT events are classified as either Automatic Exits (AE) or Non-Automatic Exits (NAE). AE events are generally events that occur asynchronously with respect to the guest execution (e.g. interrupts) or events that need not involve exposing any guest register state. All other #VMEXIT events are classified as NAE events, and with NAE events the guest is allowed to determine what register state (if any) to expose in the GHCB. During guest execution, #VMEXIT events (both AE and NAE) are only taken if the corresponding intercept bit in the VMCB control area is set.

The hypervisor is informed of specific AE events exclusively via the #VMEXIT codes within the EXITCODE field of the VMCB control area. NAE events result in a #VC exception which is handled by the guest. Table 15-35 lists the possible AE events, all other events are considered NAE events.

Table 15-35. AE Exitcodes

Code	Name	Notes	HW Advances RIP
52h	VMEXIT_MC	Machine check exception	No
60h	VMEXIT_INTR	Physical INTR	No
61h	VMEXIT_NMI	Physical NMI	No
62h	VMEXIT_SMI	Physical SMI	No
63h	VMEXIT_INIT	Physical INIT	No
64h	VMEXIT_VINTR	Virtual INTR	No
77h	VMEXIT_PAUSE	PAUSE instruction	Yes
78h	VMEXIT_HLT	HLT instruction	Yes
7Fh	VMEXIT_SHUTDOWN	Shutdown	No
8Fh	VMEXIT_EFER_WRITE_TRAP	See section 15.35.10	Yes
90h -9Fh	VMEXIT_CR[0-15]_WRITE_TRAP	See section 15.35.10	Yes
400h	VMEXIT_NPF	Only if PFCODE[3]=0 (no reserved bit error)	No
403h	VMEXIT_VMGEXIT	VMGEXIT instruction	Yes
–1	VMEXIT_INVALID	Invalid guest state	–
–2	VMEXIT_BUSY	Busy bit was set in guest state (see Section 15.36.16)	–

In the case of exits due to specific instructions, the CPU will automatically advance the guest RIP in response to the AE so that execution will resume at the next instruction on a subsequent VMRUN.

In the case of nested page faults, these are treated as AEs only if there was no reserved bit error. This is intended to be used to help distinguish nested page faults due to demand misses (hypervisor needs to allocate a page) vs MMIO emulation (hypervisor needs to emulate a device). Consequently, the hypervisor should set a reserved page table bit, such as a reserved address bit, on all MMIO pages that it intends to emulate. (This can include address bits that may become reserved when SEV is enabled; see Section 15.34.1.) This will ensure that MMIO page faults become NAE events, which is critical so the guest #VC handler can be invoked to assist in the MMIO emulation. Nested page faults that are AE events do not invoke any guest handler and the hypervisor is intended to allocate memory as needed and then resume the guest.

Note that when a guest is running with SEV-ES enabled, instruction bytes (VMCB offset D0h) are never saved to the VMCB on a nested page fault.

15.35.5 #VC Exception

The VMM Communication Exception (#VC) is always generated by hardware when an SEV-ES enabled guest is running and an NAE event occurs. The #VC exception is a precise, contributory, fault-type exception utilizing exception vector 29. This exception cannot be masked. The error code of the #VC exception is equal to the #VMEXIT code (see Appendix C) of the event that caused the NAE.

In response to a #VC exception, a typical flow would involve the guest handler inspecting the error code to determine the cause of the exception and deciding what register state must be copied to the GHCB for the event to be handled. The handler should then execute the VMGEXIT instruction to create an AE and invoke the hypervisor. After a later VMRUN, guest execution will resume after the VMGEXIT instruction where the handler can view the results from the hypervisor and copy state from the GHCB back to its internal state as needed. This flow is shown in Figure 15-29.

Note that it is inadvisable for the hypervisor to set the VMCB intercept bit for the #VC exception as this would prevent proper handling of NAEs by the guest. Similarly, the hypervisor should avoid setting intercept bits for events that would occur in the #VC handler (such as IRET).

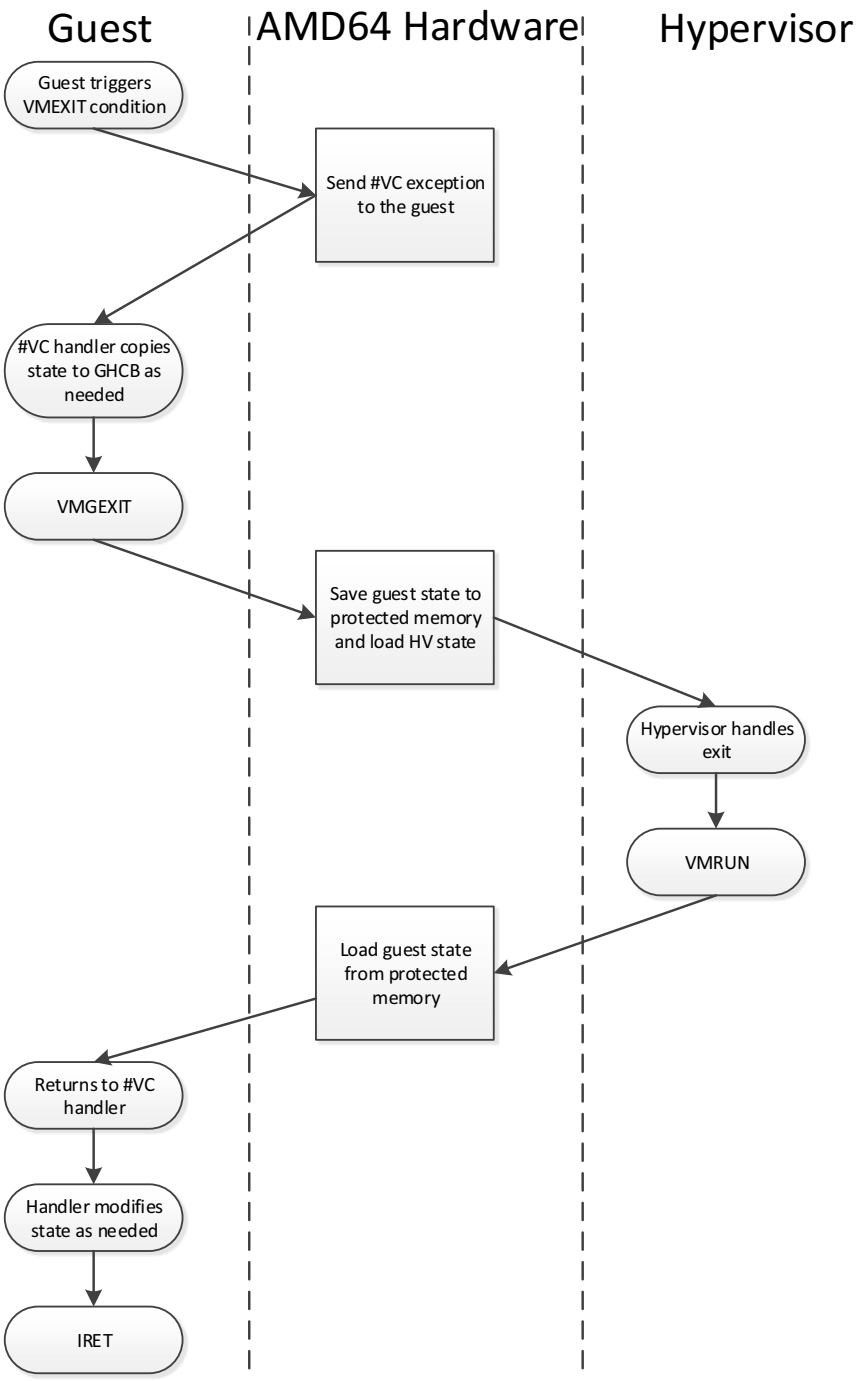


Figure 15-29. EXAMPLE #VC FLOW

15.35.6 VMGExit

The VMGEXIT instruction creates an AE and is intended to allow a guest #VC handler to invoke the hypervisor when needed. VMGEXIT causes an AE with the VMEXIT_VMGEXIT code and behaves like a trap so that upon a subsequent VMRUN, execution resumes following the VMGEXIT. There is no hypervisor intercept bit for VMGEXIT as the instruction unconditionally causes an AE when executed in an SEV-ES guest.

The VMGEXIT opcode is only valid within a guest when run with SEV-ES mode active. If the guest is not run with SEV-ES mode active, the VMGEXIT opcode will be treated as a VMMCALL opcode and will behave exactly like a VMMCALL.

15.35.7 GHCB

The GHCB is an unencrypted memory page used to communicate register state between the SEV-ES guest and the hypervisor. The guest VM is able to set the location of the GHCB via the GHCB MSR (C001_0130). This value is also included in the VMCB and is saved/restored on VMRUN/#VMEXIT respectively.

The GHCB MSR is used to set up the location of the GHCB memory page. The format of this MSR is defined below:

Bit	Function
63:0	Guest physical address of GHCB

The value of this MSR is saved/restored from the VMCB offset 0A0h. It is recommended software write this MSR with a page-aligned address. The GHCB MSR can be read/written only in guest mode, attempts to access this MSR in host mode will result in a #GP.

Hardware never accesses the GHCB directly, and as a result the format of the GHCB is not fixed.

15.35.8 VMRUN

When SEV-ES is enabled, the VM save state area does not reside at offset 400h in the VMCB page. Instead it resides starting at offset 0h in a separate page called the VM Save Area (VMSA) as indicated by the VMSA Pointer at offset 108h. The VMSA Pointer value is stored as a host physical address. Hardware always accesses the VMSA save state area using encrypted memory accesses utilizing the guest's memory encryption key.

When hardware executes a VMRUN instruction and the VMCB indicates SEV-ES is enabled for the guest, the hardware loads guest state from the encrypted save state area indicated by the VMSA Pointer. Also, the VMRUN instruction will perform the following actions in addition to the standard VMRUN behavior:

- Calculate a checksum over guest state to verify integrity
- Perform a VMLOAD to load additional guest register state
- Load guest GPR state

- Load guest FPU state

When a guest has SEV-ES enabled, the encrypted VM state save area definition is expanded to include all GPR and FPU state (see Appendix B). If any part of the VMRUN flow faults or if the integrity checksum fails to match, a #VMEXIT(VMEXIT_INVALID) is generated.

Note that if SEV-ES is enabled, the VMRUN instruction ignores bits 10:5 of the VMCB clean bits and always reloads the full guest state.

Also note that for SEV-ES guests, while the full guest state is loaded on VMRUN only the minimal hypervisor state defined by the legacy VMRUN instruction (see Section 15.5.1) is saved to the host save area. The hypervisor itself should save its desired additional segment state and GPR values to the host save area since these values will be restored by hardware on a subsequent VMEXIT. Hardware does not automatically save host state such as FS, STAR, or GPR values from the hypervisor on a VMRUN. See Appendix B for a detailed breakdown of each piece of VMCB state.

Finally, note that event injection for SEV-ES guests is restricted. Software interrupts and exception vectors 3 and 4 may not be injected. If this is attempted, the VMRUN will fail with a VMEXIT_INVALID error code.

15.35.9 Automatic Exits

When an automatic exit event occurs while an SEV-ES enabled guest is executing, hardware automatically saves guest state to the encrypted save state area and restores hypervisor state from the host save area. Specifically, in addition to the standard state saved/restored by the VMEXIT flow, hardware will also perform the following steps:

- Perform a VMSAVE to save additional guest register state
- Save guest GPR state
- Save guest FPU state
- Calculate and store a checksum over the guest state for use in a subsequent VMRUN
- Perform a VMLOAD to load additional host register state
- Load host GPR state
- Re-initialize FPU state to their reset values

The loading of host GPR state from the host save area is done using the format of the expanded VMCB described in Appendix B. All register state is either loaded from this location or re-initialized to default values so no guest register state is visible to the hypervisor.

15.35.10 Control Register Write Traps

The use of CR[0-15]_WRITE intercepts are discouraged for guests that are run with SEV-ES. These intercepts occur prior to the control register being modified, and the hypervisor is not able to modify the control register itself since the register is located in the encrypted state image. Hypervisors are encouraged to use the new CR[0-15]_WRITE_TRAP and EFER_WRITE_TRAP intercept bits instead which cause an AE after a control register has been modified. These intercepts enable the

hypervisor to track the guest mode and verify if desired features are being enabled. When these traps are taken, the new value of the control register is saved in EXITINFO1. CR write traps are only supported for SEV-ES guests.

Note that writes by SEV-ES guests to EFER.SVME are always ignored by hardware.

15.35.11 Interaction with SMI and #MC

If an SMI occurs while an SEV-ES guest is executing, the platform SMI handler is not immediately executed. Instead, the SMI will remain pending and a #VMEXIT(SMI) is generated. The SMI will then be taken in hypervisor context after STGI is executed. Note that this behavior occurs regardless of the value of the SMI intercept bit in the VMCB.

In some systems, machine check errors are first delivered as an SMI. If this occurs while an SEV-ES guest is executing, #VMEXIT(SMI) will be generated and EXITINFO1[MCREDIR] will be set to 1 (“SMI Intercept” on page 507). As described above, the SMI will be held pending until STGI is executed. After the platform SMI handler executes following STGI, the hypervisor should check the MCREDIR bit to determine if the #VMEXIT(SMI) was due to a machine check error in the guest and handle it appropriately.

15.36 Secure Nested Paging (SEV-SNP)

The SEV-SNP features enable additional protection for encrypted VMs designed to achieve stronger isolation from the hypervisor. SEV-SNP is used with the SEV and SEV-ES features described in Section 15.34 and Section 15.35 respectively and requires the enablement and use of these features.

Primarily, SEV-SNP provides integrity protection of VM memory to help prevent hypervisor-based attacks that rely on guest data corruption, aliasing, replay, and various other attack vectors. To achieve this, a new system-wide data structure called the Reverse Map Table (RMP) is used to perform additional security checks on memory access as described in Section 15.36.3.

In addition to memory protection, SEV-SNP also includes several security features including a new Virtual Machine Privilege Level (VMPL) architecture, interrupt injection restrictions, and side-channel protection. These features are designed to enable additional use models and enhanced security protections.

While this chapter describes the CPU hardware behavior of SEV-SNP, the technology also requires the use of the AMD Secure Processor (AMD-SP) SEV-SNP Application Binary Interface (ABI) to manage the lifecycle events of SEV-SNP VMs. See the SEV-SNP ABI specification (PID#56860) on AMD’s website for more details.

15.36.1 Determining Support for SEV-SNP

Support for SEV-SNP can be determined by reading CPUID Fn8000_001F[EAX] as described in Section 15.34.1. Bit 4 indicates support for SEV-SNP, while bit 5 indicates support for VMPLs. The

number of VMPLs available in an implementation is indicated in bits [15:12] of CPUID Fn8000_001F[EBX].

CPUID Fn8000_001F[EAX] also indicates support for additional security features used with SEV-SNP guests, which are described in the following sections.

15.36.2 Enabling SEV-SNP

SEV-SNP depends on SEV for confidentiality protection. Before enabling SEV-SNP, the MemEncryptionModEn bit in MSR C001_0010 (SYSCFG) must be set, and all programming requirements described in Section 15.34.3 must be satisfied. After SecureNestedPagingEn is set to 1 in MSR C001_0010, certain MSRs may no longer be modified. This includes the fixed range MTRR registers (see Section 7.7.2), the IORR registers (see Section 7.9.2), the TOP_MEM and TOP_MEM2 registers (see Section 7.9.4), as well as the SYS_CFG MSR. Attempts to write the SYS_CFG MSR via WRMSR after SecureNestedPagingEn is set to 1 will be ignored while attempts to write the other MSRs mentioned will result in #GP(0).

Enabling SEV-SNP requires a two-step initialization procedure:

1. Construct the Reverse Map Table (RMP) as described in Section 15.36.4.
2. Set VMPLEn and SecureNestedPagingEn in MSR C001_0010 (SYSCFG) on every core in the system.

After the SEV-SNP feature has been globally enabled, SEV-SNP can be activated on a per-VM basis by setting bit 0 of the SEV_FEATURES field at offset 3B0h of the VMSA during VM creation. SEV-SNP activated VMs must also enable SEV-ES as described in Section 15.35.2 and SEV as described in Section 15.34.3.

In this chapter, the term *SNP-enabled* indicates that SEV-SNP is globally enabled in the SYSCFG MSR. The term *SNP-active* indicates that SEV-SNP is enabled for a specific VM in the SEV_FEATURES field of its VMSA. While SNP-enabled systems support both SNP-active and non-SNP-active VMs, SNP-active VMs can only run on SNP-enabled systems.

15.36.3 Reverse Map Table

The Reverse Map Table (RMP) is a structure shared globally by all logical processors that resides in system memory and is used to ensure a one-to-one mapping between system physical addresses and guest physical addresses. Each page of physical memory that is potentially assignable to guests has one entry within the RMP. RMP entries contain the security attributes of the system physical page as described in Table 15-36.

Table 15-36. Fields of an RMP Entry

Name	Notes
Assigned	Flag indicating that the system physical page is assigned to a guest or to the AMD-SP. 0: Owned by the hypervisor 1: Owned by a guest or the AMD-SP
Page_Size	Encoding of the page size. 0: 4KB page 1: 2MB page
Immutable	Flag indicating that software can alter the entry via x86 RMP manipulation instructions. 0: RMP entry can be altered by software 1: RMP entry cannot be altered by software
Guest_Physical_Address	Guest physical address associated with the page
ASID	ASID of guest to which page is assigned
VMSA	Flag indicating that the page is a VMSA page. 0: Non-VMSA page 1: VMSA page
Validated	Flag indicating that the guest has validated the page. See Section 15.36.6 for details. 0: The guest has not yet validated the page 1: The guest validated the page with PVALIDATE
Permissions[0]	VMPL permission masks for the page. See Section 15.36.7 for details.
...	
Permissions[n-1]	

The integrity of the RMP is maintained by restricting software manipulation of it to the following special-purpose instructions:

- RMPUPDATE: Available to the hypervisor to alter the Guest_Physical_Address, Assigned, Page_Size, Immutable, and ASID fields of an RMP entry. See Section 15.36.5 for details.
- PSMASH: Allows the hypervisor to split a 2MB entry in the RMP into 512 4KB entries in the RMP. See Section 15.36.11 for details.
- RMPADJUST: Allows a guest to alter the VMPL permission masks of the RMP entry. See Section 15.36.7 for details.
- PVALIDATE: Allows a guest to write to the Validated flag in the RMP entry. See Section 15.36.6 for details.

When SEV-SNP is globally enabled, it adds more restrictions to page access controls. The hypervisor and the guests use the above instructions to enforce these restrictions on memory accesses. A violation

of memory access restrictions indicated by the RMP will result in an exception. See Section 15.36.10 for details.

15.36.4 Initializing the RMP

MSR C001_0132 (RMP_BASE) defines the system physical address of the first byte of the RMP. The MSR C001_0133 (RMP_END) defines the system physical address of last byte of the RMP. Software must program RMP_BASE and RMP_END identically for each core in the system and before enabling SEV-SNP globally.

RMP_BASE and (RMP_END+1) must be 8KB aligned. The AMD-SP may place further alignment requirements on these registers. Refer to the latest AMD-SP specifications to determine the required alignment.

The region of memory between RMP_BASE and RMP_END contains a 16KB region used for processor bookkeeping followed by the RMP entries, which are each 16B in size. The size of the RMP determines the range of physical memory that the hypervisor can assign to SNP-active virtual machines at runtime. The RMP covers the system physical address space from address 0h to the address calculated by:

((RMP_END + 1 – RMP_BASE – 16KB) / 16B) x 4KB

For example, if the RMP_BASE is equal to 10_0000h, then to cover the first 4GB of physical memory, RMP_END must be set to 110_3FFFh, which makes the RMP just over 16MB.

Once SEV-SNP is globally enabled, memory accesses are restricted by RMP checks. To ensure that the RMP starts in a known and non-restrictive state, software should write zeros to all memory from RMP_BASE to RMP_END before setting the SecureNestedPagingEn bit in the SYSCFG MSR. The hypervisor then requests the AMD-SP to finalize the initialization of the RMP. The AMD-SP initializes the RMP to prevent all software from directly writing to the memory between RMP_BASE and RMP_END. All subsequent RMP entry manipulation must occur either via the x86 RMP manipulation instructions or through interactions with the AMD-SP.

15.36.5 Hypervisor RMP Management

The hypervisor manages the SEV-SNP security attributes of pages assigned to SNP-active guests by altering the RMP entries of those pages. Because the RMP is initialized by the AMD-SP to prevent direct access to the RMP, the hypervisor must use the RMPUPDATE instruction to alter the entries of the RMP. RMPUPDATE allows the hypervisor to alter the Guest_Physical_Address, Assigned, Page_Size, Immutable, and ASID fields of an RMP entry.

SEV-SNP associates an owner with each system physical page through settings of the Assigned, ASID, and Immutable fields of the page's RMP entry according to Table 15-37. A page can be owned by the hypervisor, a guest, or the AMD-SP.

Table 15-37. RMP Page Assignment Settings

Owner	Assigned	ASID	Immutable
Hypervisor	0	0	-
Guest	1	ASID of the guest	-
AMD-SP	1	0	1

When the hypervisor assigns a page to a guest, it must also set the Guest_Physical_Address and Page_Size to match the nested page table mapping for the guest. If not, access to the page by the guest will result in a fault. See Section 15.36.10 for details on the RMP access checks.

The hypervisor may transition any page that has Immutable set to 0 into a hypervisor-owned page by using RMPUPDATE to set Assigned to 0 and ASID to 0. To transition a page that has Immutable set to 1, the hypervisor must request the AMD-SP to transition the page.

The RMP initialization requirement to write zeros to the RMP (see Section 15.36.4) results in all pages in the system initially belonging to the hypervisor. Any memory pages which are not covered by the RMP are considered permanent hypervisor pages. For example, if the RMP is configured to only cover the first 4GB of memory then all memory above 4GB is considered hypervisor memory for the purpose of RMP access checks.

15.36.6 Page Validation

Each page assigned to a VM is either validated or unvalidated, as indicated by the Validated flag in the page's RMP entry. Memory accesses by the VM to private pages that are unvalidated generate a #VC. All pages are initially assigned as unvalidated.

The VM may use the PVALIDATE instruction to either set or clear the Validated flag of a page. It is expected that VMs would use PVALIDATE to set the Validated flag during VM startup to gain access to the memory the hypervisor has assigned. The VM may later use PVALIDATE to clear the Validated flag if its memory space is being reduced, such as after a memory hot-plug event.

Page validation allows a VM to detect an unexpected remapping of its pages by the hypervisor. Before accessing a page, the VM must validate the page. Once validated, any use of RMPUPDATE by the hypervisor to unassign, reassign, or remap the page will cause the page to become unvalidated. The VM can then detect tampering with the page mapping via the #VC that occurs from accessing unvalidated pages.

PVALIDATE takes a page size as an input parameter indicating that either a 4KB or 2MB page should be validated. If the VM attempts to use PVALIDATE on a 4KB page that is mapped to a 2MB or 1GB page in the nested page table, PVALIDATE generates an #VMEXIT(NPF). In this case, the hypervisor can smash the larger page into 4KB pages using the PSMASH instruction as described in Section 15.36.11. If the VM attempts to use PVALIDATE on a 2MB guest page that is mapped to 4KB nested pages, PVALIDATE returns an error indication to the VM. The VM can instead attempt to execute PVALIDATE for each of the 4KB pages individually.

15.36.7 Virtual Machine Privilege Levels

A typical guest VM may consist of multiple vCPUs. SEV-SNP expands on this capability by enabling the vCPUs to run at distinct Virtual Machine Privilege Levels (VMPLs). Within a vCPU, the different VMPLs are represented by unique VMSAs and are expected to be run in a mutually exclusive manner. Each VMSA is assigned a VMPL as indicated by the VMPL field in the VMSA.

VMPLs are identified numerically starting at 0 with VMPL0 being the most privileged. The number of VMPLs available in an implementation is indicated in bits [15:12] of CPUID Fn8000_001F[EBX]. The VMPL feature enables a guest to sub-divide its address space and implement vCPU-specific access controls on a page-by-page basis.

The processor restricts guest memory accesses based on VMPL permission masks in RMP entries. Each RMP entry contains a set of permission masks, one mask for each implemented VMPL. On memory accesses, the processor checks the current VMPL permission mask of the page to determine whether the access is allowed. The permission mask bits are defined in Table 15-38.

Table 15-38. VMPL Permission Mask Definition

Bit	Name	Settings
0	Read	0: Reads cause #VMEXIT(NPF) 1: Reads are allowed
1	Write	0: Writes cause #VMEXIT(NPF) 1: Writes are allowed
2	Execute-User	0: Execution at CPL 3 causes #VMEXIT(NPF) 1: Execution at CPL 3 is allowed
3	Execute-Supervisor	0: Execution at CPL < 3 causes #VMEXIT(NPF) 1: Execution at CPL < 3 is allowed
4-7	Reserved	SBZ

When a guest access results in a #VMEXIT(NPF) due to a VMPL permission violation, an error code bit in EXITINFO1 is set as described in Section 15.36.10. It is illegal to configure a page with VMPL write permissions but not read permissions. A data access to such a page will result in a #VMEXIT(NPF).

When the hypervisor assigns a page to a guest using RMPUPDATE, full permissions are enabled for VMPL0 and are disabled for all other VMPLs. A VM can then use the RMPADJUST instruction to modify the permissions of VMPLs numerically higher than its own. For example, a vCPU executing at VMPL0 could use RMPADJUST to restrict a page of memory to be only read-write but not executable at VMPL1. However, the vCPU executing at VMPL1 could not alter its own permissions or the permissions of VMPL0.

Further, RMPADJUST cannot be used to grant greater permissions than what is allowed by the permission mask for the current VMPL. For example, if VMPL1 attempts to grant write permission to a page to VMPL2, but VMPL1 does not have write permission to the page, RMPADJUST will fail.

15.36.8 Virtual Top-of-Memory

In the VMSA of an SNP-active guest, the VIRTUAL_TOM field designates a 2MB aligned guest physical address called the virtual top of memory. When bit 1 (vTOM) of SEV_FEATURES is set in the VMSA of an SNP-active VM, the VIRTUAL_TOM field is used to determine the C-bit for data accesses instead of the guest page table contents. All data accesses below VIRTUAL_TOM are accessed with an effective C-bit of 1 and all addresses at or above VIRTUAL_TOM are accessed with an effective C-bit of 0. Note that page table accesses and instruction fetches always have an effective C-bit of 1, regardless of the value of VIRTUAL_TOM or whether the feature is enabled.

When virtual top of memory is enabled in SEV_FEATURES, the C-bit in the guest page table entries must be zero for all accesses. Any guest memory accesses with the C-bit set to 1 in the guest page tables will result in a #PF due to a reserved bit error.

15.36.9 Reflect #VC

When running an SEV-SNP VM, the CPU generates #VC exceptions in response to events that may require hypervisor interaction. #VC exceptions and the events that may lead to them are discussed in Section 15.35.5. SEV-SNP VMs may either chose to handle #VC exceptions directly in their current guest context or turn #VC exceptions into Automatic Exits. This behavior is controlled by bit 2 (ReflectVC) of SEV_FEATURES. If this bit is set to 1, then any event that would otherwise lead to a #VC exception is instead turned into an Automatic Exit.

When a #VC is turned into an Automatic Exit, the guest VM terminates with an exit code of VMEXIT_VC. The error code for the #VC, which reflects the event which led to the #VC (e.g., VMEXIT_CPUID), is saved to the GUEST_EXITCODE field in the VMSA. Additional information about the event that caused the #VC is saved to the GUEST_EXITINFO1, GUEST_EXITINFO2, GUEST_EXITINTINFO, and GUEST_NRIP fields in the VMSA. The information saved to these fields is the same as the standard exit information provided for the event that occurred. For example, if the VM performs a port I/O instruction which is marked for interception, the GUEST_EXITCODE field will be set to VMEXIT_IOIO and the GUEST_EXITINFO1 field will contain information about the I/O port access, as defined in Section 15.10.2.

The Reflect #VC feature enables #VC events to be handled by a vCPU at a VMPL other than the one that initiated them. For example, a guest may contain a vCPU which consists of two different VMSAs. One VMSA is defined to execute at VMPL0, while the other has ReflectVC enabled and is defined to execute at VMPL3. When the vCPU running at VMPL3 encounters a #VC condition, the information is saved to its VMSA and control is returned to the hypervisor. The hypervisor may then run the vCPU at VMPL0 which can read the exit information saved to the VMPL3 VMSA, interact with the hypervisor as required, write appropriate response data back into the VMPL3 VMSA, and instruct the hypervisor to resume execution of the vCPU at VMPL3.

If the #VC event occurred during the processing of an interrupt or exception, the GUEST_EXITINTINFO.V bit will be set. If the Alternate Injection feature is enabled (see Section 15.36.15), hardware will automatically set the VINTR_CTL[BUSY] bit in the VMSA. This enables a higher privileged VMPL to re-inject the event that caused the #VC.

The GUEST_EXITCODE, GUEST_EXITINFO1, GUEST_EXITINFO2, GUEST_EXITINTINFO, and GUEST_NRIP fields are populated by hardware on every Automatic Exit, regardless of the ReflectVC feature. For Automatic Exits other than reflected #VC's, these fields are set to the same values that are set in the unencrypted VMCB.

15.36.10 RMP and VMPL Access Checks

When SEV-SNP is enabled globally, the processor places restrictions on all memory accesses based on the contents of the RMP, whether the accesses are performed by the hypervisor, a legacy guest VM, a non-SNP guest VM or an SNP-active guest VM. The processor may perform one or more of the following checks depending on the context of the access:

- RMP-Covered: Checks that the target page is covered by the RMP. A page is covered by the RMP if its corresponding RMP entry is below RMP_END. Any page not covered by the RMP is considered a Hypervisor-Owned page.
- Hypervisor-Owned: Checks that if the target page is covered by the RMP then the Assigned bit of the target page is 0. If the page table entry that specifies the sPA indicates that the target page size is 2MB, then all RMP entries for the 4KB constituent pages of the target page must have the Assigned bit set to 0. Accesses to 1GB pages only install 2MB TLB entries when SEV-SNP is enabled, therefore this check treats 1GB accesses as 2MB accesses for purposes of this check.
- Guest-Owned: Checks that the ASID field of the RMP entry of the target page matches the ASID of the current VM.
- Reverse-Map: Checks that the Guest_Physical_Address of the RMP entry of the target page matches the guest physical address of the translation.
- Validated: Checks that the Validated field of the RMP entry of the target page is 1.
- Mutable: Checks that the Immutable field of the RMP entry of the target page is 0.
- Page-Size: Checks that the following conditions are met:
  - If the nested page table indicates a 2MB or 1GB page size, the Page_Size field of the RMP entry of the target page is 1.
  - If the nested page table indicates a 4KB page size, the Page_Size field of the RMP entry of the target page is 0.
- VMPL: Checks that the VMPL permission mask allows access. See Section 15.36.7 for details.

Table 15-39 describes under which conditions each check is performed and what fault is produced on failure.

Table 15-39. RMP Memory Access Checks

Host/Guest	SNP-Active	Type of Access	C-Bit	Check	Fault
Host	-	Data write, Page Table Access	-	Hypervisor-Owned	#PF
Guest	No	Data write, Page Table Access	-	Hypervisor-Owned	#VMEXIT(NPF)
Guest	Yes	Instruction Fetch, Page Table Access	-	RMP-Covered, Guest-Owned, Reverse-Map, Mutable, Page-Size	#VMEXIT(NPF)
				Validated	#VC
				VMPL	#VMEXIT(NPF)
Guest	Yes	Data write	0	Hypervisor-Owned	#VMEXIT(NPF)
Guest	Yes	Data write, Data read	1	RMP-Covered, Guest-Owned, Reverse-Map, Mutable, Page-Size	#VMEXIT(NPF)
				Validated	#VC
				VMPL	#VMEXIT(NPF)

In addition, any memory access that results in an RMP check may result in an RMP violation (#PF or #VMEXIT(NPF)) if the accessed RMP entries are in use by other logical processors. In this case, software should retry the access.

If a memory access results in a modification of the Accessed or Dirty bits in a page table entry, this page table modification is treated similarly to data write accesses by SEV-SNP. For any such page table modification access, the page size of the access is inherently 4KB.

If the virtual TOM feature (see Section 15.36.8) is enabled, then the Virtual TOM setting is used to determine the C-bit for a given guest access. Guest physical addresses below Virtual TOM are considered to have a C-bit set to 1.

The following page-fault error bits are set on an RMP check related #PF:

- Bit 31 (RMP): Set to 1 if the fault was caused due to an RMP check or a VMPL check failure, 0 otherwise. All RMP violations described in this section will set this bit to 1.

Additionally, the following page-fault error bits may be set on a #VMEXIT(NPF) in EXITINFO1:

- Bit 34 (ENC): Set to 1 if the guest's effective C-bit was 1, 0 otherwise.

- Bit 35 (SIZEM): Set to 1 if the fault was caused by a size mismatch between PVALIDATE or RMPADJUST and the RMP, 0 otherwise.
- Bit 36 (VMPL): Set to 1 if the fault was caused by a VMPL permission check failure, 0 otherwise.

The effective C-bit is always a 1 on any guest instruction fetch, page table access, or data write to private (C=1) memory.

All RMP checks described in this section occur after page table and nested page table access checks and have lower priority than existing paging checks. Table 15-39 reflects the relative priority of RMP checks. Namely, VMPL checks have the lowest priority, preceded by page validation checks. For example, if a guest access fails the Page-Size check and the Validated check, a #VMEXIT(NPF) will occur instead of a #VC since the Page-Size check has priority over the page validation check.

A failure of the page validation check results in a #VC with error code PAGE_NOT_VALIDATED (0x404). The faulting guest virtual address is saved to CR2 when this error occurs.

15.36.11 Large Page Management

The hypervisor may need to convert a 2MB page assigned to a guest into 4KB pages. This conversion is called page smashing and requires the hypervisor to alter the RMP. The hypervisor can use RMPUPDATE to alter the size of the page in the RMP, but this will clear the validated bit.

To convert a 2MB page into 4KB pages without altering the validated status of the region, the hypervisor may use the PSMASH instruction. PSMASH takes a 2MB aligned system physical address and smashes the page while preserving the Validated bit in the RMP. After PSMASH successfully completes, the RMP entries of the resulting 4KB pages have the following contents:

- Consecutive values in the Guest_Physical_Address fields
- Page_Size set to 0 indicating 4KB pages
- All other RMP fields copied from the original 2MB page RMP entry

One reason the hypervisor may need to smash a 2MB page is if the guest executes PVALIDATE or RMPADJUST on a 4KB page that is backed by a 2MB page. In that case, the instructions generate a #VMEXIT(NPF) with the SIZEM bit set in EXITINFO1. To resolve this, the hypervisor can smash the page and then have the guest restart the instruction.

If the guest wishes to validate a 2MB aligned region, the guest should first attempt to execute PVALIDATE with a size of 2MB. If the page is backed by 4KB pages, PVALIDATE terminates with a FAIL_SIZEMISMATCH error. In this case, the guest should then execute PVALIDATE on each 4KB page individually. This allows the guest to take advantage of the more efficient 2MB mappings and avoid having the hypervisor unnecessarily smash the page.

Table 15-40 summarizes the potential page size mismatches and how to resolve them.

Table 15-40. PVALIDATE/RMPADJUST Page Size Mismatch Combinations

Requested Page Size	Page Size in RMP	Error Condition	Recommended Handling
4KB	2MB	#VMEXIT(NPF)	PSMASH
2MB	4KB	FAIL_SIZEMISMATCH	Guest retries on each 4KB constituent page

The reverse operation of converting a set of consecutive 4KB pages into a single 2MB page requires assistance from either the guest or the AMD-SP to ensure that the operation is safe to perform.

15.36.12 Running SNP-Active Virtual Machines

As with SEV-ES guests, SNP-active guests are described by a hypervisor controlled VMCB and a guest encrypted VMSA. The initial VMSA for an SNP-active guest must be set up through coordination with the AMD-SP, the details of which are beyond the scope of this manual. This includes the initial configuration of the SEV_FEATURES field in the VMSA which indicates which guest security features are enabled for that particular VM instance. VMRUN to an SNP-active guest will fail with a VMEXIT_INVALID error code if SEV-SNP is not globally enabled.

VMRUN Checks. When SEV-SNP is globally enabled on a system, the VMRUN instruction performs additional security checks on various memory pages. These checks are similar to the ones described in Section 15.36.10. Note that where a check depends on page size, a page size of 4KB is used. In addition to the checks described in that section, an additional check exists:

- VMSA: Checks that the VMSA field in the RMP entry equals 1.

The VMSA field in an RMP entry may be set by the AMD-SP, or by a vCPU running at VMPL0 using the RMPADJUST instruction.

The checks performed on VMRUN are as follows:

Table 15-41. VMRUN Page Checks

Page Type	SNP-Active	Check	Fault
VMCB	-	Hypervisor-Owned	#GP(0)
AVIC Backing Page	-	Hypervisor-Owned	#VMEXIT(VMEXIT_INVALID)
VMSA	No	Hypervisor-Owned	#VMEXIT(VMEXIT_INVALID)
VMSA	Yes	RMP-Covered Guest-Owned Reverse-Map Mutable VMSA	#VMEXIT(VMEXIT_INVALID)

The AVIC Logical Table, AVIC Physical Table, IOPM_BASE_PA, MSRPM_BASE_PA, and nCR3 are not checked by VMRUN as these structures are only read by the hardware.

After a successful VMRUN, the VMCB page, as well as any AVIC Backing Page and VMSA Page are marked as in-use by hardware, and any attempt to modify the RMP entries for these pages via instructions like RMPUPDATE will result in a FAIL_INUSE response. The in-use marking is automatically cleared by hardware after a #VMEXIT event.

Other Checks. In addition to the RMP checks performed by VMRUN, a few other VM-related operations perform special RMP checks.

The address written to the VM_HSAVE_PA MSR, which holds the address of the page used to save the host state on a VMRUN, must point to a hypervisor-owned page. If this check fails, the WRMSR will fail with a #GP(0) exception. Note that a value of 0 is not considered valid for the VM_HSAVE_PA MSR and a VMRUN that is attempted while the HSAVE_PA is 0 will fail with a #GP(0) exception.

The VMSAVE instruction also performs checks to ensure that the target page is hypervisor-owned. The VMSAVE instruction is not expected to be used with SEV-ES and SNP-active guests, as described in Section 15.36.8, but may be used with other guests.

If VMSAVE is executed in host mode and the target page fails the RMP check, a #GP(0) exception is generated. If VMSAVE is executed in a guest when the VMSAVE instruction is virtualized (see Section 15.33.1) and the target page fails the RMP check, then a #VMEXIT(NPF) is generated indicating an RMP permission error. In processors that support SEV-SNP, the execution of the VMSAVE instruction inside an SEV-ES or SNP-active guest is not supported and will result in a #VMEXIT(VMSAVE).

Intercept Behavior. All port I/O (IN, INS, OUT, OUTS) and CPUID instructions executed by an SNP-Active guest are treated as intercepted regardless of the intercept bits set in the VMCB and IOPM. Execution of these instructions in an SNP-Active guest will unconditionally generate a Non-Automatic Exit.

15.36.13 Debug Registers

SEV-ES and SNP-active guests may choose to enable full virtualization of CPU debug registers through SEV_FEATURES bit 5 (DebugSwap).

When enabled, the DR[0-3] registers and DR[0-3]_ADDR_MASK registers are swapped as type ‘B’ state (see Appendix B). Note that the DR6 and DR7 registers are always swapped as type ‘A’ state for any SEV guest.

15.36.14 Memory Types

When an SNP-active guest accesses memory, the hardware forces the use of coherent memory types. This prevents the hypervisor from attempting to corrupt guest memory by the use of non-coherent memory types for accesses by the guest.

If a guest memory access is determined to be non-coherent after the memory type determination logic described in Section 15.25.8, the hardware forces a coherent type as described in Table 15-42.

Table 15-42. Non-Coherent Memory Type Conversion

Non-Coherent Memory Type	Forced Coherent Memory Type
UC	CD
WC	WC+

15.36.15 TLB management

For non-SNP-active guests, when a hypervisor moves a VMSA to a new logical processor it must ensure that the VMSA cannot use any stale (incorrect) TLB translations to prevent corruption of the guest. For SNP-active guests, to avoid any dependency on the hypervisor for correctly managing guest TLB contents, the hardware detects when the VMSA is moved and manages the TLB for that guest automatically. The hardware uses two VMSA fields to track this information: the TLB_ID (byte offset 3D0h) and the PCPU_ID (byte offset 3D8h).

During guest creation, software should initialize the TLB_ID and PCPU_ID by setting both to zero. The hardware subsequently manages the values in both fields throughout the lifetime of that VMSA. During operation, software may explicitly write PCPU_ID to 0 to force a TLB flush on the next VMRUN to that VMSA if desired. If this occurs, the hardware will set the PCPU_ID field to a non-zero value when it flushes the TLB.

For example, when guest software performs an RMPADJUST to alter the permissions of a VMPL, it may need to ensure that the existing TLB entries of all vCPUs executing at the targeted VMPL are not used anymore. The guest software can do this by writing zero to PCPU_ID of the affected VMSAs. When these VMSAs are re-entered with VMRUN, the hardware will ensure existing TLB entries are no longer used and set PCPU_ID to a non-zero value. This value can be checked for non-zero by the guest software to ensure the operation has completed before proceeding.

As with any guest, the hypervisor may use the TLB_CONTROL field in the VMCB to force TLB flushes when desired. When the hypervisor writes 3h or 7h to TLB_CONTROL, both global and non-global TLB entries of the guest are invalidated.

15.36.16 Interrupt Injection Restrictions

SNP-active guests may choose to enable the Restricted Injection or Alternate Injection features through SEV_FEATURES bits 3 and 4 respectively. These features enforce additional interrupt and event injection security protections designed to help protect against malicious injection attacks. The two are mutually exclusive for a specific VMSA and an attempt to enable both will result in a #VMEXIT(VMEXIT_INVALID) when the VMSA is run.

Restricted Injection Operation. This feature disables all hypervisor-based interrupt queuing and event injection of all vectors except a new exception vector, #HV (28), which is reserved for SNP guest use, but never generated by hardware. #HV is only allowed to be injected into VMSAs that

execute with Restricted Injection. #HV is a benign exception and can only be injected as an exception (VMCB.EVENTINJ[Type]=3) and without an error code. Guests running with Restricted Injection are expected to communicate with the hypervisor about events via a software-managed para-virtualization interface. This interface can use #HV injection as a doorbell to inform the guest that new events have been added.

The VMRUN instruction with Restricted Injection enabled will fail with a VMEXIT_INVALID error code if the hypervisor attempts the injection of any unsupported event or attempts to run the guest with AVIC enabled.

Alternate Injection Operation. This feature replaces all hypervisor-based interrupt queuing and event injection with guest-controlled queuing and injection. When this is enabled on a VMSA, event injection information on VMRUN is read from the EventInjCtrl field in the VMSA (offset 3E0h) and interrupt queuing information is read from the VIntrCtrl field in the VMSA (offset 3B8h). This feature is intended to be used in a multi-VMPL architecture where a high privilege VMSA injects events and interrupts directly into a low privilege VMSA by writing to its encrypted VMSA.

When Alternate Injection is enabled, the EventInjCtlr field in the unencrypted VMCB (offset A8h) is ignored on VMRUN. The VIntrCtrl field in the unencrypted VMCB (offset 70h) is processed, but only the V_INTR_MASKING, Virtual GIF Mode, and AVIC Enable bits are used. The AVIC Enable bit must be 0 if the guest is running with Alternate Injection enabled, otherwise the VMRUN will fail with a VMEXIT_INVALID error code.

The remaining fields of VIntrCtrl (V_TPR, V_IRQ, VGIF, V_INTR_PRIO, V_IGN_TPR, V_INTR_VECTOR) are read exclusively from the encrypted version in the VMSA. Additionally, bit 10 of the encrypted VIntrCtrl field is defined as the INT_SHADOW bit and the unencrypted INT_SHADOW bit in VMCB offset 68h bit 0 is ignored. On a VMEXIT, the V_TPR, V_IRQ, and INT_SHADOW values are written back to the encrypted VIntrCtrl only.

In guests that run with Alternate Injection, bit 63 of the encrypted VIntrCtrl field is defined as a BUSY bit. On VMRUN, if VIntrCtrl[BUSY] is set to 1, then the VMRUN fails with a VMEXIT_BUSY error code. The BUSY bit enables a VMSA to be temporarily marked non-runnable while software modifications are in progress.

Additional Intercept Behavior. Additional hardware-forced intercept behavior exists in guests that run with either of these features enabled:

- For either feature, hardware treats physical INTR, NMI, INIT, and #MC events as intercepted regardless of the intercept bit set in the VMCB.
- Under Alternate Injection, any MSR access to the x2APIC MSR range (MSR 0x800-0x8FF) by the guest is intercepted regardless of the MSR_PROT intercept and MSR protection bitmap. In this case, the interception behavior is the same as what would occur if the MSR bitmap indicated an interception of the corresponding MSR.

15.36.17 Side-Channel Protection

SEV-SNP provides optional protections against certain side channel attacks.

Branch Target Buffer Isolation

SNP-active guests may choose to enable the Branch Target Buffer Isolation mode through SEV_FEATURES bit 7 (BTBIsolation). The Branch Target Buffer (BTB) is an internal CPU structure that is used when predicting indirect branches, and SNP-active guests may choose to impose additional restrictions on it in order to help prevent certain types of speculative execution-based side channels.

When executing an SNP-active guest when BTB Isolation is enabled, CPU hardware will ensure that no code outside of that guest context is able to influence the BTB-based predictions performed by hardware within the guest. Hardware tracks the source of prediction information in the BTB and may flush BTB contents when required to maintain this isolation.

In hardware that supports BTB Isolation, new BTB prediction information is never written if SPEC_CTRL[IBRS] is enabled in the current context. Therefore, it is recommended that non-guest software that executes temporarily (e.g., hypervisor exit handling code) run with SPEC_CTRL[IBRS] set to 1. This ensures that indirect branch information from that context is not stored in the BTB and may avoid the need for a BTB flush when guest execution is resumed.

Instruction Based Sampling

SEV-ES and SNP-active guests may choose to disallow the use of Instruction Based Sampling (IBS) by the hypervisor in order to limit the information that may be gathered about their execution. Guests may enable this restriction through SEV_FEATURES bit 6 (PreventHostIBS). When a VMRUN is executed on a guest that has enabled this protection, the IbsFetchCtl[IbsFetchEn] and IbsOpCtl[IbsOpEn] MSR bits must be 0. If either of these bits are not 0 then the VMRUN will fail with a VMEXIT_INVALID error code.

VMSA Register Protection

SNP-active guests may choose to enable the VMSA Register Protection feature through SEV_FEATURES bit 14 (VmsaRegProt). When an Automatic Exit occurs and this feature is enabled, CPU hardware will obfuscate the values of certain registers before they are written to the VMSA. The hardware will then deobfuscate these values on a subsequent VMRUN. This obfuscation may help prevent certain types of side channel attacks on the encrypted VMSA ciphertext.

The obfuscation is performed with a bit-wise XOR operation between the register value and an 8B nonce. The nonce value is stored in the VMSA and is updated in a pseudo-random manner by the CPU hardware on every Automatic Exit. When initializing a new VMSA, it is recommended that the nonce is set to a random value.

The specific VMSA fields which are obfuscated when this feature is enabled may vary by implementation. More information about this may be found in the AMD-SP SEV-SNP ABI specification.

15.36.18 Secure TSC

SNP-active guests may choose to enable the Secure TSC feature through SEV_FEATURES bit 9 (SecureTscEn). When enabled, Secure TSC changes the guest view of the Time Stamp Counter when read by the guest via either the TSC MSR, RDTSC, or RDTSCP instructions. The TSC value is first scaled with the GUEST_TSC_SCALE value from the VMSA and then is added to the VMSA GUEST_TSC_OFFSET value. The P0 frequency, TSC_RATIO (C001_0104h) and TSC_OFFSET (VMCB offset 50h) values are not used in the calculation.

The GUEST_TSC_SCALE is an 8.32 fixed point binary number which is composed of 8 bits of integer and 32 bits of fraction. The AMD-SP SEV-SNP ABI specification provides additional information about the Secure TSC feature and initialization of GUEST_TSC_SCALE.

Guests that run with Secure TSC enabled may read the GUEST_TSC_FREQ MSR (C001_0134h) which returns the effective frequency in MHz of the guest view of TSC. This MSR is read-only and attempting to write the MSR or read it when outside of a guest with Secure TSC enabled causes a #GP(0) exception.

Guests that run with Secure TSC enabled are not expected to perform writes to the TSC MSR (10h). If such a write occurs, subsequent TSC values read are undefined.

15.37 SPEC_CTRL Hypervisor Model

A hypervisor may wish to impose speculation controls on guest execution or a guest may want to impose its own speculation controls. Therefore, the processor implements both host and guest versions of SPEC_CTRL. The presence of this feature is indicated by CPUID Fn8000_000A_EDX[20]=1.

When in host mode, the host SPEC_CTRL value is in effect and writes update only the host version of SPEC_CTRL. On a VMRUN, the processor loads the guest version of SPEC_CTRL from the VMCB. For most guests, processor behavior is controlled by the logical OR of the two registers. When the guest writes SPEC_CTRL, only the guest version is updated. On a VMEXIT, the guest version is saved into the VMCB and the processor returns to only using the host SPEC_CTRL for speculation control.

SNP-active guests with BTB isolation enabled behave differently (see section 15.36.17). For these guests the host value of IBRS does not override the guest value.

16 Advanced Programmable Interrupt Controller (APIC)

The Advanced Programmable Interrupt Controller (APIC) provides interrupt support on AMD64 architecture processors. The local APIC accepts interrupts from the system and delivers them to the local CPU core interrupt handler.

Support for APIC is indicated by CPUID Fn0000_0001_EDX[APIC] = 1. For information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 71.

The APIC block diagram is provided in Figure 16-1.

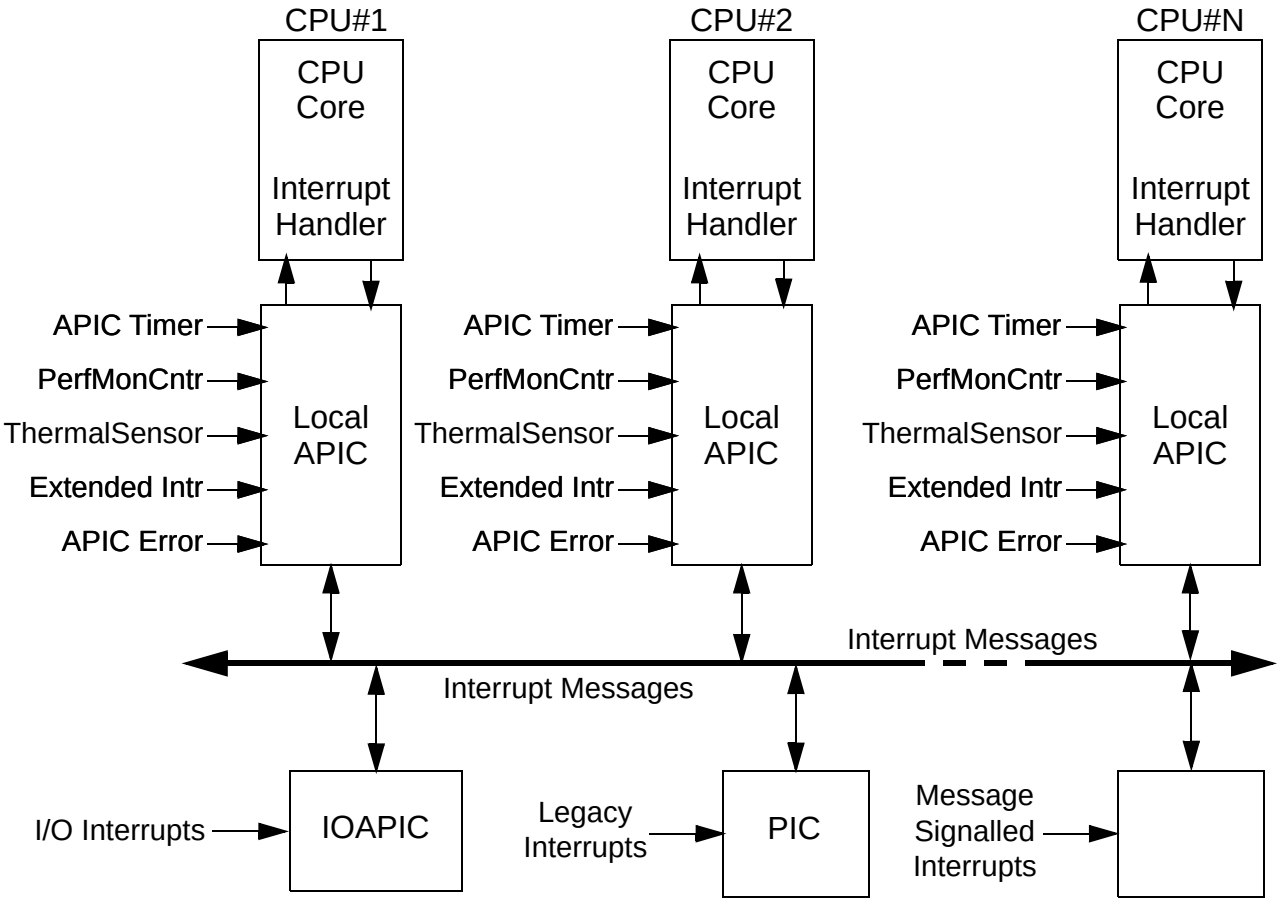


Figure 16-1. Block Diagram of a Typical APIC Implementation

16.1 Sources of Interrupts to the Local APIC

Each CPU core has an associated local APIC which receives interrupts from the following sources:

- I/O interrupts from the IOAPIC interrupt controller (including LINT0 and LINT1)
- Legacy interrupts (INTR and NMI) from the legacy interrupt controller
- Message Signalled Interrupts
- Interprocessor Interrupts (IPIs) from other local APICs. Interprocessor Interrupts are used to send interrupts or to execute system wide functions between CPU cores in the system, including the originating CPU core (self-interrupt).
- Locally generated interrupts within the local APIC. The local APIC receives local interrupts from the APIC timer, Performance Monitor Counters, thermal sensors, APIC errors and extended interrupts from implementation specific sources.

The sources of interrupts for the local APIC are provided in Table 16-1.

Table 16-1. Interrupt Sources for Local APIC

Source	Description	Message Type to Local APIC
I/O interrupts	System interrupts from I/O devices or system hardware received through the I/O APIC and sent to the local APIC as interrupt messages. They may be edge-triggered or level-sensitive.	Fixed, Lowest Priority, SMI, NMI, INIT, STARTUP, External interrupt, LINT0, LINT1
Legacy Interrupts	Legacy interrupts (INT and NMI) from the PIC and sent to the local APIC as interrupt messages.	NMI, INT
Interprocessor (IPI)	Interprocessor interrupts. Used for interrupt forwarding, system-wide functions, or software self-interrupts.	Fixed, lowest priority, SMI, read request, NMI, INIT, STARTUP, External interrupt
APIC Timer	Local interrupt from the programmed APIC timer reaches zero, under control of TIMER_LVT.	Fixed
Performance Monitor Counter	Local interrupt from the performance monitoring counter when it overflows, under control of PERF_CNT_LVT.	Fixed, SMI, or NMI
Thermal Sensor	Local interrupt from internal thermal sensors when it has tripped, under control of THERMAL_LVT.	Fixed, SMI, or NMI
Extended Interrupt[3:0]	Local Interrupts from programmable internal CPU core sources, under the control of the EXTENDED_INTERRUPT[3:0]_LVT.	Fixed, SMI, NMI, or External interrupt
APIC Internal Error	Local interrupt when an error is detected within the local APIC, under control of ERROR_LVT.	Fixed, SMI, or NMI

16.2 Interrupt Control

I/O, legacy, and interprocessor interrupts are sent via interrupt messages. The interrupt messages contain the following information:

- Destination address of the local APIC.
- VECTOR[7:0] indicating interrupt priority of up to 256 interrupt vectors. This information is captured in the IRR register for Fixed and Lowest Priority interrupt message types.
- Trigger Mode indicating edge triggered or level-sensitive (which requires an EOI response to the source).
- Message Type[3:0] indicating the type of interrupt to be presented to the local APIC. For Fixed and Lowest Priority message types, the interrupt is processed through the target local APIC. For all other message types, the interrupt is sent directly to the destination CPU core. There is a 5-line interrupt interface to the CPU core for INTR, SMI, NMI, INIT and STARTUP interrupts. For locally-generated interrupts, control is provided by local vector tables or LVTs. Separate LVTs are provided for each interrupt source, allowing for a unique entry point for each source. The LVT contains the VECTOR[7:0], trigger mode and message type as well as other fields associated with the specific interrupt. The message type may be Fixed, SMI, NMI, or External interrupt. A Mask bit is also provided to mask the interrupt.

16.3 Local APIC

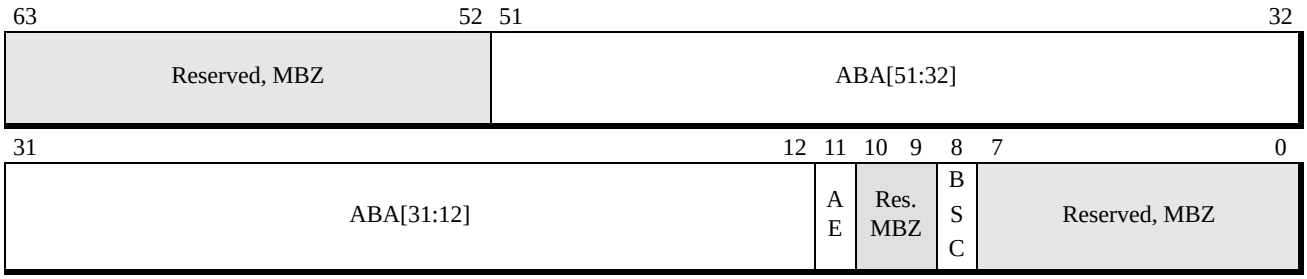
16.3.1 Local APIC Enable

The local APIC is controlled by the APIC enable bit (AE) in the APIC Base Address Register (MSR 0000_001Bh). See Figure 16-2 on page 604.

When AE is set to 1, the local APIC is enabled and all interrupt types are accepted. When AE is cleared to 0, the local APIC is disabled, including all local vector table interrupts.

Software can disable the local APIC, using the APIC_SW_EN bit in the Spurious Interrupt Vector Register (APIC_F0). When this bit is cleared to zero, the local APIC is temporarily disabled:

- SMI, NMI, INIT, Startup, and Remote Read interrupts may be accepted.
- Pending interrupts in the ISR and IRR are held.
- Further fixed, lowest-priority, and ExtInt interrupts are not accepted.
- All LVT entry mask bits are set and cannot be cleared.



Bits	Mnemonic	Description	Access Type
63:52	—	Reserved, MBZ	
51:12	ABA	APIC Base Address	R/W
11	AE	APIC Enable	R/W
8	BSC	Boot Strap CPU Core	RO
7:0	Reserved	Reserved, Must be Zero	

Figure 16-2. APIC Base Address Register (MSR 0000_001Bh)

The fields within the APIC Base Address register are as follows:

- *Boot Strap CPU Core (BSC)*—Bit 8. The BSC bit indicates that this CPU core is the boot core of the BSP. Each CPU core that is not the boot core of the boot processor is an AP (Application Processor).
- *APIC Enable (AE)*—Bit 11. This is the APIC enable bit. The local APIC is enabled and all interruption types are accepted when AE is set to 1. Clearing AE to 0 disables the local APIC, and no local vector table interrupts are supported.
- *APIC Base Address (ABA)*—Bits 51:12. Specifies the base physical address for the APIC register set. The address is extended by 12 bits at the least-significant end to form the 52-bit physical base address. The reset value of the APIC base address is 0_0000_FEE0_0000h. This address is not affected by INIT.

Note that a given processor may implement a physical address less than 52 bits in length.

16.3.2 APIC Registers

The system programming interface of the local APIC is made up of the registers listed in Table 16-2 below. All APIC registers are memory-mapped into the 4-Kbyte APIC register space, and are accessed with memory reads and writes. The memory address is indicated as:

APIC Register address = APIC Base Address + Offset

where the APIC Base Address must point to an uncacheable memory region, and is located in APIC Base Address Register, MSR 0000_001Bh. See Figure 16-2.

APIC registers are aligned to 16-byte offsets and must be accessed using naturally-aligned DWORD size read and writes. All other accesses cause undefined behavior.

The table includes the value of each register after reset and INIT.

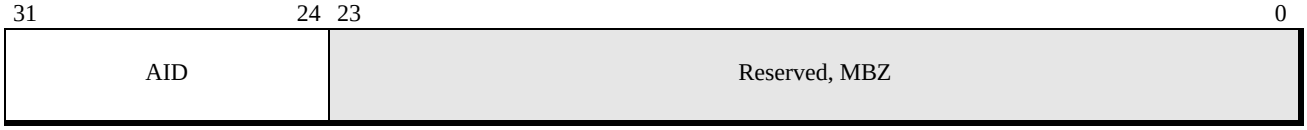
Table 16-2. APIC Registers

Offset	Name	Reset
20h	APIC ID Register	??000000h
30h	APIC Version Register	80??0010h
80h	Task Priority Register (TPR)	00000000h
90h	Arbitration Priority Register (APR)	00000000h
A0h	Processor Priority Register (PPR)	00000000h
B0h	End of Interrupt Register (EOI)	–
C0h	Remote Read Register	00000000h
D0h	Logical Destination Register (LDR)	00000000h
E0h	Destination Format Register (DFR)	FFFFFFFF
F0h	Spurious Interrupt Vector Register	000000FFh
100-170h	In-Service Register (ISR)	00000000h
180-1F0h	Trigger Mode Register (TMR)	00000000h
200-270h	Interrupt Request Register (IRR)	00000000h
280h	Error Status Register (ESR)	00000000h
300h	Interrupt Command Register Low (bits 31:0)	00000000h
310h	Interrupt Command Register High (bits 63:32)	00000000h
320h	Timer Local Vector Table Entry	00010000h
330h	Thermal Local Vector Table Entry	00010000h
340h	Performance Counter Local Vector Table Entry	00010000h
350h	Local Interrupt 0 Vector Table Entry	00010000h
360h	Local Interrupt 1 Vector Table Entry	00010000h
370h	Error Vector Table Entry	00010000h
380h	Timer Initial Count Register	00000000h
390h	Timer Current Count Register	00000000h
3E0h	Timer Divide Configuration Register	00000000h
400h	Extended APIC Feature Register	00040007h
410h	Extended APIC Control Register	00000000h
420h	Specific End of Interrupt Register (SEOI)	–
480-4F0h	Interrupt Enable Registers (IER)	FFFFFFFFh
500-530h	Extended Interrupt [3:0] Local Vector Table Registers	00000000h

16.3.3 Local APIC ID

Unique local APIC IDs are assigned to each CPU core in the system. The value is determined by hardware, based on the number of CPU cores on the processor and the node ID of the processor.

The APIC ID is located in the APIC ID register at APIC offset 20h. See Figure 16-3. It is model dependent, whether software can modify the APIC ID Register. The initial value of the APIC ID (after a reset) is the value returned in CPUID function 0000_0001h_EBX[31:24].



Bits	Mnemonic	Description	R/W
31:24	AID	APIC ID	R/W
23:0	—	Reserved, MBZ	

Figure 16-3. APIC ID Register (APIC Offset 20h)

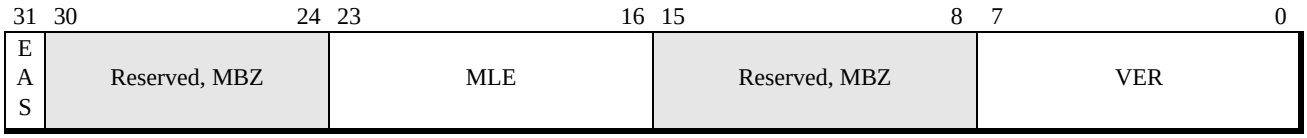
- *APIC ID (AID)*—Bits 31:24. The APIC ID field contains the unique APIC ID value assigned to this specific CPU core. A given implementation may use some bits to represent the CPU core and other bits represent the processor.

16.3.4 APIC Version Register

A version register is provided to allow software to identify which APIC version is used. Bits 7:0 of the APIC Version Register indicate the version number of the APIC implementation.

The number of entries in the local vector table are specified in bits 23:16 of the register as the maximum number minus one.

Bit 31 indicates the presence of extended APIC registers which have an offset starting at 400h.



Bits	Mnemonic	Description	R/W
31	EAS	Extended APIC Register Space Present	RO
30:24	—	Reserved, MBZ	
23:16	MLE	Max LVT Entries	RO
15:8	—	Reserved, MBZ	
7:0	VER	Version	RO

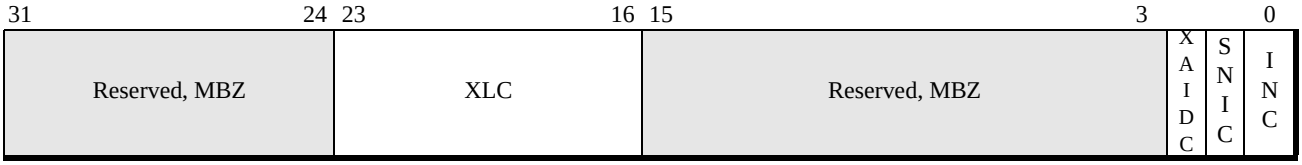
Figure 16-4. APIC Version Register (APIC Offset 30h)

The fields within the APIC Version register are as follows:

- *Version (VER)*—Bits 7:0. The VER field indicates the version number of the APIC implementation. The local APIC implementation is identified with a value=1Xh (20h-FFh are reserved).
- *Max LVT Entries (MLE)*—Bits 23:16. The MLE field specifies the number of entries in the local vector table minus one.
- *Extended APIC Register Space Present (EAS)*—Bit 31. The EAS bit when set to 1 indicates the presence of an extended APIC register space, starting at offset 400h.

16.3.5 Extended APIC Feature Register

The Extended APIC Feature Register indicates the number of extended Local Vector Table registers in the local APIC, whether the Interrupt Enable Registers are present, and whether the 8-bit Extended APIC ID and Specific End Of Interrupt (SEOI) Register are supported.



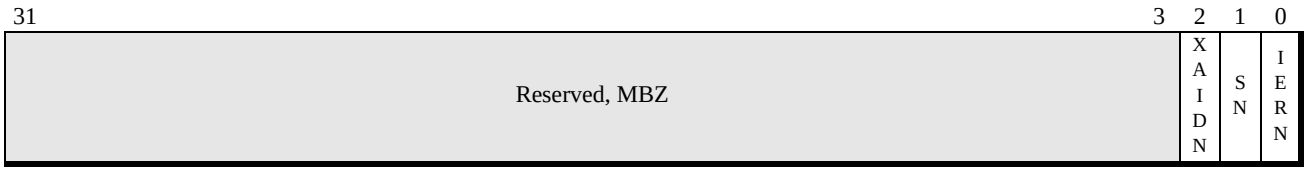
Bits	Mnemonic	Description	R/W
31:24	Reserved	Reserved, Must be Zero	
23:16	XLC	Extended LVT Count	RO
15:3	Reserved	Reserved, Must be Zero	
2	XAIDC	Extended APIC ID Capable	RO
1	SNIC	Specific End of Interrupt Capable	RO
0	INC	Interrupt Enable Register Capable	RO

Figure 16-5. Extended APIC Feature Register (APIC Offset 400h)

- Extended LVT Count (XLC)—(Bits 23:16) Specifies the number of extended local vector table registers in the local APIC.
- Extended APIC ID Capability (XAIDC)—(Bit 2) Indicates that the processor is capable of supporting an 8-bit APIC ID.
- Specific End of Interrupt Capable—(Bit 1) Indicates that the Specific End Of Interrupt Register is present.
- Interrupt Enable Register Capable—(Bit 0) Read-only. Indicates that the Interrupt Enable Registers are present.

16.3.6 Extended APIC Control Register

This bit enables writes to the interrupt enable registers.



Bits	Mnemonic	Description	R/W
31:3	—	Reserved, MBZ	
2	XAIDN	Extended APIC ID Enable.	R/W
1	SN	Enable SEOI Generation	R/W
0	IERN	Enable Interrupt Enable Registers	R/W

Figure 16-6. Extended APIC Control Register (APIC Offset 410h)

- Extended APIC ID Enable (XAIDN)—Bit 2. Setting XAIDN to 1 enables the upper four bits of the APIC ID field described in “APIC ID Register (APIC Offset 20h)” on page 606. Clearing this bit, specifies a 4-bit APIC ID using only the lower four bits of the APIC ID field of the APIC ID register.
- Enable SEOI Generation (SN)—Bit 1. Read-write. This bit enables Specific End of Interrupt (SEOI) generation when a write to the specific end of interrupt register is received.
- Enable Interrupt Enable Registers (IERN)—Bit 0. This bit enables writes to the interrupt enable registers.

16.4 Local Interrupts

The local APIC handles the following local interrupts:

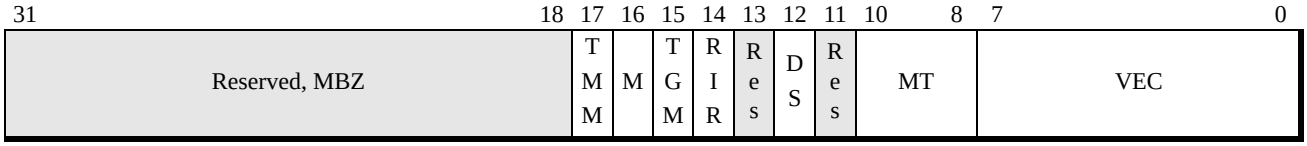
- APIC Timer
- Local Interrupt 0 (LINT0)
- Local Interrupt 1 (LINT1)
- Performance Monitor Counters
- Thermal Sensors
- APIC internal error
- Extended (Implementation dependent)

A separate entry in the local vector table is provided for each interrupt to allow software to specify:

- Whether the interrupt is masked or not.
- The delivery status of the interrupt.
- The message type.
- The unique address vector.
- For LINT0 and LINT1 interrupts, the trigger mode, remote IRR, and input pin polarity.

- For the APIC timer interrupt, the timer mode.

The general format of a Local Vector Table Register is shown in Figure 16-7.



Bits	Mnemonic	Description	R/W
31:18	—	Reserved, MBZ	
17	TMM	Timer Mode	R/W
16	M	Mask	R/W
15	TGM	Trigger Mode	R/W
14	RIR	Remote IRR	RO
13	—	Reserved, MBZ	
12	DS	Delivery Status	RO
11	—	Reserved, MBZ	
10:8	MT	Message Type	R/W
7:0	VEC	Vector	R/W

Figure 16-7. General Local Vector Table Register Format

The fields within the General Local Vector Table register are as follows:

- *Vector (VEC)*—Bits 7:0. The VEC field contains the vector that is sent for this interrupt source when the message type is fixed. It is ignored when the message type is NMI and is set to 00h when the message type is SMI. Valid values for the vector field are from 16 to 255. A value of 0 to 15 when the message type is fixed results in an illegal vector APIC error.
- *Message Type (MT)*—Bits 10:8. The MT field specifies the delivery mode sent to the CPU core interrupt handler. The legal values are:
  - 000b = Fixed - The vector field specifies the interrupt delivered.
  - 010b = SMI - An SMI interrupt is delivered. In this case, the vector field should be set to 00h.
  - 100b = NMI - A NMI interrupt is delivered with the vector field being ignored.
  - 111b = External interrupt is delivered.
- *Delivery Status (DS)*—Bit 12. The DS bit indicates the interrupt delivery status. The DS bit is set to 1 when the interrupt is pending at the CPU core interrupt handler. After a successful delivery of the interrupt, the associated bit in the IRR is set and this bit is cleared to zero. See Section 16.6.2, “Lowest Priority Messages and Arbitration,” on page 620 for details. The bit is cleared to 0 when the interrupt is idle.
- *Remote IRR (RIR)*—Bit 14. The RIR bit is set to 1 when the local APIC accepts an LINT0 or LINT1 interrupt with the trigger mode=1 (level sensitive). The bit is cleared to 0 when the interrupt completes, as indicated when an EOI is received.

- *Trigger Mode (TGM)*—Bit 15. Specifies how interrupts to the local APIC are triggered. The TGM bit is set to 1 when the interrupt is level-sensitive. It is cleared to 0 when the interrupt is edge-triggered. When the message type is SMI or NMI, the trigger mode is edge triggered.
- *Mask (M)*—Bit 16. When the M bit is set to 1, reception of the interrupt is disabled. When the M bit is cleared to 0, reception of the interrupt is enabled.
- *Timer Mode (TMM)*—Bit 17. Specifies the timer mode for the APIC Timer interrupt. The TMM bit set to 1 indicates periodic timer interrupts. The TMM bit cleared to 0 indicates one-shot operation.

16.4.1 APIC Timer Interrupt

The APIC timer is a programmable 32-bit counter used by software to time operations or events. The timer can operate in two modes, periodic and one-shot, under the control of bit 17 (Timer Mode) in APIC Timer Local Vector Table Register (see Figure 16-8). In both modes, the APIC timer is set to a programmable initial value and starts to decrement at a programmable clock rate. When the Initial Count Register is written to a non-zero value, the APIC timer is initialized to the value just written and starts decrementing. When the Initial Count Register is written to zero, the APIC timer is initialized to zero and stops decrementing. In one-shot mode, the APIC timer stops counting when the timer reaches zero. In periodic mode, the APIC timer is initialized again when it reaches zero, and it starts to decrement again. Whenever the timer value is decremented to zero, an APIC timer interrupt is generated under the control of bit 16 (Mask) in the APIC Timer Local Vector Table Register.

To avoid race conditions, software should initialize the Divide Configuration Register and the Timer Local Vector Table Register prior to writing the Initial Count Register to start the timer.

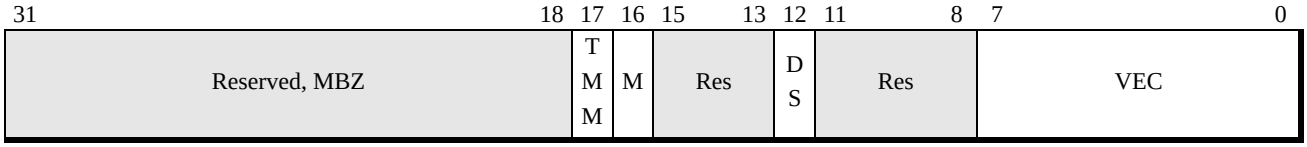


Figure 16-8. APIC Timer Local Vector Table Register (APIC Offset 320h)

Three APIC registers are defined for the APIC timer function:

- Current Count Register (CCR) is the actual APIC timer. Whenever the ICR is written to a non-zero value, or when the CCR reaches zero while in periodic mode, it is initialized to a start count loaded from the ICR and then decrements. The APIC timer interrupt is generated when the CCR value reaches zero. The counting rate is controlled by the DCR. See Figure 16-9.
- Initial Count Register (ICR) provides the initial value for the APIC timer. See Table 16-10.
- Divide Configuration Register (DCR) controls the counting rate of the APIC timer by dividing the CPU core clock by a programmable amount. See Figure 16-11. For the specific details on the implementation of the APIC timer base clock rate, see the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.

Bits	Mnemonic	Description	R/W
31:0	APICTCC	APIC Timer Current Count	RO

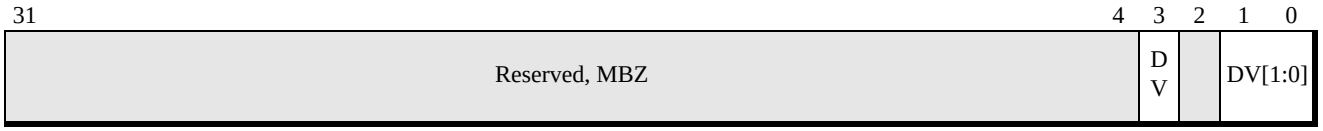
Figure 16-9. Timer Current Count Register (APIC Offset 390h)

- *APIC Timer Current Count (APICTCC)*—Bits 31:0. The APICTCC field contains the current value of the APIC timer.

Bits	Mnemonic	Description	R/W
31:0	APICTIC	APIC Timer Initial Count	R/W

Figure 16-10. Timer Initial Count Register (APIC Offset 380h)

- *APIC Timer Initial Count (APICTIC)*—Bits 31:0. The APICTIC field contains the value that is loaded into the APIC Timer Current Count Register when the APIC timer is initialized.



Bits	Mnemonic	Description	R/W
31:4	—	Reserved, MBZ	
3	DV[2]	Divide Value[2]	R/W
2	—	Reserved, MBZ	
1:0	DV[1:0]	Divide Value[1:0]	R/W

Figure 16-11. Divide Configuration Register (APIC Offset 3E0h)

- *Divide Value (DV)*—Bits 3, 1:0. The DV field specifies the value of the CPU core clock divisor. Table 16-3 lists the allowable values.

Table 16-3. Divide Values

Bits 3, 1:0	Resulting Timer Divide
000b	Divide by 2
001b	Divide by 4
010b	Divide by 8
011b	Divide by 16
100b	Divide by 32
101b	Divide by 64
110b	Divide by 128
111b	Divide by 1

16.4.2 Local Interrupts LINT0 and LINT1

When the target local APIC receives an interrupt message from an IOAPIC with the LINT0 or LINT1 message type, the appropriate local interrupt is generated under the control of bit 16 (Mask) in the APIC LINT0 or LINT1 Local Vector Table Register. See Figure 16-12.

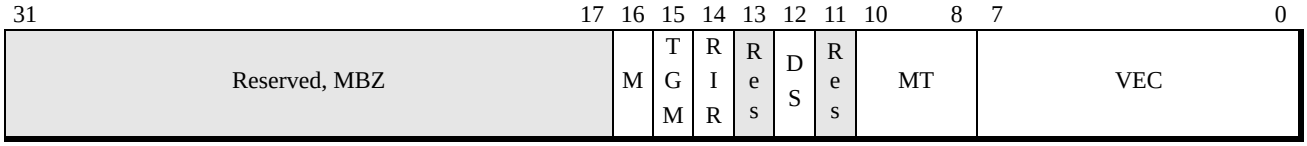


Figure 16-12. Local Interrupt 0/1 (LINT0/1) Local Vector Table Register (APIC Offset 350h/360h)

In addition to the normal LVT control bits (mask, delivery status and vector offset), the LINT0/LINT1 interrupts provide the following controls:

- Trigger Mode - indicates whether the interrupt pin is edge triggered or level sensitive when the message type is fixed.
- Remote IRR - When the trigger mode indicates level, this flag is set when the local APIC accepts the interrupt, and is reset when the local APIC receives an EOI. When the flag is set, no additional local interrupt requests are sent to the local APIC, and they remain pending.

16.4.3 Performance Monitor Counter Interrupts

When a performance monitor counter overflows, an APIC interrupt is generated under the control of bit 16 (Mask) in the APIC Performance Monitor Counter Local Vector Table Register. See Figure 16-13 on page 613.

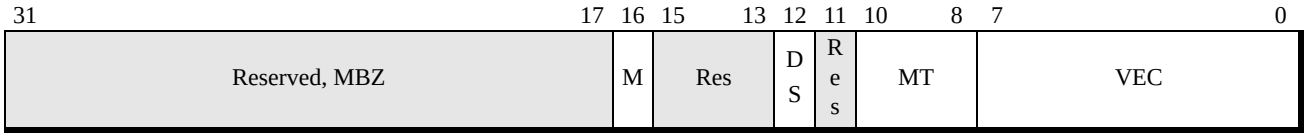


Figure 16-13. Performance Monitor Counter Local Vector Table Register (APIC Offset 340h)

16.4.4 Thermal Sensor Interrupts

When a thermal event occurs, an APIC interrupt is generated under the control of bit 16 (Mask) in the APIC Thermal Sensor Local Vector Table Register. See Figure 16-14. See the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product for more information on thermal events. This interrupt may not be supported in all implementations.

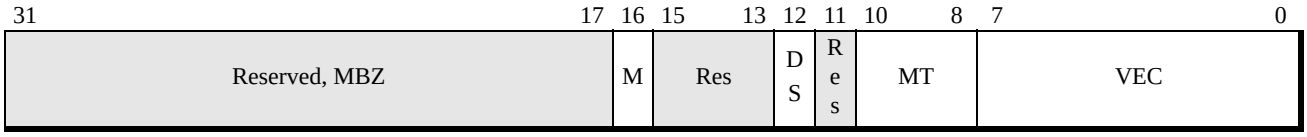


Figure 16-14. Thermal Sensor Local Vector Table Register (APIC Offset 330h)

16.4.5 Extended Interrupts

The local interrupts are extended to include more LVT registers, to allow additional interrupt sources. The additional sources are model dependent and can include:

- Counter overflow from the Machine Check Miscellaneous Threshold Register. See "Machine-Check Miscellaneous-Error Information Register 0 (MC*i*_MISC0)" on page 302 for details.
- ECC Error Count Threshold in memory system.
- Instruction Sampling.

The LVT register used for each interrupt source is specified by the control register associated with the source.

The Extended LVT Count field (bits 23:16) of the Extended APIC Feature Register specifies the number of extended LVT registers. Currently there are four additional LVT registers defined, Extended Interrupt [3:0], Local Vector Table Register, located at APIC offsets 500h–530h. (See Section 16.7.1, "Specific End of Interrupt Register," on page 627 and Figure 16-5 on page 607.)

16.4.6 APIC Error Interrupts

Errors that are detected while handling interrupts cause an APIC error interrupt to be generated under the control of bit 16 (Mask) in the APIC Error Local Vector Table Register. See Figure 16-15 on page 614.

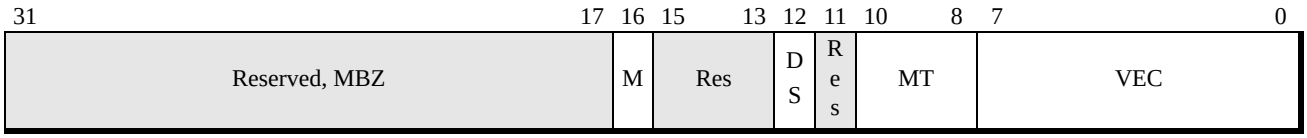
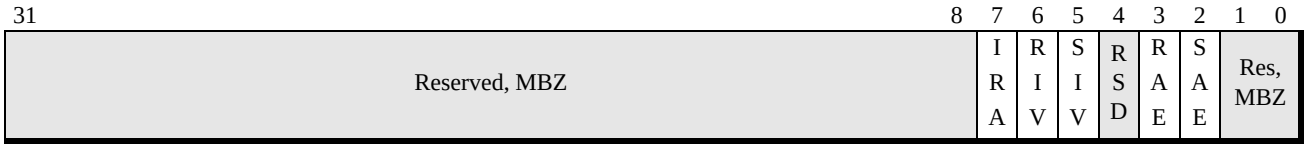


Figure 16-15. APIC Error Local Vector Table Register (APIC Offset 370h)

The error information is recorded in the APIC Error Status Registers. The APIC Error Status Register is a read-write register. Writes to the register cause the internal error state to be recorded in the register, clearing the original error. See Figure 16-16.



Bits	Mnemonic	Description	R/W
31:8	—	Reserved, MBZ	
7	IRA	Illegal Register Address	R/W
6	RIV	Received Illegal Vector	R/W
5	SIV	Sent Illegal Vector	R/W
4	—	Reserved, MBZ	
3	RAE	Receive Accept Error	R/W
2	SAE	Sent Accept Error	R/W
1:0	—	Reserved, MBZ	

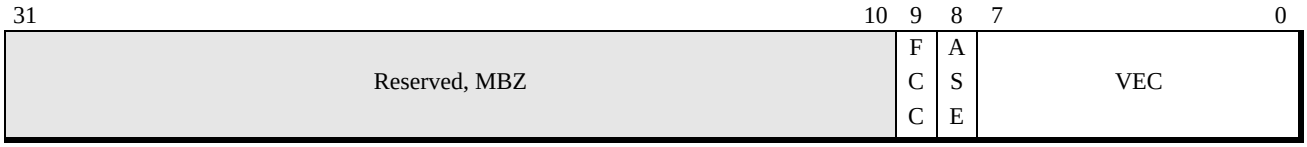
Figure 16-16. APIC Error Status Register (APIC Offset 280h)

The fields within the APIC Error Status register are as follows:

- *Sent Accept Error (SAE)*—Bit 2. The SAE bit when set to 1 indicates that a message sent by the local APIC was not accepted by any other APIC.
- *Receive Accept Error (RAE)*—Bit 3. The RAE bit when set to 1 indicates that a message received by the local APIC was not accepted by this or any other APIC
- *Sent Illegal Vector (SIV)*—Bit 5. The SIV bit when set to 1 indicates that the local APIC attempted to send a message with an illegal vector value.
- *Receive Illegal Vector (RIV)*—Bit 6. The RIV bit when set to 1 indicates that the local APIC has received a message with an illegal vector value.
- *Illegal Register Address (IRA)*—Bit 7. The IRA bit when set to 1 indicates that an access to an unimplemented register location within the local APIC register range (APIC Base Address + 4 Kbytes) was attempted.

16.4.7 Spurious Interrupts

A timing issue exists between software and hardware that, though rare, results in spurious interrupts. In the event that the task priority is set to or above the level of the interrupt to be serviced while the interrupt is being acknowledged, the local APIC delivers a spurious interrupt to the CPU core instead, with the vector number specified by the Vector field of the Spurious Interrupt Register. The ISR is unaffected by the spurious interrupt, so the interrupt handler completes without sending an EOI back to the issuing local APIC.



Bits	Mnemonic	Description	R/W
31:10	Reserved	Reserved, Must be Zero	
9	FCC	Focus CPU Core Checking	R/W
8	ASE	APIC Software Enable	R/W
7:0	VEC	Vector	R/W

Figure 16-17. Spurious Interrupt Register (APIC Offset F0h)

The fields within the Spurious Interrupt register are as follows:

- *Vector (VEC)*—Bits 7:0. The VEC field contains the vector that is sent to the CPU core in the event of a spurious interrupt.
- *APIC Software Enable (ASE)*—Bit 8. The ASE bit when set to 0 disables the local APIC temporarily. When the local APIC is disabled, SMI, NMI, INIT, Startup, and Remote Read may be accepted; pending interrupts in the ISR and IRR are held, but further fixed, lowest-priority, and ExtInt interrupts are not accepted. All LVT entry mask bits are set and cannot be cleared. Setting the ASE bit to 1, enables the local APIC.
- *Focus CPU Core Checking (FCC)*—Bit 9. The FCC bit when set to 1 disables focus CPU core checking when the lowest-priority message type is used. A CPU core is the focus of an interrupt if it is already servicing that interrupt (ISR=1) or if it has a pending request for that interrupt (IRR=1). Clearing the FCC bit to 0 disables focus CPU core checking.

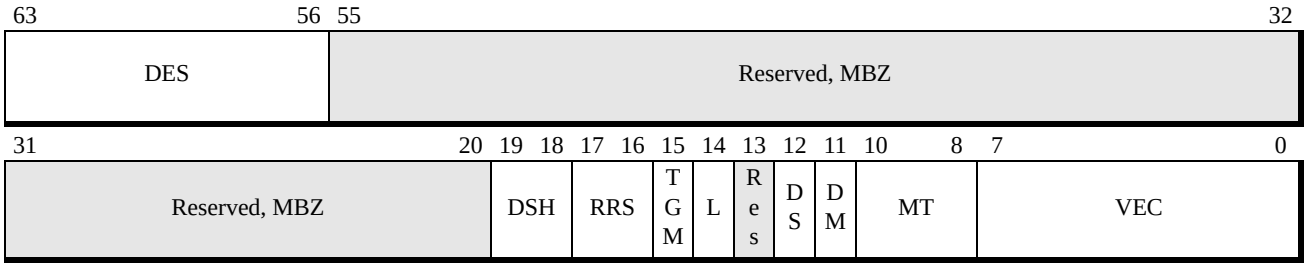
16.5 Interprocessor Interrupts (IPI)

A local APIC can send interrupts to other local APICs (or itself) using software-initiated Interprocessor Interrupts (IPIs) using the Interrupt Command Register (ICR). Writing into the low order doubleword of the ICR causes the IPI to be sent.

The ICR can issue the following types of interrupt messages:

- basic interrupt message to another local APIC, including forwarding an interrupt that was received but not serviced
- basic interrupt message to the same local APIC (self-interrupt)
- system management interrupt (SMI)
- remote read message to another local APIC to read one of its APIC registers.
- non-maskable interrupt (NMI) delivered to another local APIC
- initialization message (INIT) to a target local APIC to be reset to their INIT state and await a STARTUP IPI.
- startup message (SIPI) to the target local APICs, pointing to a start-up routine.

The format of the Interrupt Command Register is shown in Figure 16-18.



Bits	Mnemonic	Description	R/W
63:56	DES	Destination	R/W
55:20	—	Reserved, MBZ	
19:18	DSH	Destination Shorthand	R/W
17:16	RRS	Remote Read Status	RO
15	TGM	Trigger Mode	R/W
14	L	Level	R/W
13	—	Reserved, MBZ	
12	DS	Delivery Status	RO
11	DM	Destination Mode	R/W
10:8	MT	Message Type	R/W
7:0	VEC	Vector	R/W

Figure 16-18. Interrupt Command Register (APIC Offset 300h–310h)

The fields within the Interrupt Command register are as follows:

- *Vector (VEC)*—Bits 7:0. The function of this field varies with the Message Type field. The VEC field contains the vector that is sent for this interrupt source for fixed and lowest priority message types.
- *Message Type (MT)*—Bits 10:8. The MT field specifies the message type sent to the CPU core interrupt handler. The legal values are:
  - 000b = Fixed - The IPI delivers an interrupt to the target local APIC specified in Destination field.

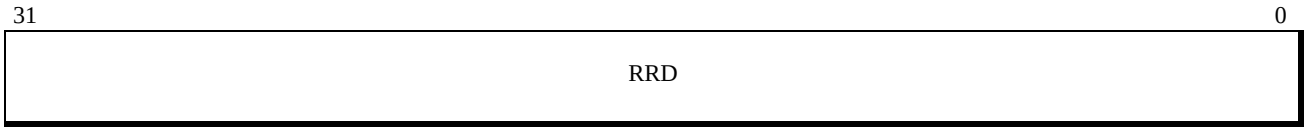
- 001b = Lowest Priority - The IPI delivers an interrupt to the local APIC executing at the lowest priority of all local APICs that match the destination logical ID specified in the Destination field. See Section 16.6.1, “Receiving System and IPI Interrupts,” on page 619.
- 010b = SMI - The IPI delivers an SMI interrupt to target local APIC(s). The trigger mode is edge-triggered and the Vector field must = 00h.
- 011b = Remote read - The IPI delivers a read request to read an APIC register in the target local APIC specified in Destination field. The trigger mode is edge triggered and the Vector field specifies the APIC offset of the APIC register to be read. The Remote Status field provides the current status of the remote read access after it has been issued. Data is returned from the target local APIC and captured in the Remote Read Register of the issuing local APIC. See Figure 16-19 on page 618.
- 100b = NMI - The IPI delivers a non-maskable interrupt to the target local APIC specified in the Destination field. The Vector field is ignored.
- 101b = INIT - The IPI delivers an INIT request to the target local APIC(s) specified in the Destination field, causing the CPU core to assume the INIT state. The trigger mode is edge-triggered, and the Vector field must =00h. In the INIT state, the target APIC is responsive only to the STARTUP IPI. All other interrupts (including SMI and NMI) are held pending until the STARTUP IPI has been accepted.
- 110b = STARTUP - The IPI delivers a start-up request (SIPI) to the target local APIC(s) specified in Destination field, causing the CPU core to start processing the platform firmware boot-strap routine whose address is specified by the Vector field.
- 111b = External interrupt - The IPI delivers an external interrupt to the target local APIC specified in Destination field. The interrupt can be delivered even if the APIC is disabled.

- *Destination Mode (DM)*—Bit 11. The DM bit when set to 1 specifies a logical destination which may be one or more local APICs with a common destination logical ID. When cleared to 0, the DM bit specifies a physical destination which indicates a single local APIC ID.
- *Delivery Status (DS)*—Bit 12. The DS bit indicates the interrupt delivery status. The DS bit is set to 1 when the local APIC has sent the IPI and is waiting for it to be accepted by another local APIC (the ICR is not idle). Clearing the DS bit indicates that the target local APIC is idle. Code may repeatedly write ICRL without polling the DS bit; all requested IPIs will be delivered.
- *Level (L)*—Bit 14. The L bit when set to 1 indicates assert. Clearing the L bit to 0 indicates deassert.
- *Trigger Mode (TGM)*—Bit 15. Specifies how IPIs to the local APIC are triggered. The TGM bit is set to 1 when the interrupt is level-sensitive. It is cleared to 0 when the interrupt is edge-triggered.
- *Remote Read Status (RRS)*—Bits 17:16. The RRS field indicates the current read status of a Remote Read from another local APIC. The encoding for this field is as follows:
  - 00b = Read was invalid
  - 01b = Delivery pending
  - 10b = Delivery done and access was valid. Data available in Remote Read Register.

- 11b = Reserved

- *Destination Shorthand (DSH)*—Bits 19:18. The DSH field indicates whether a shorthand notation is used, and provides a quick way to specify a destination for a message. It replaces the Destination field, when the destination field is not required (DSH > 00b), allowing software to use a single write to the low order ICR. The encoding are as follows:
  - 00b = Destination - The Destination field is required to specify the destination.
  - 01b = Self - The issuing APIC is the only destination.
  - 10b = All including self - The IPI is sent to all local APICs including itself (destination field=FFh).
  - 11b = All excluding self - The IPI is sent to all local APICs except itself (destination field=FFh).

    Note that if the lowest priority is used, the message could end up being reflected back to this local APIC. If DS=1xb, the destination mode is ignored and physical is automatically used.
- *Destination (DES)*—Bits 63:56. The DES field identifies the target local APIC(s) for the IPI and contains the destination encoding used when the Destination Shorthand field=00b. The field indicates the target local APIC when the destination mode=0 (physical), and the destination logical ID (as indicated by LDR and DFR) when the destination mode=1 (logical).



Bits	Mnemonic	Description	R/W
31:0	RRD	Remote Read Data	RO

Figure 16-19. Remote Read Register (APIC Offset C0h)

- *Remote Read Data (RRD)*—Bits 31:0. The RRD field contains the data resulting from a valid completion of a remote read interprocessor interrupt.

Not all combinations of ICR fields are valid. Only the combinations indicated in Table 16-4 are valid.

Table 16-4. Valid ICR Field Combinations

Message Type	Trigger Mode	Level	Destination Shorthand
Fixed	Edge	x	x
	Level	Assert	x
Lowest Priority, SMI, NMI, INIT	Edge	x	Destination or all excluding self.
	Level	Assert	Destination or all excluding self
Startup	x	x	Destination or all excluding self

Note: *x indicates a don't care.*

16.6 Local APIC Handling of Interrupts

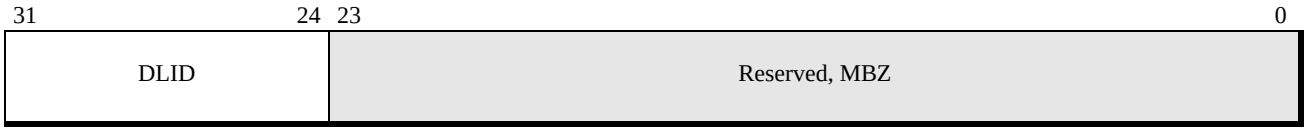
16.6.1 Receiving System and IPI Interrupts

Each local APIC verifies the destination ID, the destination mode and the message type of an APIC interrupt to determine if it is the target of the interrupt.

The destination mode is either physical or logical. In physical destination mode, the value of the interrupt message destination field is compared with the unique APIC ID value of each local APIC to select the target local APIC. If the destination field of the Interrupt Command Register is set to FFh, the interrupt is broadcasted and accepted by all local APICs. In physical destination mode, the lowest priority message type is not supported.

In logical destination mode, all local APICs use the Logical Destination Register and the Destination Format Register to determine if the interrupt is directed to them. The value of the interrupt message destination field is compared with the value in the Logical Destination Register (see Figure 16-20) of all local APICs.

The logical APIC ID must be unique. Since the comparison with the interrupt message destination field is on a bit-basis, there are only 8 unique logical IDs (01h, 02h, 04h, 08h, 10h, 20h, 40h, and 80h). For flat mode, the logical ID must be one of these values (for a total of eight local APICs supported). In cluster mode, the value of the logical ID is constrained to be *xy*h, where $0 \leq x \leq$ Eh and *y* = either 1,2,4, or 8, for a total of (15 × 4) possible unique logical IDs.

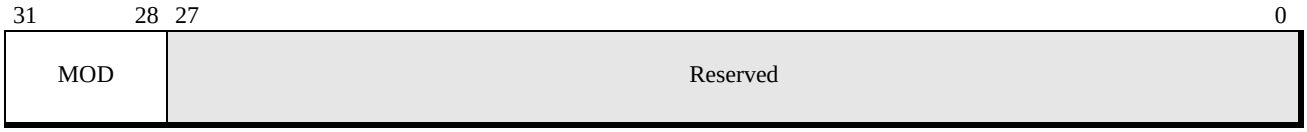


Bits	Mnemonic	Description	R/W
31:24	DLID	Destination Logical ID	R/W
23:0	—	Reserved, MBZ	

Figure 16-20. Logical Destination Register (APIC Offset D0h)

- *Destination Logical ID (DLID)*—Bits 31:24. The DLID field contains the logical APIC ID assigned to this specific CPU core. The logical APIC ID must be unique.

Two interrupt models are defined for the logical destination mode, the flat model and the cluster model, under the control of the Destination Format Register. See Figure 16-21.



Bits	Mnemonic	Description	R/W
31:28	MOD	Model	R/W
27:0	—	Reserved (all ones)	R

Figure 16-21. Destination Format Register (APIC Offset E0h)

- *Model (MOD)*—Bits 31:28. The MOD field controls which format to use when accepting interrupts in logical destination mode. The allowable values are 0h = cluster model and Fh = flat model.

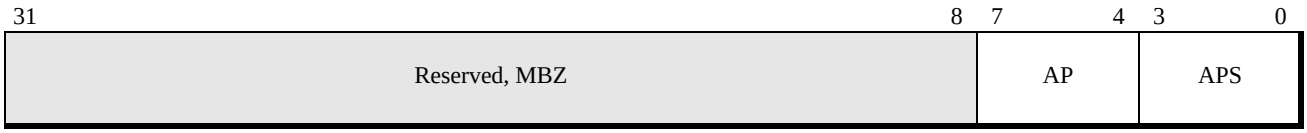
With the flat model, up to eight unique logical APIC ID values can be provided by software by setting a different bit in the LDR. When the logical ID of the destination is compared with the LDR, if any bit position is set in both fields, this local APIC is a valid destination. A broadcast to all local APICs occurs when the LDR is set to all ones.

In the cluster model, bits 31:28 of the logical ID of the destination are compared with bits 31:28 of the LDR. If there is a match, then bits 27:24 are tested for matching ones, similar to the flat model. If bits 31:28 match, and any of bits 27:24 are set in both fields, this local APIC is a valid destination. The cluster model allows for 15 unique clusters to be defined, with each cluster having four unique logical APIC values to be addressed. In cluster logical destination mode, lowest priority message type is not supported.

In both the flat model and the cluster model, if the destination field = FFh, the interrupt is accepted by all local APICs.

16.6.2 Lowest Priority Messages and Arbitration

In the case where the interrupt is valid for several local APICs in logical destination mode with a lowest priority message type, the interrupt is accepted by the local APIC with the lowest arbitration priority, as indicated by the *Arbitration Priority* field in the Arbitration Priority Register (APR). The value in the *Arbitration Priority* field indicates the current priority for a pending interrupt or task, or an interrupt being serviced by the CPU core. See Figure 16-22.



Bits	Mnemonic	Description	R/W
31:8	—	Reserved, MBZ	
7:4	AP	Arbitration Priority	RO
3:0	APS	Arbitration Priority Sub-class	RO

Figure 16-22. Arbitration Priority Register (APIC Offset 90h)

The fields within the Arbitration Priority register are as follows:

- *Arbitration Priority Sub-class (APS)*—Bits 3:0. The APS field indicates the current sub-priority to handle arbitrated interrupts to be serviced by the CPU core.
- *Arbitration Priority (AP)*—Bits 7:4. The AP field indicates the current priority to handle arbitrated interrupts to be serviced by the CPU core. The priority is used to arbitrate between CPU cores to determine which core accepts a lowest-priority interrupt request.

The value in the Arbitration Priority field is equal to the highest priority of the Task Priority field of the Task Priority Register (TPR), the highest bit set in the In-Service Register (ISR) vector, or the highest bit set in the Interrupt Request Register (IRR) vector. The value in the Arbitration Priority Sub-class field is equal to the Task Priority Sub-class if the APR is equal to the TPR, and zero otherwise.

If focus CPU core checking is enabled (Spurious Interrupt Register bit 9=0), the focus CPU core for an interrupt can always accept the interrupt. A CPU core is the focus of an interrupt if it is already servicing that interrupt (corresponding ISR bit is set) or if it already has a pending request for that interrupt (corresponding IRR bit is set). If there is no focus CPU core for an interrupt or if focus CPU core checking is disabled (Spurious Interrupt Register bit 9=1), all target local APICs identified as candidates for the interrupt arbitrate to determine which is executing with the lowest arbitration priority. If there is a tie for lowest priority, the local APIC with the highest APIC ID is selected.

16.6.3 Accepting System and IPI Interrupts

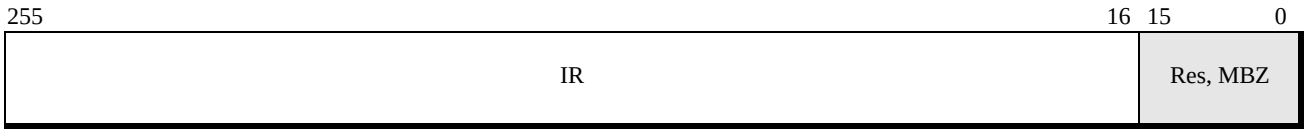
If the local APIC accepting the interrupt determines that the message type for the interrupt request indicates SMI, NMI, INIT, STARTUP or ExtINT, it sends the interrupt directly to the CPU core for handling. If the message type is fixed or lowest priority, the accepting local APIC places the interrupt into an open slot in either the IRR or ISR registers. If there is no free slot, the interrupt is rejected and sent back to the sender with a retry request.

Three 256-bit acceptance registers support interrupts accepted by the local APIC. Bits 255:16 correspond to interrupt vectors 255:16 with 255 being the highest priority; bits 15:0 are reserved.

- Interrupt Request Register (IRR), which contains interrupt requests that have been accepted but have not been sent to the CPU core for interrupt handling. When a system interrupt is accepted, the associated bit corresponding to the interrupt vector is set in the IRR. When the CPU core requests a

new interrupt, the local APIC selects the highest priority IRR interrupt and sends it to the CPU core. The local APIC then sets the corresponding bit in the ISR and resets the associated IRR bit. See Figure 16-23 on page 622.

- In-Service Register (ISR) contains the bit map of the interrupts that have been sent to the CPU core and are still being serviced. When the CPU core writes to the EOI register indicating completion of the interrupt processing, the associated ISR bit is reset and a new interrupt is selected from the IRR register. If a higher priority interrupt is accepted by the local APIC while the CPU core is servicing another interrupt, the higher priority interrupt is sent directly to the CPU core (before the current interrupt finishes processing) and the associated IRR bit is set. The CPU core interrupts the current interrupt handler to service the higher priority interrupt. When the interrupt handler for the higher priority interrupt completes, the associated IRR bit is reset and the interrupt handler returns to complete the previous interrupt handler routine. If a second interrupt with the same interrupt vector number is received by the local APIC while the ISR bit is set, the local APIC sets the IRR bit. No more than two interrupts can be pending for the same interrupt vector number. Subsequent interrupt requests to the same interrupt vector number will be rejected. See Figure 16-24 on page 623.
- Trigger Mode Register (TMR) indicates the trigger mode of the interrupt and determines whether an EOI message is sent to the I/O APIC for level-sensitive interrupts. When the interrupt is accepted by the local APIC and the IRR bit is set, the associated TMR bit is set for level-sensitive interrupts or reset for edge-triggered interrupts. At the end of the interrupt handler routine, when the EOI is received at the local APIC, an EOI message is sent to the I/O APIC if the associated TMR bit is set for a system interrupt. See Figure 16-25 on page 624.

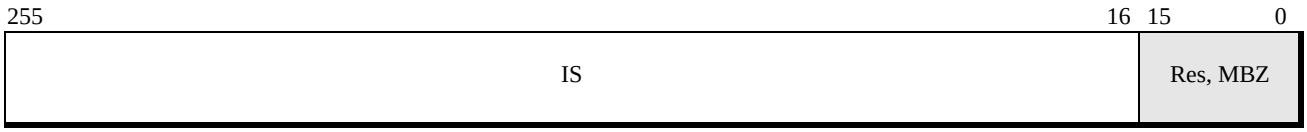


Bits	Mnemonic	Description	R/W
255:16	IR	Interrupt Request bits	RO
15:0	—	Reserved, MBZ	

Figure 16-23. Interrupt Request Register (APIC Offset 200h–270h)

- *Interrupt Request bits (IR)*—Bits 255:16. The corresponding request bit is set when an interrupt is accepted by the local APIC. The interrupt request registers provide a bit per interrupt to indicate that the corresponding interrupt has been accepted by the local APIC. Interrupts are mapped as follows:

Register	Interrupt Number
IRR (APIC offset 200h)	31–16
IRR (APIC offset 210h)	63–32
IRR (APIC offset 220h)	95–64
IRR (APIC offset 230h)	127–96
IRR (APIC offset 240h)	159–128
IRR (APIC offset 250h)	191–160
IRR (APIC offset 260h)	223–192
IRR (APIC offset 270h)	255–224

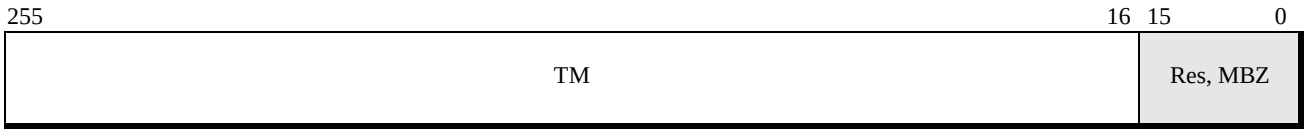


Bits	Mnemonic	Description	R/W
255:16	IS	In Service bits	RO
15:0	—	Reserved, MBZ	

Figure 16-24. In Service Register (APIC Offset 100h–170h)

- *In Service bits (IS)*—Bits 255:16. These bits are set when the corresponding interrupt is being serviced by the CPU core. The in-service registers provide a bit per interrupt to indicate that the corresponding interrupt is being serviced by the CPU core. Interrupts are mapped as follows:

Register	Interrupt Number
ISR (APIC offset 100h)	31–16
ISR (APIC offset 110h)	63–32
ISR (APIC offset 120h)	95–64
ISR (APIC offset 130h)	127–96
ISR (APIC offset 140h)	159–128
ISR (APIC offset 150h)	191–160
ISR (APIC offset 160h)	223–192
ISR (APIC offset 170h)	255–224



Bits	Mnemonic	Description	R/W
255:16	TM	Trigger Mode bits	RO
15:0	—	Reserved, MBZ	

Figure 16-25. Trigger Mode Register (APIC Offset 180h–1F0h)

- *Trigger Mode bits (TM)*—Bits 255:16. These bits provide a bit per interrupt to indicate the assertion mode of each interrupt. Interrupts are mapped as follows:

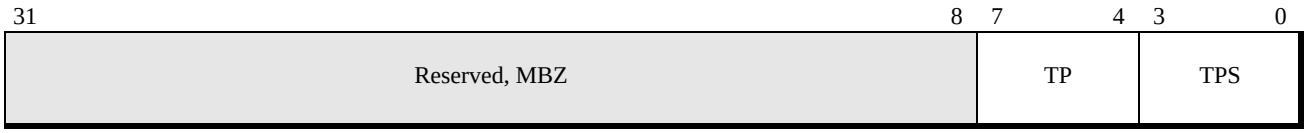
Register	Interrupt Number
TMR (APIC offset 180h)	31–16
TMR (APIC offset 190h)	63–32
TMR (APIC offset 1A0h)	95–64
TMR (APIC offset 1B0h)	127–96
TMR (APIC offset 1C0h)	159–128
TMR (APIC offset 1D0h)	191–160
TMR (APIC offset 1E0h)	223–192
TMR (APIC offset 1F0h)	255–224

16.6.4 Selecting and Handling Interrupts

Interrupts are selected by the local APIC for delivery to the CPU core interrupt handler on a priority determined by the interrupt vector number. Of the 15 priority levels, 15 is the highest and 1 is the lowest. The priority level for an interrupt is equal to the interrupt vector number divided by 16, rounded down to the nearest integer, with vectors 0Fh–00h reserved. Therefore, interrupt vectors 79h and 70h have the same priority level. The high-order hex digit indicates the priority level while the low-order hex digit indicates the priority within the same priority level.

Two registers are used to determine the priority threshold for selecting interrupts to be delivered to the CPU core, the Task Priority Register (TPR) and the Processor Priority Register (PPR). Software uses the TPR to set a priority threshold for interrupts to the CPU core, allowing the OS to block specific interrupts. See Figure 16-26 on page 625 for more details on the TPR.

The value in the *Task Priority* field is set by software to set a threshold priority at which the processor is to be interrupted. The value varies from 0 (all interrupts are allowed) to 15 (all interrupts with fixed delivery mode are inhibited). See Figure 16-26.



Bits	Mnemonic	Description	R/W
31:8	—	Reserved, Must be Zero	
7:4	TP	Task Priority	R/W
3:0	TPS	Task Priority Sub-class	R/W

Figure 16-26. Task Priority Register (APIC Offset 80h)

The fields within the Task Priority register are as follows:

- *Task Priority Sub-class (TPS)*—Bits 3:0. The TPS field indicates the current sub-priority to be used when arbitrating lowest-priority messages. This field is written with zero when TPR is written using the architectural CR8 register.
- *Task Priority (TP)*—Bits 7:4. The TP field indicates the current priority to be used when a core is deciding when to handle interrupts. A value of zero allows all interrupts; a value of Fh disables all interrupts. TP is also used to arbitrate between CPU cores to determine which core accepts a lowest-priority interrupt request. This field can also be written using the architectural CR8 register.

The PPR is set by the CPU core and represents the current priority level at which the CPU core is executing. The PPR determines whether a pending interrupt in the local APIC can be selected for interrupt handling in the CPU core. The value set by hardware is either the interrupt priority level of the highest priority ISR bit set or the value in the TPR, whichever is higher. The PPR is equal to the TPR when the CPU core is not servicing a higher priority interrupt. See Figure 16-27 on page 625.

Bits	Mnemonic	Description	R/W
31:8	—	Reserved, MBZ	
7:4	PP	Processor Priority	RO
3:0	PPS	Processor Priority Sub-class	RO

Figure 16-27. Processor Priority Register (APIC Offset A0h)

The fields within the Processor Priority register are as follows:

- *Processor Priority Sub-class (PPS)*—Bits 3:0. The PPS field is set to the Task Priority sub-class field of the Task Priority Register (TPR) if the PP field is equal to the Task Priority field of the TPR.
- *Processor Priority (PP)*—Bits 7:4. The PP field indicates the CPU core's current priority for servicing a task or interrupt, and is used to determine if any pending interrupts should be serviced.

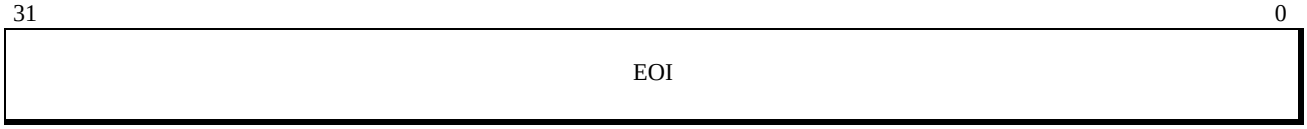
It is the higher value of either the interrupt priority level of the highest priority ISR bit set or the value in the TPR.

Pending interrupts must have a higher priority level than the value in the PPR to be selected by the local APIC for interrupt handling in the core; otherwise, they remain pending in the IRR until the PPR is lowered below the pending interrupt priority level. No pending interrupts are selected by the local APIC when the TPR=15.

The local APIC selects the highest priority pending interrupt (highest priority IRR) when the CPU core is ready, and sends the interrupt (with the IRR vector) to the CPU core. The local APIC resets the highest priority IRR bit and sets the associated ISR bit.

As part of the completion of the interrupt handling routine, software writes a value of zero to the End-of-Interrupt Register (EOI) in the local APIC, which causes the local APIC to reset the associated ISR bit. The EOI register is a write-only register.

If a higher priority interrupt is accepted by the local APIC while the CPU core is servicing another interrupt, the higher priority interrupt is sent directly to the CPU core (before the current interrupt finishes processing) and the associated ISR bit is set. The CPU core interrupts the current interrupt handler to service the higher priority interrupt. When the interrupt handler for the higher priority interrupt completes, the associated ISR bit is reset and the interrupt handler returns to complete the previous interrupt handler routine.



Bits	Mnemonic	Description	R/W
31:0	EOI	End of Interrupt	WO

Figure 16-28. End of Interrupt (APIC Offset B0h)

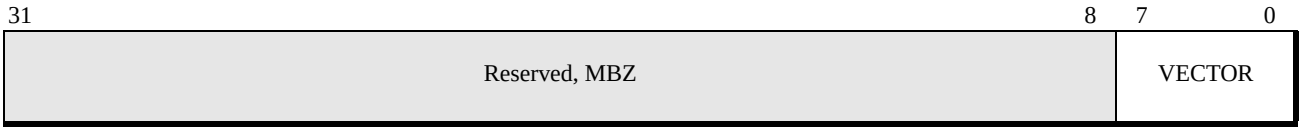
- *End of Interrupt (EOI)*—Bits 31:0. Write-only operation signals end of interrupt processing to source of interrupt.

16.7 SVM Support for Interrupts and the Local APIC

The SVM hypervisor uses the Extended APIC Feature Register, Extended APIC Control Register, Specific End of Interrupt Register (SEOI), and Interrupt Enable Register (IER) to control virtualized interrupts. When guests have direct access to devices, interrupts arriving at the local APIC can usually be dismissed only by the guest that owns the device causing the interrupt. To prevent one guest from blocking other guests' interrupts (by never processing their own), the VMM can mask pending interrupts in the local APIC, so they do not participate in the prioritization of other interrupts.

16.7.1 Specific End of Interrupt Register

Software issues a specific EOI (SEOI) by writing the vector number of the interrupt to the SEOI register in the local APIC. The SEOI register is located at offset 420h in the APIC space. The SEOI register format is shown in Figure 16-29.

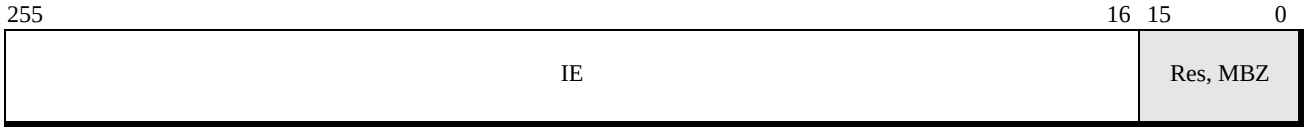


Bits	Mnemonic	Description	R/W
31:8	—	Reserved, MBZ	
7:0	VECTOR	Vector Number of Interrupt	R/W

Figure 16-29. Specific End of Interrupt (APIC Offset 420h)

16.7.2 Interrupt Enable Register

The IER is made available to software by means of eight 32-bit registers in the local APIC; bit *i* of the 256-bit IER is located at bit position (*i* mod 32) in the local APIC register IER[*i* / 32]. The eight IER registers are located at offsets 480h, 490h, ...,4F0h in APIC space. The IER format is shown in Figure 16-30.



Bits	Mnemonic	Description	R/W
255:16	IE	Interrupt Enable	R/W
15:0	—	Reserved, MBZ	

Figure 16-30. Interrupt Enable Register (APIC Offset 480h–4F0h)

- *Interrupt Enable (IE)*—Bits 255:16. Interrupts are mapped as follows:

Register	Interrupt Number
IER (APIC offset 480h)	31–16
IER (APIC offset 490h)	63–32
IER (APIC offset 4A0h)	95–64
IER (APIC offset 4B0h)	127–96
IER (APIC offset 4C0h)	159–128
IER (APIC offset 4D0h)	191–160

Register	Interrupt Number
IER (APIC offset 4E0h)	223–192
IER (APIC offset 4F0h)	255–224

The IER and SEOI registers are located in the APIC Extended Space area. The presence of the APIC Extended Space area is indicated by bit 31 of the APIC Version Register (at offset 30h in APIC space).

The presence of the IER and SEOI functionality is identified by bits 0 and 1, respectively, of the APIC Extended Feature Register (located at offset 400h in APIC space). IER and SEOI are enabled by setting bits 0 and 1, respectively, of the APIC Extended Control Register (located at offset 410h).

Only vectors that are enabled in IER participate in APIC's computation of the highest-priority pending interrupt. The reset value of IER is all ones.

16.8 x2APIC Mode

x2APIC mode is an extension to the local APIC architecture designed to support larger CPU topologies and to enhance delivery of interrupts. When enabled, x2APIC mode adds the following features:

- All APIC and x2APIC registers are accessed via an MSR-based interface.
- The Local APIC ID and Logical APIC ID registers are expanded to 32 bits.
- Logical and physical destinations are extended to 32 bits.
- A new self-IPI register simplifies sending IPIs to self.

In general, x2APIC mode maintains backwards compatibility with the key elements of the local APIC functionality previously described in Section 16. However, the following local APIC functionality is changed in x2APIC mode:

- The Destination Format Register (DFR) is no longer needed and is not supported.
- The two 32-bit Interrupt Command Registers (ICRs) are merged into a single 64-bit ICR.
- Local APIC Base (MSR 01Bh) bit 10, previously reserved, is used to enable x2APIC mode.

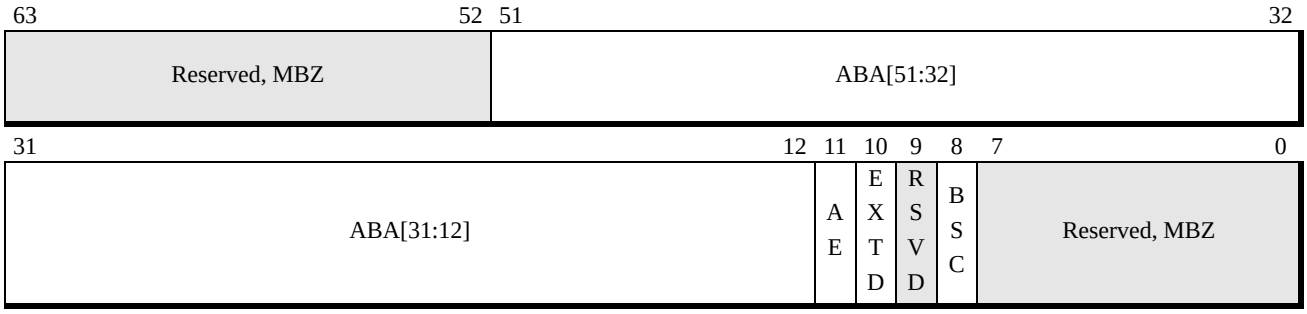
16.8.1 x2APIC Terminology

Although x2APIC is an operational mode of the local APIC, the following sections use the term ‘x2APIC’ as a qualifier to more conveniently identify a given aspect of the local APIC when that mode is enabled. For example, the term ‘x2APIC programming interface’ may be used instead of ‘the programming interface when the local APIC is in x2APIC mode’.

16.9 Detecting and Enabling x2APIC Mode

System software can detect the presence of the x2APIC feature by executing the CPUID instruction. Support for x2APIC mode is indicated by CPUID Fn000_0001_ECX[x2APIC] (bit 21) = 1.

If the feature is present, the local APIC is placed into x2APIC mode by setting bit 10 in the Local APIC Base register (MSR 01Bh). Before entering x2APIC mode, the local APIC must first be enabled (AE=1, EXTD=0). System software can then place the local APIC into x2APIC mode by executing a WRMSR with both AE=1 and EXTD=1. The format of the Local APIC Base register for x2APIC-capable processors is shown in Figure 16-31.



Bits	Mnemonic	Description	R/W
63:52	—	Reserved, MBZ	
51:12	ABA	APIC Base Address	R/W
11	AE	APIC Enable (xAPIC mode)	R/W
10	EXTD	X2APIC Mode Enable	R/W
9	—	Reserved, MBZ	
8	BSC	Boot Strap CPU Core	R
[7:0]	—	Reserved, MBZ	

Figure 16-31. APIC Base Address Register (MSR 01Bh) support for x2APIC

Not all combinations of the local APIC mode bits are valid. See Table 16-5. Attempting to set the combination of AE=0 and EXTD=1 is invalid and causes the WRMSR instruction to generate a #GP(0) exception.

Table 16-5. Local APIC Operating Modes

AE (bit 11)	EXTD (bit 10)	Local APIC Mode
0	0	local APIC disabled
0	1	Invalid
1	0	local APIC enabled in xAPIC mode
1	1	local APIC enabled in x2APIC mode

Similarly, not all transitions between local APIC states are valid. The valid state transitions are illustrated in Figure 16-2. Once the local APIC has been placed into x2APIC mode, the only valid transition (other than reset) is to “APIC Disabled” mode by simultaneously clearing AE and EXTD to zero. Executing a WRMSR instruction to attempt a transition other than those specified in Figure 16-32 results in a #GP(0) exception.

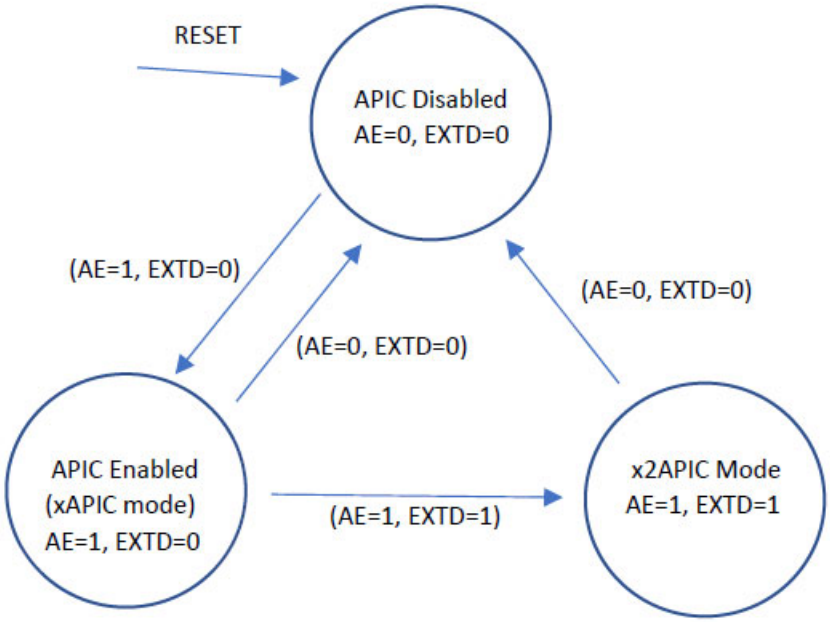


Figure 16-32. Valid APIC State Transitions

16.9.1 Enabling x2APIC Mode

Starting from APIC Disabled mode, enabling x2APIC mode is a two-step process. First, a transition is made to APIC_Enabled mode by setting APIC_Base bit 11 (AE) to 1 with bit 10 (EXTD) left as zero. Next, the transition to x2APIC mode is made by setting both AE and EXTD to 1 simultaneously.

Most APIC registers previously written by software while in APIC_Enabled mode are not affected by the transition to x2APIC mode. The exceptions are:

- The Logical Destination Register (LDR) is not preserved.
- The upper half of the Interrupt Command Register (ICR_High) is not preserved.
- A value previously written by software to the 8-bit APIC_ID register (MMIO offset 30h) is converted by hardware into the appropriate format and reflected into the 32-bit x2APIC_ID register (MSR 802h).

Leaving x2APIC mode. Once in x2APIC mode, the only valid mode transition (other than a reset) is to APIC_Disabled mode by using WRMSR to clear APIC_Base bit 11 (AE) bit 10 (EXTD) to 0 at the same time.

16.10 x2APIC Initialization

A RESET clears the APIC Base Address Register AE bit and EXTD bit, disabling the local APIC. All local APIC registers are initialized to their reset values as described in section 16.3.2 “APIC Registers”.

An INIT does not modify the APIC Base Address Register AE and EXTD bits, thus the local APIC mode is not changed. All other APIC registers are initialized to their values as described in “Reset in x2APIC mode” above.

16.11 Accessing x2APIC Register

The x2APIC system programming interface consists of the Model Specific Registers listed in Table 16-6 below. All APIC registers except the APIC Base Address Register are mapped into the architecturally dedicated MSR range 800h to 8FFh, and accessed using WRMSR and RDMSR instructions.

The RDMSR/WRMSR instructions read and write the MSR specified in the ECX register using the EDX:EAX register pair as the destination and source operand. Bits 31:0 of the APIC register are mapped into EAX[31:0]. For 64-bit x2APIC registers, the high-order bits (bits 63:32) are mapped to EDX[31:0]. A #GP(0) exception is generated if an unimplemented APIC register is specified in ECX. When not in x2APIC mode, attempts to access the APIC register set using the MSR interface results in the WRMSR or RDMSR instruction generating a #GP(0) exception. See APM volume 3 for more information on the WRMSR and RDMSR instructions.

In x2APIC mode, the legacy MMIO access to the APIC register set is disabled. Attempts to use the legacy MMIO access mechanism may result in an unintended memory access or a memory-related exception such as #GP or #PF.

16.11.1 x2APIC Register Address Space

The x2APIC registers are mapped into the MSR address range 800h through 8FFh. This address range is reserved for accessing the APIC register set in x2APIC mode.

The legacy APIC registers are mapped to this MSR address space using on the following formula:

x2APIC MSR address = 800h + ((APIC MMIO offset) >> 4)

The following registers are exceptions to the above formula:

- The two 32-bit Interrupt Command Registers in APIC mode (MMIO offsets 300h and 310h) are merged into a single 64-bit x2APIC register at MSR address 830h.
- The x2APIC Self_IPI register is added at MSR address 83Fh
- The Destination Format Register (DFR) at MMIO offset E0h and the Remote Read Register (RRR) at MMIO offset C0h are not supported in x2APIC mode. Accordingly, MSR addresses 80Eh and 80Ch are not used and are reserved.

The system programming interface of the local APIC in x2APIC mode is made up of the MSRs listed in Table 16-6 below.

Table 16-6. x2APIC Register

MSR Address (x2APIC mode)	MMIO Offset (xAPIC mode)	Register Name	Read/ Write	Notes
802h	20h	x2APIC ID Register [31:0]	RO	Expanded to 32-bits in x2APIC mode
803h	30h	APIC Version Register	RO	
808h	80h	Task Priority Register (TPR)	RW	
809h	90h	Arbitration Priority Register (APR)	RO	
80Ah	A0h	Processor Priority Register (PPR)	RO	
80Bh	B0h	End of Interrupt Register (EOI)	WO	#GP(0) if non-zero value is written
-	C0h	Remote Read Register	-	Eliminated in x2APIC mode
80Dh	D0h	Logical Destination Register (LDR)	RO	Expanded to 32-bits in x2APIC mode
-	E0h	Destination Format Register	-	Eliminated in x2APIC mode
80Fh	F0h	Spurious Interrupt Vector Register	RW	
800-817h	100-170h	In-Service Register (ISR)	RO	
818-81Fh	180-1F0h	Trigger Mode Register (TMR)	RO	
820-827h	200-270h	Interrupt Request Register (IRR)	RO	
828h	280h	Error Status Register (ESR)	RW	#GP(0) if non-zero value is written
830h	300h	Interrupt Command Register (bits 63:0)	RW	
832h	320h	Timer Local Vector Table Entry	RW	
833h	330h	Thermal Local Vector Table Entry	RW	
834h	340h	Perf Counter Local Vector Table Entry	RW	
835h	350h	Local Interrupt 0 Vector Table Entry	RW	
836h	360h	Local Interrupt 1 Vector Table Entry	RW	
837h	370h	Error Vector Table Entry	RW	
838h	380h	Timer Initial Count Register	RW	
839h	390h	Timer Current Count Register	RO	
83Eh	3E0h	Timer Divide Configuration Register	RW	
83Fh	—	Self IPI Register	WO	See Figure 16-6
840h	400h	Extended APIC Feature Register	RO	
841h	410h	Extended APIC Control Register	RW	
842h	420h	Specific End of Interrupt Register (SEOI)	RW	
848-84Fh	480-4F0h	Interrupt Enable Registers (IER)	RW	
850-853h	500-530h	Extended Interrupt [3:0] Local Vector Table Registers	RW	

MSR addresses in the range 800h through 8FFh that are not listed in Table 16-2 are unimplemented and reserved. A #GP(0) exception is generated if a WRMSR or an RDMSR instruction attempts to access an unimplemented MSR in the x2APIC address range.

16.11.2 WRMSR / RDMSR serialization for x2APIC Register

The WRMSR instruction is used to write the APIC register set in x2APIC mode. Normally WRMSR is a serializing instruction, however when accessing x2APIC registers, the serializing aspect of WRMSR is relaxed to allow for more efficient access to those registers. Consequently, a WRMSR write to an x2APIC register may complete before older store operations are complete and have become globally visible. When strong ordering of an x2APIC write access is required with respect to preceding memory operations, software can insert a serializing instruction (such as MFENCE) before the WRMSR instruction.

The RDMSR instruction is not a serializing instruction and remains non-serializing when reading x2APIC MSRs. However, WRMSR and RDMSR instructions targeting the x2APIC MSRs are always executed in program order with respect to each other.

16.11.3 Reserved Bit Checking in x2APIC Mode

When writing x2APIC MSRs, the WRMSR instruction checks for reserved bits. Attempting to write a ‘1’ to a reserved bit causes a #GP(0) exception. For x2APIC MSRs, WRMSR reserved bit checks are summarized as follows:

Legacy APIC registers. Reserved bit checks for existing APIC registers are the same as described for each register in non-x2APIC mode. For details, see the APIC register descriptions in sections 16.3 through section 16.6 above. Except for the Interrupt Command Register, attempting to write a ‘1’ into bits 63:32 of the legacy APIC registers causes a #GP(0) exception.

Interrupt Command Register (ICR). See the description of the 64-bit ICR register in section 16.13.

Error Status Register (ESR). A WRMSR of a non-zero value causes a #GP(0) exception.

SELF IPI register. See the description of the 32-bit SELF IPI register in section 16.15.

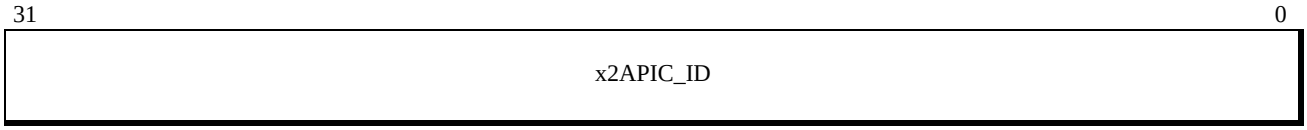
The RDMSR instruction returns a zero for any reserved bit.

16.12 x2APIC_ID

Unique local APIC IDs are assigned to each logical processor in the system. In x2APIC mode, the APIC ID is expanded to 32 bits and is referred to as the ‘x2APIC_ID’. It is assigned by hardware at reset time based on the processor topology of the system. The x2APIC_ID is a concatenation of several fields such as socket ID, core ID and thread ID.

Because the number of sockets, cores and threads may differ for each SOC, the format of x2APIC ID is model-dependent. Some fields may not be present, depending on the processor model and the

processor topology. The presence, size and position of each field is discoverable using the CPUID instruction (see “Cache and Processor Topology” on page 207).



Bits	Mnemonic	Description	R/W
31:0	x2APIC_ID	x2APICID	R

Figure 16-33. x2APIC_ID Register (MSR 802h)

System software can read x2APIC_ID using either of the following mechanisms:

RDMSR. An RDMSR of MSR 0802h returns the x2APIC_ID in EAX[31:0]. The x2APIC_ID is a read-only register. Attempting to write MSR 802h or attempting to read this MSR when not in x2APIC mode causes a #GP(0) exception. See 16.11 “Accessing x2APIC Registers”.

CPUID. The x2APIC ID is reported by the following CPUID functions Fn0000_000B (Extended Topology Enumeration) and CPUID Fn8000_001E (Extended APIC ID) as follows:

- Fn0000_000B_EDX[31:0]_x0 reports the full 32-bit ID, independent of APIC mode (i.e. even with APIC disabled)
- Fn8000_001E_EAX[31:0] conditionally reports APIC ID. There are 3 cases:
  - 32-bit x2APIC_ID, in x2APIC mode.
  - 8-bit APIC ID (upper 24 bits are 0), in xAPIC mode.
  - 0, if the APIC is disabled.

The above CPUID functions also report the presence, width and location of the sub-fields comprising the x2APIC ID. See APM Volume 3 Appendix E, “Obtaining Processor Information Via the CPUID Instruction” for detailed information.

16.13 x2APIC Interrupt Command Register (ICR) Operations

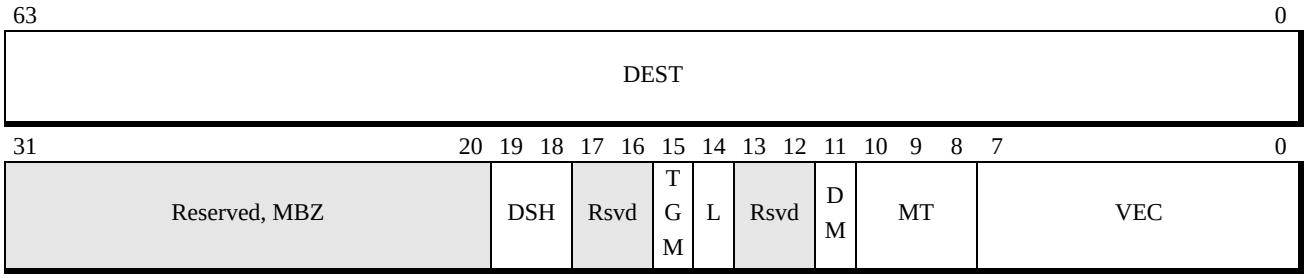
In legacy APIC mode, two 32-bit registers (ICR Low and ICR High) are used by system software to send Inter-Processor Interrupts (IPIs) to other local APICs. The x2APIC architecture combines these two registers into a single 64-bit Interrupt Command Register located at MSR address 830h. Thus in x2APIC mode sending an IPI requires only a single WRMSR to the ICR as opposed to two MMIO accesses in xAPIC mode.

The upper half of the x2APIC ICR (bits 63:32) contains the Destination ID (DEST) field, which is expanded to 32 bits in x2APIC mode. A DEST value of FFFF_FFFFh is used to broadcast IPIs to all local APICs.

The lower half of the x2APIC ICR (bits 31:0) is identical to the APIC Interrupt Command Register Low[31:0] (see Fig 16-18 on page 582), with the following exceptions:

- The Remote Read Status field (bits 17:16) is eliminated and must be zero.
- Message Type field (bits 10:8). Encodings 1, 3 and 7 are eliminated and the encodings are reserved.
- The Delivery Status field (bit 12) is eliminated and must be zero.

The format of the x2APIC ICR register is shown in Figure 16-34.

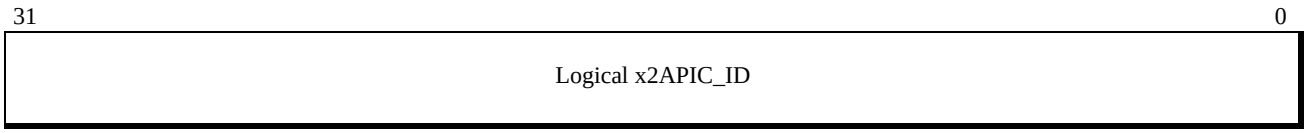


Bits	Mnemonic	Description	R/W
63:32	DEST	Destination	RW
55:20	—	Reserved, MBZ	
19:18	DSH	Destination Shorthand	RW
17:16	—	Reserved, MBZ	
15	TMG	Trigger Mode	RW
14	L	Level	RW
13:12	—	Reserved, MBZ	
11	DM	Destination Mode	RW
10:8	MT	Message Type	RW
7:0	VEC	Vector	RW

Figure 16-34. Interrupt Command Register (MSR 830h)

16.14 Logical Destination Register

When an IPI is sent using logical destination mode, all local APICs in the system use the Logical Destination Register to determine if the interrupt message is directed to them (see 16.6.1 Receiving System and IPI Interrupts). In x2APIC mode, the Logical Destination Register (LDR) is expanded to 32 bits and contains the ‘logical x2APIC_ID’. System hardware initializes LDR with the 32-bit logical x2APIC_ID whenever x2APIC mode is enabled. The LDR is a read-only register located at MSR address 080Dh. The format of the Logical Destination Registers is shown inFigure 16-35.



Bits	Mnemonic	Description	R/W
31:0	x2Logical_ID	Logical x2APICID Identifier	R

Figure 16-35. Logical Destination (MSR 80Dh)

The logical x2APIC_ID consists of two 16-bit sub-fields: cluster_ID and logical_ID.

- LDR[31:16] cluster_ id. Identifies the cluster of which this processor is a member.
- LDR[15:0] logical_id. A bit vector uniquely identifying this processor within the cluster.

In logical destination mode, a given logical processor is addressed by its unique cluster ID and logical_ID combination. The use of a bit vector for logical_ID allows an interrupt message to be routed to multiple processors within the addressed cluster.

The partitioning of logical x2APIC_ID provides for a possible 65,535 (216-1) clusters, with each cluster having up to 16 logical processors. The legacy “flat logical” addressing is not supported in x2APIC mode.

Upon receiving an interrupt message in logical destination format, each x2APIC compares bits 31:16 of the message destination with LDR[31:16] (cluster_id). If there is a match, then bits 15:0 of the destination and LDR[15:0] are tested for matching ones. If bits[31:16] (cluster_id) match and any bit in 15:0 (logical_id) match, this x2APIC is a valid destination.

A DEST value of FFFF_FFFFh in the ICR is used to broadcast IPIs to all local APICs.

The two sub-fields comprising logical x2APIC_ID are derived from the value of local x2APIC_ID. The 16-bit logical_ID sub-field is initialized by setting a single bit, ‘n’, where n= the 4 least significant bits of local x2APIC_ID. The 16-bit cluster_id is derived from the remaining bits of the x2APIC_ID. Specifically, logical_id[15:0] = 1 << x2APIC_ID[3:0] and cluster_id[15:0] = x2APIC_ID[19:4].

16.15 Self_IPI Register

The Self_IPI register (MSR 83Fh) provides a performance-optimized interface for system software to send interrupt messages to the local APIC. This register is write-only and attempts to read it cause a #GP(0) exception. The format of the Self_IPI registers is shown in Figure 16-36.

Bits	Mnemonic	Description	R/W
31:8	—	Reserved, MBZ	
7:0	VEC	Interrupt Vector	WO

Figure 16-36. Self_IPI Register (MSR 83Fh)

The Self_IPI register contains a single field: an interrupt vector. Writing to this register causes a to-self IPI to be generated, equivalent to a to-self IPI generated by writing the Interrupt Command Register (ICR, MSR 830h) with the following settings:

- Destination shorthand = self
- Trigger Mode = edge-triggered
- Message Type = fixed
- Vector = interrupt vector as specified in the Self_IPI register

The x2APIC's response to a self-IPI sent via the Self_IPI register is architecturally identical to one sent via the ICR. In particular, the operation of the Interrupt Response Register (IRR), In-Service Register (ISR) and Trigger Mode Register (TMR) is the same. See "Accepting System and IPI Interrupts" on page 621 for IRR, ISR and TMR details.

The Interrupt Request Register (IRR) contains interrupt requests that have been accepted by the processor core. Completion of the WRMSR to the Self_IPI register ensures that the resulting IPI has been entered into the IRR, and that the associated TMR bit is cleared (as expected for edge-triggered interrupts).

17 Hardware Performance Monitoring and Control

The AMD64 architecture provides several mechanisms by which software can monitor and control processor performance to optimize power use. The following lists the facilities that are described in the sections that follow:

- The P-state control interface allows dynamic control of performance states. See Section 17.1 which follows immediately below.
- Core performance boost (CPB) dynamically increases core clock rate beyond that defined for the P0 power state to achieve higher performance while maintaining power consumption below a preset level. See Section 17.2 on page 641.
- The effective frequency interface provides a measure of the actual core clock rate over a specified period of time. See Section 17.3 on page 642.
- The processor power reporting interface allows system software to measure average processor core power over a given time period. See Section 17.5 on page 644.

17.1 P-State Control

P-states are operational performance states (states in which the processor is executing instructions, that is, running software) characterized by a unique frequency of operation for a CPU core. The P-state control interface supports dynamic P-state changes in up to 16 P-states called P-states 0 through 15 or P0 though P15. P0 is the highest power, highest performance P-state; each ascending P-state number represents a lower-power, lower-performance state.

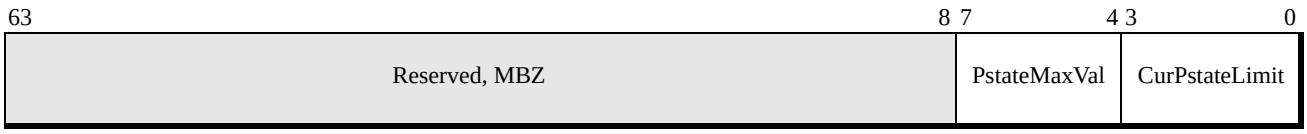
Core P-states are controlled by software. Each CPU core contains one set of P-state control registers. Software controls the P-states of each CPU core independently; however, hardware may include interdependencies that affect the P-state achieved by each core.

Hardware provides the highest P-state value in the PstateMaxVal field of the P-State Current Limit Register. P-states may be limited to a lower performance value under certain conditions. The current P-state limit is dynamic and is specified in the CurPstateLimit field of the P-State Current Limit Register.

Software requests a core P-state change by writing a 4-bit index corresponding to the desired core P-state number to the P-State Control Register of the appropriate core. For example, to request the P3 state for core 0, software writes 3h to the core 0's PstateCmd field in MSR C001_0062h. If the P-state value is greater than the value in PstateMaxVal, the value written is clipped to that value.

As the current P-state limit changes, the P-state for the CPU core is either set to the software-requested P-state value or the new current P-state limit, whichever is the higher P-state value.

The current P-state value can be read using the P-State Status Register. The P-State Current Limit Register and the P-State Status Register are read-only registers. Writes to these registers cause a #GP exception. Support for hardware P-state control is indicated by CPUID Fn8000_0007_EDX[HwPstate] = 1. Figure 17-1 below shows the format of the P-State Current Limit register.

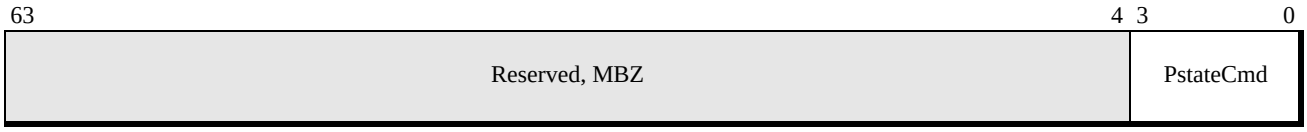


Bits	Mnemonic	Description	R/W
63:8	—	Reserved, MBZ	
7:4	PstateMaxVal	P-state maximum value	R
3:0	CurPstateLimit	Current P-state limit	R

Figure 17-1. P-State Current Limit Register (MSR C001_0061h)

The fields within the P-State Current Limit register are:

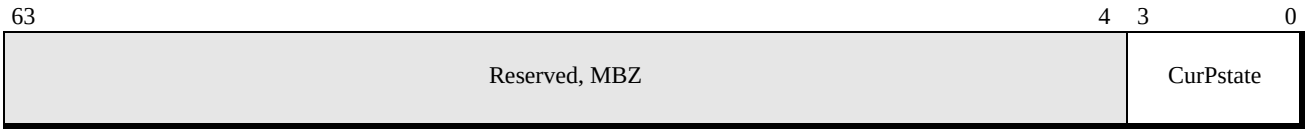
- *Current P-State Limit (CurPstateLimit)*—Bits 3:0. Provides the current P-state limit, which is the lowest P-state value (highest-performance state) that is currently supported by the hardware. This is a dynamic value controlled by hardware. Reset value is implementation specific.
- *P-State Maximum Value (PstateMaxVal)*—Bits 7:4. Specifies the highest P-state value (lowest performance state) supported by the hardware. Attempts to change the current P-state number to a higher value by writes to the P-State Control Register are clipped to the value of this field. Reset value is implementation specific.



Bits	Mnemonic	Description	R/W
63:4	—	Reserved, MBZ	
3:0	PstateCmd	P-state change command	R/W

Figure 17-2. P-State Control Register (MSR C001_0062h)

P-State Change Command (PstateCmd)—Bits 3:0. Writes to this field cause the CPU core to change to the indicated P-state number, which may be clipped by the PstateMaxVal field of the P-State Current Limit Register. Reset value is implementation specific.



Bits	Mnemonic	Description	R/W
63:4	—	Reserved, MBZ	
3:0	CurPstate	Current P-state	R

Figure 17-3. P-State Status Register (MSR C001_0063h)

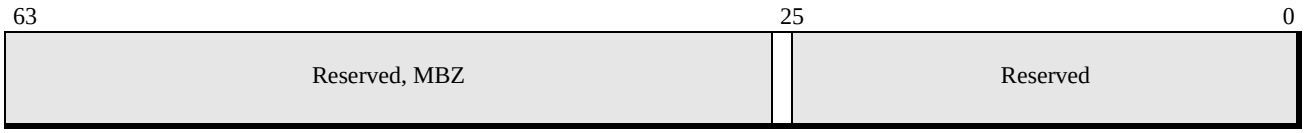
Current P-State (CurPstate)—Bits 3:0. This field provides the current P-state of the CPU core regardless of the source of the P-state change, including writes to the P-State Control Register: 0 = P-state 0, 1 = P-state 1, etc. The value of this field is updated when the frequency transitions to a new value associated with the P-state. Reset value is implementation specific.

17.2 Core Performance Boost

Core performance boost (CPB) dynamically monitors processor activity to create an estimate of power consumption. If the estimated processor consumption is below an internally defined power limit and software has requested P0 on a given core, hardware may transition the core to a frequency and voltage beyond those defined for P0. If the estimated power consumption exceeds the defined power limit, some or all cores are limited to the frequency and voltage defined by P0. CPB ensures that average power consumption over a thermally significant time period remains at or below the defined power limit.

CPB can be disabled using the CPBDis field of the Hardware Configuration Register (HWCR MSR) on the appropriate core. When CPB is disabled, hardware limits the frequency and voltage of the core to those defined by P0.

Support for core performance boost is indicated by CPUID Fn8000_0007_EDX[CPB] = 1. See Section 3.3, “Processor Feature Identification,” on page 71 for more information on using the CPUID instruction.



Bits	Mnemonic	Description	R/W
63:26	—	Reserved	
25	CPBDis	Core Performance Boost Disable	R/W
24:0	—	Reserved	

Figure 17-4. Core Performance Boost (MSR C001_0015h)

Core Performance Boost Disable (CpbDis)—Bit 25. Specifies whether core performance boost is enabled or disabled. 0 = Enabled. 1 = Disabled.

17.3 Determining Processor Effective Frequency

The Max Performance Frequency Clock Count (MPERF) and the Actual Performance Frequency Clock Count (APERF) registers constitute the effective frequency interface. This interface provides a means for software to calculate an average, or effective, frequency of a core over a known window of time. This provides software a measure of actual performance rather than forcing software to assume that the current frequency of the core is the frequency of the last P-state requested.

To calculate an effective clock frequency of a given processor core, on that processor do the following:

1. Read both MPERF and APERF and save their initial values.
   - MPERF_INIT = MPERF and APERF_INIT = APERF
2. Wait an appropriate amount of time.
3. Read both MPERF and APERF again.
4. Effective frequency = {(APERF − APERF_INIT) / (MPERF − MPERF_INIT)} * P0 frequency.

The amount of time that elapses between steps 1 and 3 is determined by software. This allows software to define the time window over which the processor frequency is averaged. Software should disable interrupts or any other events that may occur between the read of MPERF and the read of APERF in step 1 and again when the two MSRs are read in step 3. Step 4 provides the equation for the calculation of the effective frequency value. Software determines the P0 frequency using ACPI defined data structures.

The effective frequency interface only counts clock cycles while the core is in the ACPI defined C0 state.

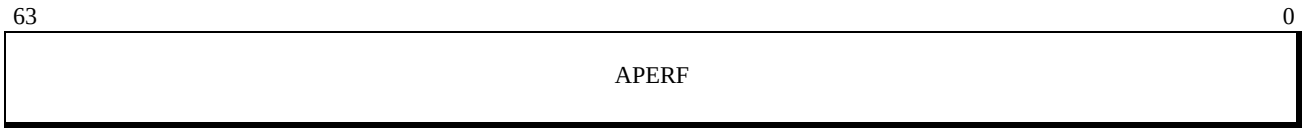
Only the ratio between MPERF and APERF is architecturally defined. Software should not assume any specific definition of the MPERF or APERF registers. If an overflow of either the MPERF or

APERF register occurs between the read of MPERF in step 1 and the read of APERF in step 3, the effective frequency calculated in step 4 is invalid.

Hardware support for the effective frequency interface is indicated by CPUID Fn0000_0006_ECX[EffFreq]. See Section 3.3, “Processor Feature Identification,” on page 71 for more information on using the CPUID instruction.

17.3.1 Actual Performance Frequency Clock Count (APERF)

Specifies the numerator of the effective frequency ratio.



Bits	Mnemonic	Description	Access Type
63:0	APERF	Actual Performance Frequency Clock Count	R/W

Figure 17-5. Actual Performance Frequency Count (MSR0000_00E8h)

17.3.2 Maximum Performance Frequency Clock Count (MPERF)

Specifies the denominator of the effective frequency ratio. The value read is scaled by the TSCRatio value (MSR C000_0104h) for guest reads, but the underlying counters are not affected. Reads in host mode or writes to MPERF are not affected.

Bits	Mnemonic	Description	Access Type
63:0	MPERF	Max Performance Frequency Clock Count	R/W

Figure 17-6. Max Performance Frequency Count (MSR0000_00E7h)

17.3.3 APERF Read-only (AperfReadOnly)

Specifies the numerator of the effective frequency ratio.

Bits	Mnemonic	Description	Access Type
63:0	APERF_RD_ONLY	APREF Read Only	R/O

Figure 17-7. APREF Read Only (MSR C000_00E8h)

17.3.4 MPERF Read-only (MperfReadOnly)

Read-only version of MPERF. The value read is scaled by the TSCRatio value (MSR C000_0104h) for guest reads.

Bits	Mnemonic	Description	Access Type
63:0	MPERF_RD_ONLY	MPERF Read Only	RO

Figure 17-8. MPERF Read Only (MSR C000_00E7h)

17.4 Processor Feedback Interface

The Processor Feedback Interface is deprecated. Some processor products may support this feature. To determine support on a given processor, software can test the feature bit CPUID Fn8000_0007_EDX[ProcFeedbackInterface]. For more information, consult the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.

17.5 Processor Core Power Reporting

The processor power reporting interface allows system software to estimate the average power consumed by a processor core over a software-determined time period. Computing the average power involves reading a "core power accumulator" register at the beginning and end of the measurement interval, taking the difference and then dividing by the length of the time interval.

Support for the processor power reporting interface is indicated by CPUID Fn8000_0007_EDX[ProcPowerReporting] = 1.

17.5.1 Processor Facilities

Estimating core average power involves the use of several processor facilities. Processors that support the processor power reporting interface define the following three facilities:

- CpuSwPwrAcc MSR
- MaxCpuSwPwrAcc MSR
- CpuPwrSampleTimeRatio (CPUID Fn8000_0007_ECX)

A fourth facility, available on all processors, is the time-stamp counter (TSC). The TSC is a free-running counter that increments on every processor clock cycle. The current value o f this counter is read using the RDTSC instruction.

The contents of the CpuSwPwrAcc register represents the cumulative *energy* consumed by the core. Each hardware-determined sample period (Tsample) a value that represents the energy consumed since the previous sample is added to the contents of this register. Tsample is on the order of a few microseconds. The exact value is immaterial because the CpuPwrSampleTimeRatio register provides the ratio of Tsample to the TSC period.

CpuSwPwrAcc is cleared to zero at power-on and is never reset. Therefore, it is possible for this counter to overflow and roll over to zero. To account for this, the interface provides the MaxCpuSwPwrAcc register. When read, this register provides a value that represents the maximum energy that the CpuSwPwrAcc register can report.

17.5.2 Software Algorithm

The following algorithm should be used to calculate the average power consumed by a processor core during the measurement interval T_M. To obtain a stable average power value, T_M should be on the order of several milliseconds.

- Determine the value of the ratio of Tsample to the TSC period (CpuPwrSampleTimeRatio) by executing CPUID Fn8000_0007. Call this value *N*.

  N = CPUID Fn8000_0007_ECX[31:0].
- Read the full range of the cumulative energy value from the register MaxCpuSwPwrAcc.

  Jmax = value returned from RDMSR MaxCpuSwPwrAcc.
- At time *x*, read CpuSwPwrAcc and the TSC

  J_x = value returned by RDMSR CpuSwPwrAcc

  T_x = value returned by RDTSC
- At time *y*, read CpuSwPwrAcc and the TSC again

  J_y = value returned by RDMSR CpuSwPwrAcc

  T_y = value returned by RDTSC

Calculate the average power consumption for the processor core over the measurement interval $T_{M} = (Ty – Tx)$.

- If ($J_y < J_x$), rollover has occurred; set $Jdelta = (J_y + Jmax) – J_x$
  else $Jdelta = J_y – J_x$
- PwrCPUave = $N * Jdelta / (T_y - T_x)$

Units of result is milliwatts.

18 Shadow Stacks

The shadow stack mechanism facilitates protection against a common form of computer exploit known as Return Oriented Programming (ROP). ROP exploits utilize intentionally corrupted stack frames to divert normal processor control flow into short fragments of existing executable code, which ultimately end with a RET instruction. These fragments are then chained together using return addresses previously written to the stack by the attacker.

The shadow stack mechanism protects against ROP exploits by ensuring that return addresses read from the stack by RET and IRET instructions originated from a CALL instruction or similar control transfer.

In the following sections, the term ‘program stack’ is used to distinguish the stack pointed to by the SP register and manipulated by instructions (such as PUSH, POP, ENTER, LEAVE) from the shadow stack.

18.1 Shadow Stack Overview

A shadow stack is a separate, protected stack that is conceptually parallel to the program stack and used only by control-transfer and return instructions. When shadow stacks are enabled, most control transfers that save a return address (such as CALL, INTn, exceptions and interrupts) write the return address to the shadow stack in addition to the program stack. Upon the subsequent RET or IRET operation, the processor reads the return address from both stacks and checks that they match. A control-protection exception (#CP) is raised if the return addresses do not match.

The shadow stack is implemented in regions of memory marked with the “shadow stack” attribute in the page tables (see “Page-Protection Checks” on page 158). Pages with the shadow stack attribute are writeable only by control transfer operations that save a return address and by the shadow stack management instructions. Shadow stack pages are not writeable by ordinary data-access instructions and thus protected from tampering.

18.1.1 Detecting and Enabling Shadow Stack Support

Support for the shadow stack feature is indicated by CPUID Fn0000_0007_ECX_x0[CET_SS](bit 7)=1. This bit also indicates the shadow stack MSRs are present.

The shadow stack feature is enabled by setting CR4.CET (bit 23) = 1 (see “CR4 Register” on page 47). The shadow stack feature is operational when in protected mode (CR0.PE=1) with paging enabled (CR0.PG=1). Shadow stacks are disabled in virtual x86 mode (rFLAGS.VM=1).

Once the feature is enabled, shadow stack operation can be separately enabled in user mode and in supervisor mode using control bits in the S_CET and U_CET MSRs (see “Shadow Stack MSRs” on page 659).

18.2 The Shadow Stack Pointer

On processors implementing the shadow stack feature, the Shadow Stack Pointer (SSP) register contains the address of the current top of the shadow stack. The width of the shadow stack is 64 bits in 64-bit mode and 32 bits in legacy and compatibility modes. The address size of SSP is 64 bits in 64-bit mode and 32 bits in legacy and compatibility modes.

The SSP register cannot be encoded as a source or destination register by regular instructions and thus is not directly accessible to software as a general operand, nor as an address operand. The SSP register is directly accessible only by using the shadow stack management instructions (see “Shadow Stack Management Instructions” on page 658).

18.3 Shadow Stack Operation for CALL (near) and RET (near)

When shadow stacks are enabled, the CALL (near) instruction pushes the return address onto the shadow stack, in addition to pushing it onto the program stack. On the subsequent near RET/RETn instruction, the return addresses are read from both stacks and compared. If the return addresses do not match, a control-protection (#CP) exception is generated.

A CALL (near) with a displacement of 0 (CALL +0) does not push the return address onto the shadow stack since the ’CALL +0’ idiom does not actually branch to another code sequence, and is typically not followed by a return instruction.

One form of the return instruction, RETn, pops ‘n’ additional parameters from the program stack before returning to the caller. Since the shadow stack does not store any data items, RETn does not pop additional parameters from the shadow stack.

18.4 Shadow Stack Operation for Far Transfers

The term ‘far transfer’ in the following discussion encompasses the following types of control transfers:

- Explicit CALL (far) to a procedure in another code segment
- Exceptions and interrupts that transfer to a handler in another code segment
- RET (far) and IRET instructions

A far transfer may also change the CPL. The exact operation of the shadow stack depends on the type of far transfer involved, the associated CPL change (if any) and whether the shadow stack capability is enabled for the target CPL. Shadow stack operations for far transfers are described in the following sections and summarized in Table 18-1. (See AMD Architecture Programmers Manual Volume 3 for detailed instruction algorithms for CALL (far), RET (far), IRET and INTn.)

Table 18-1. Shadow Stack Operations for Far Transfers

CALLF/int/excp From CPL	To CPL	Stack switch?	New SSP loaded from: (where n=new CPL)	CS, LIP saved to and restored from:	SSP saved to and restored from:	RETF/IRET checks returns address?
0,1,2,3	Same	No	Uses current SSP	Current Shadow Stack	Current Shadow Stack	Yes
3	0,1,2	Yes	PLn_SSP	<not saved/restored>	PL3_SSP	No
2	0,1	Yes	PLn_SSP	New shadow stack	New shadow stack	Yes
1	0	Yes	PLn_SSP	New shadow stack	New shadow stack	Yes

18.5 Far Transfer to the Same Privilege Level

When shadow stacks are enabled, a far transfer to the same privilege level pushes the CS and LIP (linear address of the return IP) onto the current shadow stack in addition to pushing the return IP onto the program stack. Upon the subsequent RET (far) or IRET operation, the return addresses are popped from both stacks and the linear forms of the two addresses are compared. If the return addresses do not match, a control-protection (#CP) exception is generated.

18.6 Far Transfer to Different Privilege Level

In addition to changing code segments, far transfers can also be used to change to a different privilege level (CPL). When changing the CPL, inter-privilege far transfers switch to a new program stack. In a similar manner, inter-privilege far transfers also switch to a new shadow stack, providing that shadow stacks are enabled for the new privilege level.

18.6.1 Shadow Stack Switching

When an inter-privilege far transfer switches to a new program stack, the new program stack pointer is selected from the either the inner-level stack pointers in the TSS or the Interrupt Stack Table (IST), depending on the type of far transfer. (See “Interrupt To Higher Privilege” on page 264). Similarly, when shadow stacks are enabled, inter-privilege far transfers also switch to a new shadow stack. The new shadow stack pointer is selected from shadow stack pointer MSRs (PLn_SSP) or the Interrupt Shadow Stack Table (ISST) as described in the following sections.

18.6.1.1 Shadow Stack Switching for Inter-Privilege CALL (far)

When a CALL (far) changes privilege level, a shadow stack switch occurs to an inner-level shadow stack. The new SSP is loaded from one of the following MSRs:

- PL2_SSP for transitions to CPL 2.
- PL1_SSP for transitions to CPL 1.
- PL0_SSP for transitions to CPL 0.

When switching shadow stacks, the processor validates the new shadow stack using a special value called a *shadow stack token* as described in Section 18.6.3 “Supervisor Shadow Stack Token” on page 651.

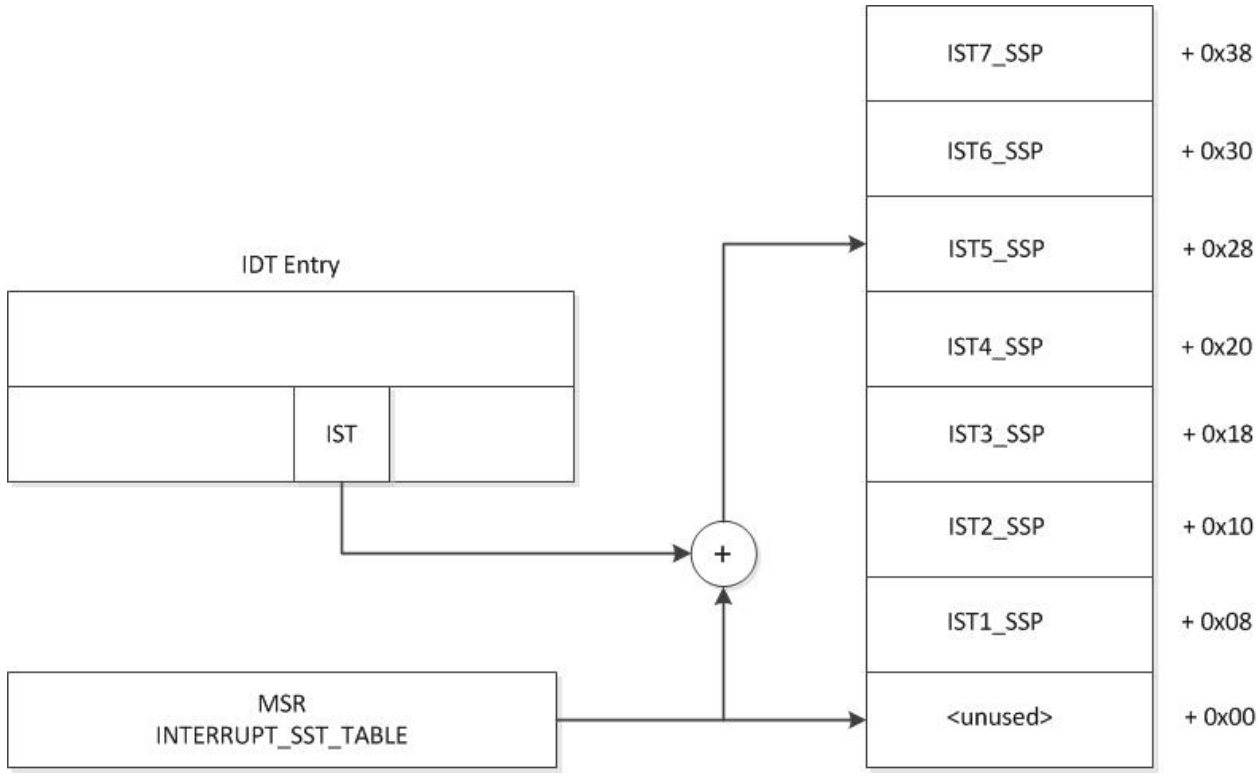
18.6.1.2 Shadow Stack Switching for Interrupts and Exceptions

In long mode (EFER.LMA=1), the processor provides an additional mechanism, the Interrupt Stack Table (IST) to switch program stacks for interrupts and exceptions. The IST mechanism uses the IST field in the Interrupt Descriptor Table (IDT) entry as an index into a table of inner-level program stack pointers (see “Interrupt-Stack Table” on page 276). The shadow stack feature provides a similar mechanism, the Interrupt Shadow Stack Table (ISST), for switching shadow stacks.

The ISST is an 8-entry table containing supervisor shadow stack pointers. The base of the ISST is specified by the INTERRUPT_SSP_TABLE MSR. If the IST field in the IDT entry is non-zero, it is used as an index into the ISST to select a new SSP as shown in Figure 18-1 “Interrupt Shadow Stack Table (ISST)” on page 650.

If the IST field is zero or if the processor is not in long mode, the ISST mechanism is not used and the new inner-level SSP is selected from the PLn_SSP MSRs as previously described in Section 18.6.4 “Shadow Stack Switching for Inter-Privilege CALL (far)” on page 649.

Figure 18-1. Interrupt Shadow Stack Table (ISST)



18.6.2 Handling CS, LIP and SSP on Privilege Transistions

When changing to a new code segment, the shadow stack behavior for saving and restoring CS, LIP and SSP depends on the CPL of the originating far transfer.

Transitions from CPL=3

Inter-privilege far transfers originating at CPL=3 save the current user-level SSP to PL3_SSP rather than to the supervisor shadow stack. Upon the subsequent RETF/IRET back to CPL=3, the user-level SSP is restored from PL3_SSP and the return CS and LIP are not verified.

Transitions from CPL=1 and CPL=2

Far transfers originating at CPL=1 or CPL=2 to an inner (more privileged) level save the current CS, LIP and SSP to the inner level shadow stack after switching shadow stacks as described in section Section 18.6.1, “Shadow Stack Switching,” on page 649. Upon the subsequent RET(far)/IRET, the SSP is restored (Section 18.6.1, “Shadow Stack Switching,” on page 649) from the inner-level shadow stack and the CS and LIP popped from the shadow stack are verified against the values read from the program stack. If the return addresses do not match, a control-protection (#CP) exception is generated.

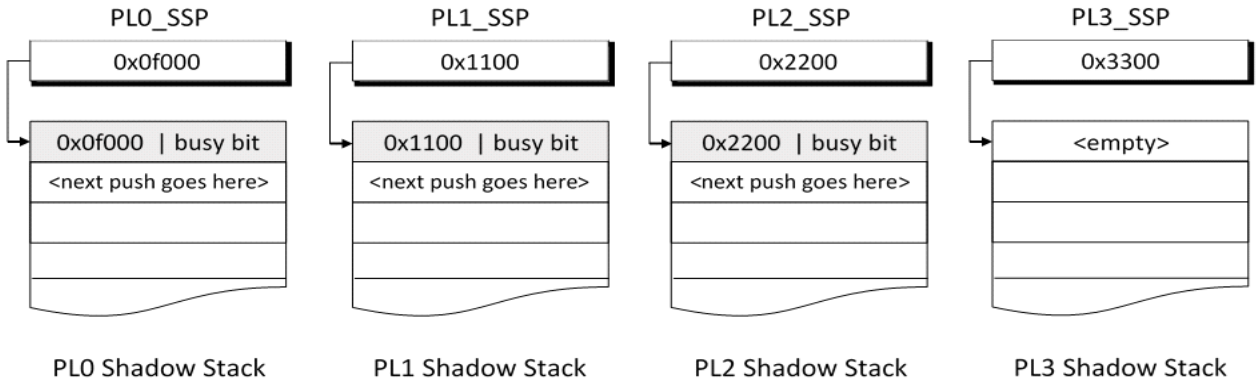
18.6.3 Supervisor Shadow Stack Token

When switching shadow stacks, the processor validates the new shadow stack by checking a *supervisor shadow stack token* located at the base of the stack.

When initially creating a shadow stack for use at privilege levels 0, 1 and 2, system software must place a supervisor shadow stack token at the base of each stack. System software must also store a pointer to each shadow stack in the appropriate shadow stack base register, PLn_SSP (n=0,1,2,3) using the WRMSR instruction.

The relationship of the shadow stacks, supervisor shadow stack tokens, and the PLn_SSP registers are shown in Figure 18.6.3 “Shadow Stacks and Supervisor Shadow Stack Tokens” on page 652. The three supervisor shadow stacks (PL0, PL1 and PL2) have shadow stack tokens stored at the base of the stack. Tokens are not used for PL3 shadow stacks.

Figure 18-2. Shadow Stacks and Supervisor Shadow Stack Tokens



The supervisor shadow stack token is a 64-bit value and is formatted as shown in Figure 18-3:

Figure 18-3. Supervisor Shadow Stack Token

The supervisor shadow stack token fields are defined as follows:

Linear Address. Bits 63:3 of the linear address of this token. The linear address is required to be 8-byte aligned.

Rsvd. Reserved, must be zero.

Busy Bit. If 0, this indicates this supervisor shadow stack is not in use by any logical processor. If 1, it indicates this supervisor shadow stack is currently in use by one of the logical processors. The processor sets the busy bit when switching to a supervisor shadow stack and clears it when switching away from that stack.

18.6.4 Shadow Stack Token Validation for Inter-privilege CALL (far) and Interrupts/Exceptions

Before switching shadow stacks, a CALL (far) or an interrupt/exception that changes to a more privileged level validates the supervisor shadow stack token located at the base of the new stack. If the validation checks pass, the supervisor shadow stack token is marked busy. The following steps are performed to validate the supervisor shadow stack token and set the busy bit:

1. Check that the incoming SSP is 8-byte aligned, otherwise a #GP exception is generated. The incoming SSP is located in PLn_SSP (where ‘n’ is the new privilege level: 0, 1, 2), or the ISST for interrupts/exceptions (see Section 18.6.1, “Shadow Stack Switching,” on page 649).

<Steps 2-6 are performed atomically.>

2. Fetch the 8-byte shadow stack token using the address specified by the incoming SSP (using a locked load with shadow stack access rights).
3. Check that the shadow stack token reserved bits and the busy bit are 0.
4. Check that the address specified in the shadow stack token matches the incoming SSP.
5. If checks 3 and 4 pass, set the supervisor shadow stack token busy bit (using a store, unlock).
6. If checks 3 or 4 fail, the shadow stack token is not modified and a #GP exception is generated.

18.6.5 Shadow Stack Token Validation for Inter-privilege RET and IRET

RET (far) and IRET to lower privilege levels validate the shadow stack token located on the current supervisor shadow stack before switching shadow stacks. If the token is valid, the token’s busy bit is cleared. The processor performs the following steps to validate the token:

1. Check that the return SSP (located on the current shadow stack) is 4-byte aligned, else a #CP exception is generated.

<Steps 2-7 are performed atomically.>

2. Fetch the 8-byte shadow stack token from the current shadow stack at SSP+24 (using a locked load with shadow stack access rights).
3. Check that the busy bit is 1.
4. Check that the reserved bits are 0.
5. Check that the address specified in the shadow stack token matches the SSP.
6. If checks 3 through 5 pass, the shadow stack token busy bit is cleared.
7. If any of checks 3 through 5 fail, the shadow stack token is not modified. A fault is not generated.

18.7 Shadow Stack Operation for SYSCALL and SYSRET

SYSCALL and SYSRET are low-latency system call and return instructions designed for use by system and application software implementing a flat-memory model. These instructions do not use the program stack to store return addresses, and therefore do not use the shadow stack to validate return addresses. However, SYSCALL and SYSRET modify SSP as described below. The Shadow stack operations described for SYSCALL and SYSRET also apply to SYSENTER and SYSEXIT respectively, although the latter are available only in legacy mode.

The SYSCALL instruction is used by application software executing at CPL=3. When shadow stacks are enabled at CPL=3, SYSCALL saves the current SSP to PL3_SSP. SYSCALL changes to CPL=0

before entering the operating system and if shadow stacks are enabled at CPL=0, then SSP is cleared to 0.

Unlike other inter-privilege far transfers, SYSCALL does not automatically perform a switch to a supervisor shadow stack. If shadow stacks are enabled at CPL=0, prior to executing a CALL instruction or similar transfer of control that pushes a return address to the shadow stack, software at the OS entry point must ensure that a supervisor shadow stack is available for use. System software can use SETSSBSY to set up a supervisor shadow stack.

The operating system uses the SYSRET instruction to return to the application running at CPL=3. If shadow stacks are enabled at CPL=3, SYSRET restores SSP from PL3_SSP. Prior to using SYSRET to return to the application, system software can use the CLRSSBSY to tear down the supervisor shadow stack. Because CLRSSYBSY clears the SSP to 0, system software must ensure that any subsequent interrupt or exception that may occur in CPL=0 prior to the SYSRET is configured to use the ISST stack-switching mechanism. Otherwise taking an interrupt or exception with SSP=0 will likely result in a fault due to SSP wrap-around.

18.8 Shadow Stack Operation for Task Switches

The legacy x86 task-switch mechanism transfers program control to a new task when any of the following control transfers occur:

- A CALL or JMP instruction references a task gate or TSS descriptor.
- A software-interrupt instruction (INTn), exception or external interrupt references a task gate.
- An IRET is executed when the EFLAGS.NT bit is set to 1.

Shadow stack operations for legacy x86 task switches are summarized in this section. Because the x86 task management feature is supported by the AMD64 architecture only in legacy mode (EFER.LMA=0), the shadow stack operations described below only apply to legacy mode. (See “Switching Tasks” on page 375 for more information on task switching).

When switching to a new task with shadow stacks enabled, the new task must use a 32-bit TSS. The SSP for the new task is located at TSS offset 104. Since the SSP is 4 bytes in legacy mode, the TSS must be at least 108 bytes in size. The SSP must be aligned to an 8-byte boundary and point to a supervisor shadow stack token.

If the task switch is initiated by a CALL/JMP/INTn instruction, or an interrupt or exception:

- For task switches originating at CPL=3, and if shadow stacks are enabled at that CPL, the current SSP is saved to PL3_SSP. Otherwise, for task switches originating at supervisor-level (CPL=0,1,2) the current SSP is saved onto the new shadow stack along with current CS and LIP. If shadow stacks are enabled at the CPL of the new task, the busy bit is set in the supervisor shadow stack token pointed to by the SSP of the incoming task.

If the task switch is initiated by an IRET instruction:

- For task switches originating at supervisor-level (CPL=0,1,2) and returning to CPL=3, the SSP is restored from PL3_SSP, otherwise the SSP is restored from the current shadow stack. The return CS and LIP are read from the current shadow stack and compared to the CS and linear form of the EIP in the TSS of the incoming task. A control-protection (#CP) exception is generated if the return addresses do not match.

18.9 Restricting Speculative Execution of RET targets

When shadow stacks are enabled, the processor will not speculatively execute instructions from a RET address on the program stack unless the address matches the corresponding address on the shadow stack, or the target of the RET is predicted by a return address branch prediction mechanism.

18.10 Shadow Stack Switching Using RSTORSSP

As previously described in Section 18.6.1, “Shadow Stack Switching,” on page 649, the processor automatically switches shadow stacks as part of the inter-privilege far transfer mechanism. In order to allow programs to switch stacks at other times (when performing operations such as initializing a shadow stack or recovering from a #CP fault) the RSTORSSP and SAVEPREVSSP instructions are provided.

The RSTORSSP instruction is used to switch shadow stacks. The instruction expects to find a shadow stack restore token at the top of the proposed new shadow stack. Upon validating this token, RSTORSSP points the SSP to the top of the new shadow stack and sets the token’s busy bit, making the new shadow stack ready for use.

Upon successful completion, the RSTORSSP instruction modifies the restore token by saving the old SSP into the token’s SSP Restore Value field in order to facilitate a later return to the old shadow stack.

The shadow stack restore token used by RSTORSSP is formatted as follows:

Figure 18-4. Shadow Stack Restore Token

The shadow stack restore token fields are defined as follows:

SSP Restore value. Bits 63:2 of the linear address of this token. Replaced by the previous SSP upon successful completion of the RSTORSSP instruction.

Busy. Initially must be 0. Set to 1 upon successful completion of the RSTORSSP instruction.

Mode. If 0, this indicates the shadow stack restore token is for use in legacy or compatibility mode. If 1, this indicates the restore token is for use in 64-bit mode.

After RSTORSSP switches to the new stack the modified shadow stack token is at the top of the new in-use shadow stack. If a return to the old stack is not required, the modified token (to which the previous SSP has been saved) can be popped from the stack using the INCSSP instruction.

If a return to the old stack is desired, the SAVEPREVSSP instruction can be used to copy the token back to the previous stack for later use by an RSTORSSP instruction. The SAVEPREVSSP expects to find a *previous SSP token* on the top of the current shadow stack. After moving this token to the previous stack, SAVEPREVSSP pops it off the new stack by incrementing SSP.

The previous SSP token used by SAVEPREVSSP is formatted as follows:

Figure 18-5. Previous SSP Token

The previous SSP token fields are defined as follows:

Previous SSP. Bits 63:2 of the previous SSP. Upon successful completion, the SAVEPREVSSP instruction copies the token to this address.

Busy. Must be set to 1 initially. Cleared to 0 upon successful completion of the SAVEPREVSSP instruction.

Mode. If 0, this indicates the previous SSP token is for use in legacy or compatibility mode. If 1, this indicates the token is for use in 64-bit mode.

Figure 18-6 “RSTORSSP and SAVEPREVSSP Operation” on page 657 illustrates the operation of the RSTORSSP instruction when switching to a new shadow stack, followed by a SAVEPREVSSP instruction to save a previous token back to the original stack.

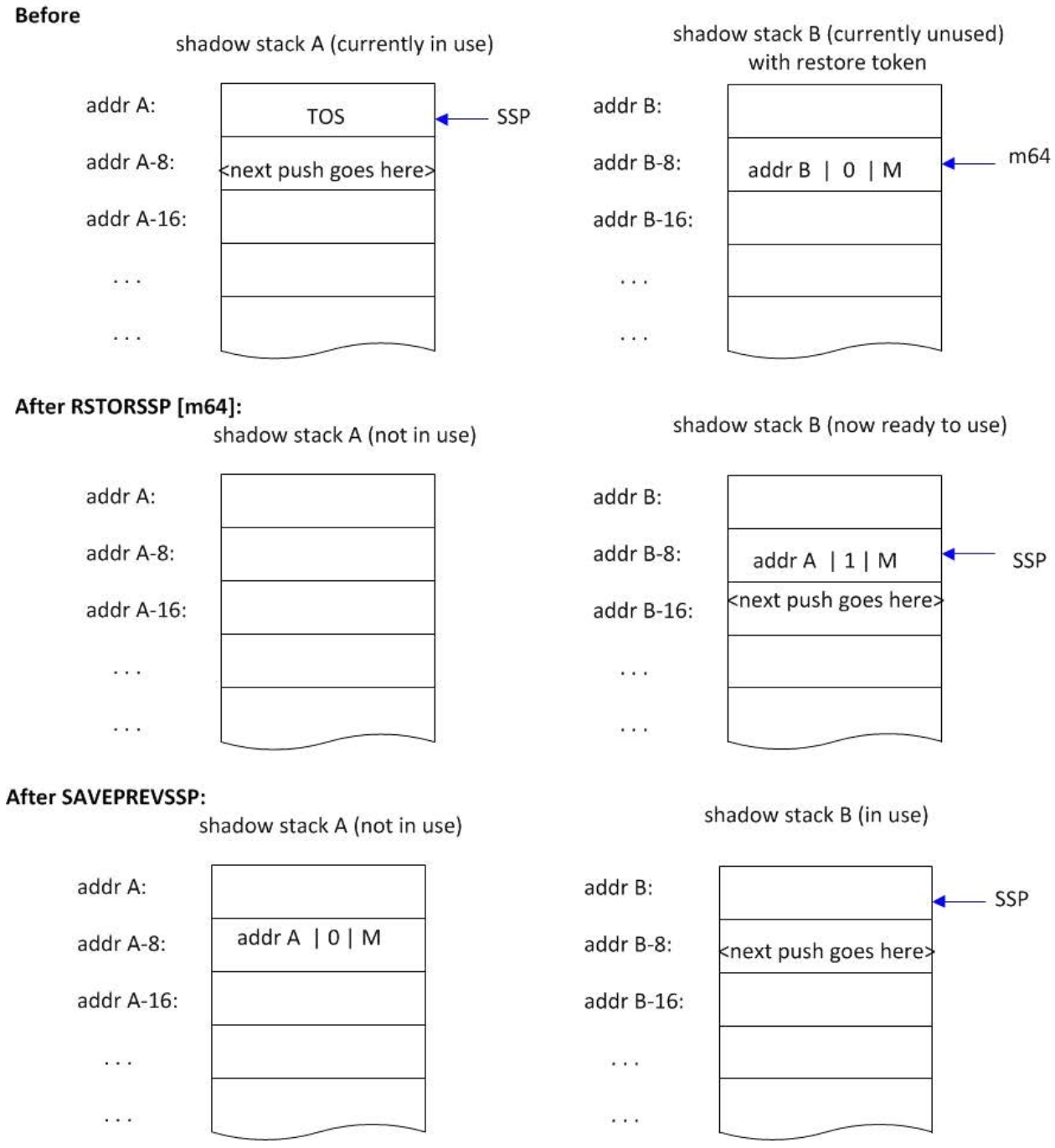


Figure 18-6. RSTORSSP and SAVEPREVSSP Operation

In this example (Figure 18-6 “RSTORSSP and SAVEPREVSSP Operation” on page 657), RSTORSSP is used to switch from shadow stack A to shadow stack B. Subsequently, a SAVEPREVSSP instruction is used to save a shadow stack restore token.

Initially, shadow stack A is in-use, and the proposed shadow stack B contains a shadow stack restore token at address B - 8. An RSTORSSP instruction is then executed with the operand pointing to the restore token.

The RSTORSSP instruction checks the restore token for validity, copies the previous SSP to the token, sets bit 1 of the token and sets the new value of SSP to address B. Shadow stack B is now ready for use.

Next, a SAVEPREVSSP instruction is executed to facilitate a later return to shadow stack A (using a subsequent RSTORSSP instruction, not shown). The SAVEPREVSSP instruction copies the token to address A - 8 and clears token bit 1, then pops the old token from shadow stack B.

For detailed RSTORSSP and SAVEPREVSSP algorithms refer to the instruction descriptions in APM Volume 3.

18.11 Shadow Stack Management Instructions

When shadow stacks are enabled, the following instructions are available to software for use in managing shadow stacks. Except for RDSSP, attempting to execute these instructions when shadow stacks are disabled results in a #UD exception. RDSSP is treated as a NOP when shadow stacks are disabled. For more information refer to the detailed instruction descriptions in APM volume 3.

Table 18-2. Shadow Stack Management Instructions

Mnemonic	Name	Description
CLRSSBSY	Clear Shadow Stack Busy	Validates a shadow stack token and clears its busy bit. This is a privileged instruction.
INCSSP	Increment Shadow Stack Pointer	Increment SSP by ‘n’ stack frames. Used to pop unneeded items from a shadow stack.
RDSSP	Read Shadow Stack Pointer	Read the SSP into a GPR. Treated as a NOP if shadow stacks are disabled.
RSTORSSP	Restore Shadow Stack Pointer	Used to switch shadow stacks. Expects a ‘shadow stack restore token’ at the top of the new shadow stack. Upon validating this token, sets the token’s busy bit and sets SSP to the top of the new shadow stack.
SAVEPREVSSP	Save Previous Shadow Stack Pointer	Copies a ‘previous SSP token’ from the current shadow stack back to the previous stack for later use by an RSTORSSP instruction.

Table 18-2. Shadow Stack Management Instructions (continued)

Mnemonic	Name	Description
SETSSBSY	Set Shadow Stack Busy	Validates the shadow stack token pointed to by the PL0_SSP MSR. If valid, clears the busy bit and sets SSP = PL0_SSP. This is a privileged instruction.
WRSS	Write Shadow Stack	Writes the source operand to a shadow stack. This instruction has associated enable bits in the U_CET and S_CET MSRs
WRUSS	Write User Shadow Stack	Writes the source operand to a user shadow stack. This is a privileged instruction.

18.12 Shadow Stack MSRs

The following MSRs are defined if the shadow stack feature is supported as indicated by CPUID Fn 0000_0007_0 ECX[CET_SS] (bit 7) = 1:

U_CET. MSR 0x6A0. Specifies the user mode shadow stack controls. The individual fields are as follows:

- Bit 0 - SH_STK_EN. When set to 1, enables the shadow stacks in user mode.
- Bit 1 - WR_SHSTK_EN. When set to 1, enables the WRSS instruction is user mode.
- Bits 63:2 – reserved, MBZ

S_CET. MSR 0x6A2. Specifies the supervisor mode shadow stack controls.

- Bit 0 - SH_STK_EN. When set to 1, enables the shadow stacks in supervisor mode.
- Bit 1 - WR_SHSTK_EN. When set to 1, enables the WRSS instruction is supervisor mode.
- Bits 63:2 – reserved, MBZ

PL0_SSP. MSR 0x6A4. Specifies the linear address to be loaded into SSP on the next transition to CPL=0. The linear address must be in canonical format and aligned to 4 bytes when initializing this register.

PL1_SSP. MSR 0x6A5. Specifies the linear address to be loaded into SSP on the next transition to CPL=1. The linear address must be in canonical format and aligned to 4 bytes when initializing this register.

PL2_SSP. MSR 0x6A6. Specifies the linear address to be loaded into SSP on the next transition to CPL=2. The linear address must be in canonical format and aligned to 4 bytes when initializing this register.

PL3_SSP. MSR 0x6A7. The user mode SSP is saved to and restored from this register. The linear address must be in canonical format and aligned to 4 bytes when initializing this register.

ISST_ADDR. MSR 0x6A8. This register specifies the linear address of the Interrupt Shadow Stack Table (ISST). The linear address must be in canonical format.

18.13 XSAVE/XRSTOR

The XSAVE/XRSTOR instructions can be used to manage the shadow stack registers as processor extended state components 11 (CET_U state) and 12 (CET_S state) as described below:

- **CET_U state.** The shadow stack user controls are identified as state component 11 in XFEATURE_ENABLED_MASK and consist of the two 64-bit MSRs U_CET and PL3_SSP. Bytes 7:0 in the XSAVE area are used for U_CET and bytes 15:8 are used for PL3_SSP. The XSAVE area size and offset for the CET_U state are available using CPUID Fn0000_000D_ECX_x0B (ECX=11).
- **CET_S state.** The shadow stack supervisor controls are identified as state component 12 in XFEATURE_ENABLED_MASK and consist of the three 64-bit MSRs PL0_SSP, PL1_SSP and PL2_SSP. Bytes 7:0 in the XSAVE area are used for PL0_SSP, bytes 15:8 are used for PL2_SSP and bytes 23:16 are used for PL3_SSP. The XSAVE area size and offset for the CET_S state are available using CPUID Fn0000_000D_ECX_x0C (ECX=12).

The CET_U and CET_S state components are managed by XSAVE/XRSTOR by setting bits 11 and 12 respectively in XFEATURE_ENABLE_MASK. See Section 11.5.2, “XFEATURE_ENABLED_MASK,” on page 349 for more details. When restoring CET_U and CET_S state, XRSTOR checks for reserved bits and canonicality as described in Section 18.11 “Shadow Stack Management Instructions” on page 658.

System software can enable XSAVES/XRSTORS management of the CET_U and CET_S state components by setting bits 11 and 12 respectively in the XSS MSR.

Appendix A MSR Cross-Reference

This appendix lists the MSRs that are defined in the AMD64 architecture. The AMD64 architecture supports some of the same MSRs as previous versions of the x86 architecture and implementations thereof. Where possible, the AMD64 architecture supports the same MSRs, for the same functions, as these previous architectures and implementations.

The first section lists the MSRs according to their MSR address, and it gives a cross reference for additional information. The remaining sections list the MSRs by their functional group. Those sections also give a brief description of the register and specify the register reset value.

Some MSRs are implementation-specific For information about these MSRs, see the documentation for specific implementations of the AMD64 architecture.

A.1 MSR Cross-Reference by MSR Address

Table A-1 lists the MSRs in the AMD64 architecture in order of MSR address.

Table A-1. MSRs of the AMD64 Architecture

MSR Address	MSR Name	Functional Group	Cross-Reference
0010h	TSC	Performance	“Time-Stamp Counter” on page 407
001Bh	APIC_BASE	System Software	“Local APIC Enable” on page 603
0048h	SPEC_CTRL	Speculation Control	“Speculation Control MSRs” on page 67
0049h	PRED_CMD	Speculation Control	“Speculation Control MSRs” on page 67
00E7h	MPERF	Performance	“Determining Processor Effective Frequency” on page 642
00E8h	APERF	Performance	“Determining Processor Effective Frequency” on page 642
00FEh	MTRRcap	Memory Typing	“Identifying MTRR Features” on page 215
0174h	SYSENTER_CS	System Software	“SYSENTER and SYSEXIT MSRs” on page 173
0175h	SYSENTER_ESP		
0176h	SYSENTER_EIP		
0179h	MCG_CAP	Machine Check	“Machine-Check Global-Capabilities Register” on page 292
017Ah	MCG_STATUS		“Machine-Check Global-Status Register” on page 293
017Bh	MCG_CTL		“Machine-Check Global-Control Register” on page 294
01D9h	DebugCtl	Software Debug	“Debug-Control MSR (DebugCtl)” on page 391

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
01DBh	LastBranchFromIP	Software Debug	"Control-Transfer Recording MSRs" on page 393
01DCh	LastBranchToIP		
01DDh	LastIntFromIP		
01DEh	LastIntToIP		
0200h	MTRRphysBase0	Memory Typing	"Variable-Range MTRRs" on page 212
0201h	MTRRphysMask0		
0202h	MTRRphysBase1		
0203h	MTRRphysMask1		
0204h	MTRRphysBase2		
0205h	MTRRphysMask2		
0206h	MTRRphysBase3		
0207h	MTRRphysMask3		
0208h	MTRRphysBase4		
0209h	MTRRphysMask4		
020Ah	MTRRphysBase5		
020Bh	MTRRphysMask5		
020Ch	MTRRphysBase6		
020Dh	MTRRphysMask6		
020Eh	MTRRphysBase7		
020Fh	MTRRphysMask7		
0250h	MTRRfix64K_00000	Memory Typing	"Fixed-Range MTRRs" on page 210
0258h	MTRRfix16K_80000		
0259h	MTRRfix16K_A0000		
0268h	MTRRfix4K_C0000		
0269h	MTRRfix4K_C8000		
026Ah	MTRRfix4K_D0000		
026Bh	MTRRfix4K_D8000		
026Ch	MTRRfix4K_E0000		
026Dh	MTRRfix4K_E8000		
026Eh	MTRRfix4K_F0000		
026Fh	MTRRfix4K_F8000		
0277h	PAT	Memory Typing	"PAT Register" on page 218
02FFh	MTRRdefType		"Default-Range MTRRs" on page 214

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
0400h	MC0_CTL	Machine Check	See the documentation for particular implementations of the architecture.
0404h	MC1_CTL		
0408h	MC2_CTL		
040Ch	MC3_CTL		
0410h	MC4_CTL		
0414h	MC5_CTL		
0401h	MC0_STATUS	Machine Check	“Machine-Check Status Registers” on page 299
0405h	MC1_STATUS		
0409h	MC2_STATUS		
040Dh	MC3_STATUS		
0411h	MC4_STATUS		
0415h	MC5_STATUS		
0402h	MC0_ADDR	Machine Check	“Machine-Check Address Registers” on page 302
0406h	MC1_ADDR		
040Ah	MC2_ADDR		
040Eh	MC3_ADDR		
0412h	MC4_ADDR		
0416h	MC5_ADDR		
0403h	MC0_MISC	Machine Check	“Machine-Check Miscellaneous-Error Information Register 0 (MC*i*_MISC0)” on page 302
0407h	MC1_MISC		
040Bh	MC2_MISC		
040Fh	MC3_MISC		
0413h	MC4_MISC		
0417h	MC5_MISC		
06A0h	U_CET	Shadow Stack	“Shadow Stack MSRs” on page 677.
06A2h	S_CET		
06A4h	PL0_SSP		
06A5h	PL1_SSP		
06A6h	PL2_SSP		
06A7h	PL3_SSP		
06A8h	ISST_ADDR		
0DA0h	XSS	-	XSAVES and XSTRORS instructions in APM volume 4.
C000_0080h	EFER	System Software	“Extended Feature Enable Register (EFER)” on page 56

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
C000_0081h	STAR	System Software	"SYSCALL and SYSRET MSRs" on page 172
C000_0082h	LSTAR	System Software	
C000_0083h	CSTAR	System Software	
C000_0084h	SF_MASK	System Software	
C000_00E7h	MPerfReadOnly	Performance	"MPERF Read-only (MperfReadOnly)" on page 644
C000_00E8h	APerfReadOnly	Performance	"APERF Read-only (AperfReadOnly)" on page 644
C000_00E9h	IRPerfCount	Performance	"Instructions Retired Performance counter" on page 406
C000_0100h	FS.Base	System Software	"FS and GS Registers in 64-Bit Mode" on page 80
C000_0101h	GS.Base	System Software	
C000_0102h	KernelGSbase	System Software	"SWAPGS Instruction" on page 173
C000_0103h	TSC_AUX	System Software	"RDTSCP Instruction" on page 176
C000_0104h	TSC Ratio	SVM	see "TSC Ratio MSR (C000_0104h)" on page 568
C000_0108h	PrefetchControl	—	Controls enabling / disabling hardware prefetchers. See appropriate BIOS and Kernel Developer's Guide (BKDG) or Processor Programming Reference (PPR) Manual for details.
C000_0408h	MC4_MISC1	Machine Check	"Machine-Check Miscellaneous-Error Information Register 0 (MC*i*_MISC0)" on page 302
C000_0409h	MC4_MISC2		
C000_040Ah	MC4_MISC3		
C000_0410h	MCA_INTR_CFG		MCA related interrupt configuration. See appropriate BIOS and Kernel Developer's Guide (BKDG) or Processor Programming Reference (PPR) Manual for details.
C000_2000h + i*10h	MCA_CTL		See the documentation for particular implementations of the architecture.
C000_2001h + i*10h	MCA_STATUS		"Machine-Check Status Registers" on page 299.
C000_2002h + i*10h	MCA_ADDR		"Machine-Check Address Registers" on page 302.
C000_2003h + i*10h	MCA_MISC0		"Machine-Check Miscellaneous-Error Information Register 0 (MC*i*_MISC0)" on page 302.
C000_2004h + i*10h	MCA_CONFIG		"MCA Configuration Register" on page 307.
C000_2005h + i*10h	MCA_IPID		"MCA IP Identification" on page 308.
C000_2006h + i*10h	MCA_SYND		"MCA Syndrome Register" on page 309.

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
C000_2008h + i*10h	MCA_DESTAT	Machine Check	“MCA Deferred Error Status Register” on page 309.
C000_2009h + i*10h	MCA_DEADDR		“MCA Deferred Error Address Register” on page 310.
(C000_200Ah : C000_200Dh) + i*10h	MCA_MISC[4-1]		“MCA Miscellaneous Registers 1 - 4” on page 310.
(C000_200Eh : C000_200Fh)+ i*10h	MCA_SYND[2-1]		“MCA Syndrome Registers 1 - 2” on page 311.
C001_0000h	PerfEvtSel0	Performance	“Core Performance Event-Select Registers” on page 402
C001_0001h	PerfEvtSel1		
C001_0002h	PerfEvtSel2		
C001_0003h	PerfEvtSel3		
C001_0004h	PerfCtr0	Performance	“Performance Counter MSRs” on page 401
C001_0005h	PerfCtr1		
C001_0006h	PerfCtr2		
C001_0007h	PerfCtr3		
C001_0010h	SYSCFG	Memory Typing	“System Configuration Register (SYSCFG)” on page 61
C001_0015h	HWCR	System Software	“Hardware Configuration Register (HWCR)” on page 71
C001_0016h	IORRBase0	Memory Typing	“IORRs” on page 224
C001_0017h	IORRMask0		
C001_0018h	IORRBase1		
C001_0019h	IORRMask1		
C001_001Ah	TOP_MEM	Memory Typing	“Top of Memory” on page 226
C001_001Dh	TOP_MEM2		
C001_0030h	Processor_Name_String	CPUID Name	See appropriate *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* for details.
C001_0031h			
C001_0032h			
C001_0033h			
C001_0034h			
C001_0035h			
C001_0056h	SMI_Trigger_IO_Cycle	SMM	See appropriate *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* for details.

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
C001_0061h	P-State Current Limit	SMM	"Hardware Performance Monitoring and Control" on page 639
C001_0062h	P-State Control		
C001_0063h	P-State Status		
C001_0074h	CPU_Watchdog_Timer	Machine Check	"CPU Watchdog Timer Register" on page 295
C001_0104h	TSC Ratio	SVM	"TSC Ratio MSR (C000_0104h)" on page 568
C001_0111h	SMBASE	SMM	"SMBASE Register" on page 319
C001_0112h	SMM_ADDR		"SMRAM Protected Areas" on page 325
C001_0113h	SMM_MASK		
C001_0114h	VM_CR	SVM	"SVM Related MSRs" on page 566
C001_0115h	IGNNE	SVM	"SVM Related MSRs" on page 566
C001_0116h	SMM_CTL	SVM	"SVM Related MSRs" on page 566
C001_0117h	VM_HSAVE_PA	SVM	"SVM Related MSRs" on page 566
C001_0118h	SVM_KEY_MSR	SVM	"SVM-Lock" on page 569
C001_0119h	SMM_KEY_MSR	SMM	"SMM-Lock" on page 570
C001_011Ah	Local_SMI_Status	SMM	See appropriate *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* for details.
C001_011Bh	Doorbell Register	SVM	"Doorbell Register" on page 563
C001_011Eh	VMPAGE_FLUSH	SVM	"Secure Encrypted Virtualization" on page 571
C001_011Fh	VIRT_SPEC_CTRL	Speculation Control	"Speculation Control MSRs" on page 67
C001_0130h	GHCB	SVM	"Guest-HV Communication Block" (see "GHCB" on page 583)
C001_0131h	SEV_STATUS	SVM	"SEV_STATUS MSR" (see "SEV_STATUS MSR" on page 577)
C001_0132h	RMP_BASE	SVM	"Initializing the RMP" (see "Initializing the RMP" on page 588)
C001_0133h	RMP_END	SVM	"Initializing the RMP" (see "Initializing the RMP" on page 588)
C001_0134h	GUEST_TSC_FREQ	SVM	"Secure TSC" (see "Secure TSC" on page 600)
C001_0140h	OSVW_ID_Length	OSVW	"OS-Visible Workarounds" on page 703
C001_0141h	OSVW Status		

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
C001_0200h	PerfEvtSel0	Performance	"Core Performance Event-Select Registers" on page 402
C001_0202h	PerfEvtSel1		
C001_0204h	PerfEvtSel2		
C001_0206h	PerfEvtSel3		
C001_0208h	PerfEvtSel4		
C001_020Ah	PerfEvtSel5		
C001_0201h	PerfCtr0		"Performance Counter MSRs" on page 401
C001_0203h	PerfCtr1		
C001_0205h	PerfCtr2		
C001_0207h	PerfCtr3		
C001_0209h	PerfCtr4		
C001_020Bh	PerfCtr5		
C001_0230h	L2I_PerfEvtSel0		"Performance-Monitoring MSRs" on page 673
C001_0232h	L2I_PerfEvtSel1		
C001_0234h	L2I_PerfEvtSel2		
C001_0236h	L2I_PerfEvtSel3		
C001_0231h	L2I_PerfCtr0		
C001_0233h	L2I_PerfCtr1		
C001_0235h	L2I_PerfCtr2		
C001_0237h	L2I_PerfCtr3		
C001_0240h	NB_PerfEvtSel0		
C001_0242h	NB_PerfEvtSel1		
C001_0244h	NB_PerfEvtSel2		
C001_0246h	NB_PerfEvtSel3		
C001_0241h	NB_PerfCtr0		
C001_0243h	NB_PerfCtr1		
C001_0245h	NB_PerfCtr2		
C001_0247h	NB_PerfCtr3		
C001_1019h	DR1_ADDR_MASK	Software Debug	"Debug Breakpoint Address Masking" on page 400
C001_101Ah	DR2_ADDR_MASK		
C001_101Bh	DR3_ADDR_MASK		
C001_1027h	DR0_ADDR_MASK		

A.2 System-Software MSRs

Table A-2 lists the MSRs defined for general use by system software in controlling long mode and in allowing fast control transfers between applications and the operating system.

Table A-2. System-Software MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
0000_001Bh	APIC_BASE	See appropriate *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* for details.	0000_0000_FEE0_0x00h
C000_0080h	EFER	Contains control bits that enable extended features supported by the processor, including long mode.	0000_0000_0000_0000h
C000_0081h	STAR	In legacy mode, used to specify the target address of a SYSCALL instruction, as well as the CS and SS selectors of the called and returned procedures.	undefined
C000_0082h	LSTAR	In 64-bit mode, used to specify the target RIP of a SYSCALL instruction.	undefined
C000_0083h	CSTAR	In compatibility mode, used to specify the target RIP of a SYSCALL instruction.	undefined
C000_0084h	SF_MASK	SYSCALL Flags Mask	undefined
C000_0100h	FS.Base	Contains the 64-bit base address in the hidden portion of the FS register (the base address from the FS descriptor).	0000_0000_0000_0000h
C000_0101h	GS.Base	Contains the 64-bit base address in the hidden portion of the GS register (the base address from the GS descriptor).	0000_0000_0000_0000h
C000_0102h	KernelGSbase	The SWAPGS instruction exchanges the value in KernelGSbase with the value in GS.base, providing a fast method for system software to load a pointer to system data-structures.	undefined
C000_0103h	TSC_AUX	The RDTSCP instruction copies the value of this MSR into the ECX register.	0000_0000_0000_0000h
C000_0104h	TSC_RATIO	Specifies the TSCRatio value which is used to scale the TSC value read by a Guest.	0000_0001_0000_0000h
0174h	SYSENTER_CS	In legacy mode, used to specify the CS selector of the procedure called by SYSENTER.	undefined
0175h	SYSENTER_ESP	In legacy mode, used to specify the stack pointer for the procedure called by SYSENTER.	undefined
0176h	SYSENTER_EIP	In legacy mode, used to specify the EIP of the procedure called by SYSENTER.	undefined

A.3 Memory-Typing MSRs

Table A-3 lists the MSRs used to control memory-typing and the page-attribute-table mechanism.

Table A-3. Memory-Typing MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
00FEh	MTRRcap	A *read-only* register containing information describing the level of MTRR support provided by the processor.	0000_0000_0000_0508h
0200h	MTRRphysBase0	Specifies the memory-range base address in physical-address space of a variable-range memory region. These registers also specify the memory type used for the memory region.	undefined
0202h	MTRRphysBase1		
0204h	MTRRphysBase2		
0206h	MTRRphysBase3		
0208h	MTRRphysBase4		
020Ah	MTRRphysBase5		
020Ch	MTRRphysBase6		
020Eh	MTRRphysBase7		
0201h	MTRRphysMask0	Specifies the size of a variable-range memory region.	Valid (bit 11) = 0 All Other Bits Undefined
0203h	MTRRphysMask1		
0205h	MTRRphysMask2		
0207h	MTRRphysMask3		
0209h	MTRRphysMask4		
020Bh	MTRRphysMask5		
020Dh	MTRRphysMask6		
020Fh	MTRRphysMask7		
0250h	MTRRfix64K_00000	Fixed-range MTRRs used to characterize the first 1 Mbyte of physical memory. Each 64-bit register contains eight type fields for characterizing a total of eight memory ranges. • MTRRfix64K_n characterizes 64 Kbyte ranges. • MTRRfix16K_n characterizes 16 Kbyte ranges. • MTRRfix4K_n characterizes 4 Kbyte ranges.	undefined
0258h	MTRRfix16K_80000		
0259h	MTRRfix16K_A0000		
0268h	MTRRfix4K_C0000		
0269h	MTRRfix4K_C8000		
026Ah	MTRRfix4K_D0000		
026Bh	MTRRfix4K_D8000		
026Ch	MTRRfix4K_E0000		
026Dh	MTRRfix4K_E8000		
026Eh	MTRRfix4K_F0000		
026Fh	MTRRfix4K_F8000		
0277h	PAT	Used to extend the page-table entry format, allowing memory-type characterization on a physical-page basis.	0007_0406_0007_0406h

Table A-3. Memory-Typing MSR Cross-Reference (continued)

MSR Address	MSR Name	Description	Reset Value
02FFh	MTRRdefType	Sets the default memory-type for physical addresses not within ranges established by fixed-range and variable-range MTRRs.	0000_0000_0000_0000h
C001_0010h	SYSCFG	Contains control bits for enabling and configuring system bus features.	0000_0000_0002_0601h
C001_0016h	IORRBase0	Specifies the memory-range base address in physical-address space of a variable-range I/O region.	undefined
C001_0018h	IORRBase1		
C001_0017h	IORRMask0	Specifies the size of a variable-range I/O region.	Valid (bit 11) = 0 All Other Bits Undefined
C001_0019h	IORRMask1		
C001_001Ah	TOP_MEM	Sets the boundary between system memory and memory-mapped I/O for addresses below 4 Gbytes.	0000_0000_0400_0000h
C001_001Dh	TOP_MEM2	Sets the boundary between system memory and memory-mapped I/O for addresses above 4 Gbytes.	undefined

A.4 Machine-Check MSRs

Table A-4 lists the MSRs used in support of the machine-check mechanism.

Table A-4. Machine-Check MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
0179h	MCG_CAP	A read-only register that specifies the machine-check mechanism capabilities supported by the processor.	0000_0000_0000_010xh
017Ah	MCG_STATUS	Provides basic information about the processor state immediately after the occurrence of a machine-check error.	undefined
017Bh	MCG_CTL	Controls global reporting of machine-check errors from various sources.	0000_0000_0000_0000h
0400h + 4i for 0 <=i <=31	MCi_CTL	Control for error reporting banks, per implementation.	0000_0000_0000_0000h
0401h + 4i	MCi_STATUS[5:0]	Status registers for each error-reporting register bank, used to report machine-check error information for the specified register bank.	undefined
0402h + 4i	MCi_ADDR[5:0]	Reports the instruction memory-address or data memory-address responsible for the machine-check error for the specified register bank.	undefined
0403h + 4i	MCi_MISC[5:0]	Reports miscellaneous information about the machine-check error for the specified register bank.	c00x_xxxx_xx00_0000
C000_0408h	MC4_MISC1		c00x_xxxx_0000_0000
C000_0409h	MC4_MISC2		
C000_040Ah	MC4_MISC3		
C000_0410h	MCA_INTR_CFG	MCA interrupt configuration.	0000_0000_0000_0000h
C000_2000h + i*10h	MCA_CTL	Control for error reporting banks per implementation.	0000_0000_0000_0000h
C000_2001h + i*10h	MCA_STATUS	Status registers for each error-reporting register bank, used to report machine-check error information for the specified register bank.	undefined
C000_2002h + i*10h	MCA_ADDR	Reports the instruction memory-address or data memory-address responsible for the machine-check error for the specified register bank.	undefined

Table A-4. Machine-Check MSR Cross-Reference (continued)

MSR Address	MSR Name	Description	Reset Value
C000_2003h + i*10h	MCA_MISC0	Reports miscellaneous information about the machine-check error for the specified register bank.	c00x_xxxx_xx00_0000
C000_2004h + i*10h	MCA_CONFIG	Controls configuration information of each MCA bank.	See BKDG/PPR
C000_2005h + i*10h	MCA_IPID	Holds information to identify the MCA bank type	See BKDG/PPR
C000_2006h + i*10h	MCA_SYND	Holds syndrome associated with the error	See BKDG/PPR
C000_2008h + i*10h	MCA_DESTAT	Holds status information for deferred errors.	See BKDG/PPR
C000_2009h + i*10h	MCA_DEADDR	Provides the address associated with the deferred error.	See BKDG/PPR
(C000_200Ah: C000_200Dh) + i*10h	MCA_MISC[4:1]	Reports miscellaneous-error information	See BKDG/PPR
(C000_200Eh: C000_200Fh) + i*10h	MCA_SYND[2:1]	Stores information associated with the error in MCA_STATUS or MCA_DESTAT	See BKDG/PPR
***i* - indicates the register bank number**			

A.5 Software-Debug MSRs

Table A-5 lists the MSRs used in support of the software-debug architecture.

Table A-5. Software-Debug MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
01D9h	DebugCtl	Provides debug controls for control-transfer recording and control-transfer single stepping, and external-breakpoint reporting and trace messages.	0000_0000_0000_0000h
01DBh	LastBranchFromIP	During control-transfer recording, this register is loaded with the segment offset of the control-transfer source.	undefined
01DCh	LastBranchToIP	During control-transfer recording, this register is loaded with the segment offset of the control-transfer target.	undefined

Table A-5. Software-Debug MSR Cross-Reference (continued)

MSR Address	MSR Name	Description	Reset Value
01DDh	LastIntFromIP	When an interrupt occurs during control-transfer recording, this register is loaded with LastBranchFromIP before LastBranchFromIP is updated.	undefined
01DEh	LastIntToIP	When an interrupt occurs during control-transfer recording, this register is loaded with LastBranchToIP before LastBranchToIP is updated.	undefined
C000_1027h	DR0_ADDR_MASK	Address mask for DR0 breakpoint [31:0]	0000_0000_0000_0000h
C000_1019h	DR1_ADDR_MASK	Address mask for DR1 breakpoint [31:0]	0000_0000_0000_0000h
C000_101Ah	DR2_ADDR_MASK	Address mask for DR2 breakpoint [31:0]	0000_0000_0000_0000h
C000_101Bh	DR3_ADDR_MASK	Address mask for DR3 breakpoint [31:0]	0000_0000_0000_0000h

A.6 Performance-Monitoring MSRs

Table A-6 lists the MSRs used in support of performance monitoring, including the time-stamp counter.

Table A-6. Performance-Monitoring MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
0010h	TSC	Counts processor-clock cycles. It is incremented once for each processor-clock cycle.	0000_0000_0000_0000h
00E7h	MPERF	Denominator of effective frequency ratio.	0000_0000_0000_0000h
00E8h	APERF	Numerator of effective frequency ratio.	0000_0000_0000_0000h
C000_00E7h	MPerfReadOnly	Read only version of MPERF.	0000_0000_0000_0000h
C000_00E8h	APerfReadOnly	Read only version of APERF.	
C000_00E9h	IRPerfCount	Dedicated instructions retired performance counter.	
C001_0000h	PerfEvtSel0	For the corresponding performance counter, this register specifies the events counted, and controls other aspects of counter operation.	0000_0000_0000_0000h
C001_0001h	PerfEvtSel1		
C001_0002h	PerfEvtSel2		
C001_0003h	PerfEvtSel3		
C001_0004h	PerfCtr0	Used to count specific processor events, or the duration of events, as specified by the corresponding PerfEvtSel*n* register.	undefined
C001_0005h	PerfCtr1		
C001_0006h	PerfCtr2		
C001_0007h	PerfCtr3		

Table A-6. Performance-Monitoring MSR Cross-Reference (continued)

MSR Address	MSR Name	Description	Reset Value
C001_0200h	PerfEvtSel0	These MSR addresses are aliases for the base set of performance event-select registers PerfEvtSel[3:0].	0000_0000_0000_0000h
C001_0202h	PerfEvtSel1		
C001_0204h	PerfEvtSel2		
C001_0206h	PerfEvtSel3		
C001_0208h	PerfEvtSel4	Extended core performance event-select registers. Support for these MSRs is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtCore] = 1.	
C001_020Ah	PerfEvtSel5		
C001_0201h	PerfCtr0	These MSR addresses are aliases for the base set of performance-monitoring counter registers PerfCtr[3:0].	undefined
C001_0203h	PerfCtr1		
C001_0205h	PerfCtr2		
C001_0207h	PerfCtr3		
C001_0209h	PerfCtr4	Extended core performance counter registers. Support for these MSRs is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtCore] = 1.	
C001_020Bh	PerfCtr5		
C001_0230h	L2I_PerfEvtSel0	Specifies the L2 cache events to be counted and controls other aspects of counter operation. Support for these MSRs is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtL2I] = 1.	0000_0000_0000_0000h
C001_0232h	L2I_PerfEvtSel1		
C001_0234h	L2I_PerfEvtSel2		
C001_0236h	L2I_PerfEvtSel3		
C001_0231h	L2I_PerfCtr0	Counts specific L2 cache events as specified by the corresponding L2I_PerfEvtSel*n* Register. Support for these MSRs is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtL2I] = 1.	undefined
C001_0233h	L2I_PerfCtr1		
C001_0235h	L2I_PerfCtr2		
C001_0237h	L2I_PerfCtr3		
C001_0240h	NB_PerfEvtSel0	Specifies northbridge events to be counted and controls other aspects of counter operation. Support for these MSRs is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtNB] = 1.	0000_0000_0000_0000h
C001_0242h	NB_PerfEvtSel1		
C001_0244h	NB_PerfEvtSel2		
C001_0246h	NB_PerfEvtSel3		
C001_0241h	NB_PerfCtr0	Counts specific northbridge events as specified by the corresponding NB_PerfEvtSel*n* Register. Support for these MSRs is indicated by CPUID Fn8000_0001_ECX[PerfCtrExtNB] = 1.	undefined
C001_0243h	NB_PerfCtr1		
C001_0245h	NB_PerfCtr2		
C001_0247h	NB_PerfCtr3		

A.7 Secure Virtual Machine MSRs

Table A-7 lists the MSRs used in support of SVM functions.

Table A-7. Secure Virtual Machine MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
C000_0104h	TSC Ratio	Ratio for scaling TSC, MPERF, and MPerfReadOnly values read by guest.	
C001_0114h	VM_CR	Controls certain global aspects of SVM.	undefined
C001_0115h	IGNNE	Sets the state of the processor-internal IGNNE signal.	
C001_0116h	SMM_CTL	Provides software control over SMM signals.	
C001_0117h	VM_HSAVE_PA	Holds the physical address of a block of memory where VMRUN saves host state, and from which #VMEXIT reloads host state.	
C001_0118h	SVM_KEY	Creates a password-protected mechanism to clear VM_CR.LOCK.	
C001_011Bh	Doorbell Register	Sends a doorbell signal to the specified physical APIC.	
C001_011Eh	VMPAGE_FLUSH	“Secure Encrypted Virtualization” on page 571	
C001_0130h	GHCB	Guest-HV Communication Block address (see section 15.35.7)	
C001_0131h	SEV_STATUS	SEV active features indication (see section 15.35.10)	
C001_0132h	RMP_BASE	Base address of RMP (see 15.36.4)	0000_0000_0000_0000h
C001_0133h	RMP_END	Ending address of RMP (see 15.36.4)	0000_0000_0000_0000h
C001_0134h	GUEST_TSC_FREQ	Guest TSC Frequency (see 15.36.18)	0000_0000_0000_0000h

A.8 System Management Mode MSRs

Table A-8 lists the MSRs used in support of SMM functions.

Table A-8. System Management Mode MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
C001_0056h	SMI_Trigger_IO_Cycle	Specifies an IO cycle that may be generated when a local SMI trigger event occurs. See the appropriate *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* for details.	0000_0000_0000_0000h
C001_0061h	P-State Current Limit		
C001_0062h	P-State Control		
C001_0063h	P-State Status		
C001_0111h	SMBASE	Contains the SMRAM base address.	0000_0000_0003_0000h
C001_0112h	SMM_ADDR	Contains the base address of protected memory for the SMM Handler.	0000_0000_0000_0000h
C001_0113h	SMM_MASK	Contains a mask which determines the size of the protected area for the SMM handler.	0000_0000_0000_0000h
C001_0119h	SMM_KEY_MSR		
C001_011Ah	Local_SMI_Status	Contains status associated with SMI sources local to the CPU core. See the appropriate *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* for details.	0000_0000_0000_0000h

A.9 CPUID Name MSR Cross-Reference

Table A-9 lists the MSRs used to support CPUID namestring.

Table A-9. CPUID Namestring MSR Cross Reference

MSR Address	MSR Name	Description	Reset Value
C001_0030h	Processor_Name_String	See appropriate *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* and *Processor Revision Guide*.	0000_0000_0000_0000h
C001_0031h			
C001_0032h			
C001_0033h			
C001_0034h			
C001_0035h			

A.10 Shadow Stack MSRs

Table A-10 lists the MSRs that support the shadow stack feature. These registers are defined if the shadow stack feature is present as indicated by CPUID Fn0000_0007_x0_ECX[CET_SS] (bit 7) =1.

Table A-10. Shadow Stack MSR Cross Reference

MSR Address	MSR Name	Description	Reset Value
06A0h	U_CET	User-mode shadow stack controls	0000_0000_0000_0000h
06A2h	S_CET	Supervisor-mode shadow stack controls	
06A4h	PL0_SSP	CPL 0 shadow stack pointer	
06A5h	PL1_SSP	CPL 1 shadow stack pointer	
06A6h	PL2_SSP	CPL 2 shadow stack pointer	
06A7h	PL3_SSP	CPL 3 shadow stack pointer	
06A8h	ISST_ADDR	Contains the base address of the Interrupt SSP Table	

A.11 Speculation Control MSRs

Table A-11 lists the MSRs that support speculation control. See “Speculation Control MSRs” on page 67 for further details, including how to determine whether these registers are defined.

Table A-11. Speculation Control MSRs

MSR Address	MSR Name	Description	Reset Value
0048h	SPEC_CTRL	Speculation Control	0000_0000_0000_0000h
0049h	PRED_CMD	Prediction Control	
C000_011Fh	VIRT_SPEC_CTRL	Virtual Speculation Control	

Appendix B Layout of VMCB

The VMCB is divided into two areas—the first one contains various control bits including the intercept vectors and the second one contains saved guest state.

Table B-1 describes the layout of the control area of the VMCB, which starts at offset zero within the VMCB page. The control area is padded to a size of 1024 bytes. All unused bytes must be zero, as they are reserved for future expansion. It is recommended that software zero out any newly allocated VMCB.

Table B-1. VMCB Layout, Control Area

Byte Offset	Bit(s)	Function
000h (vector 0)	15:0	Intercept reads of CR0–15, respectively.
	31:16	Intercept writes of CR0–15, respectively.
004h (vector 1)	15:0	Intercept reads of DR0–15, respectively.
	31:16	Intercept writes of DR0–15, respectively.
008h (vector 2)	31:0	Intercept exception vectors 0–31, respectively.
00Ch (vector 3)	0	Intercept INTR (physical maskable interrupt).
	1	Intercept NMI.
	2	Intercept SMI.
	3	Intercept INIT.
	4	Intercept VINTR (virtual maskable interrupt).
	5	Intercept CR0 writes that change bits other than CR0.TS or CR0.MP.
	6	Intercept reads of IDTR.
	7	Intercept reads of GDTR.
	8	Intercept reads of LDTR.
	9	Intercept reads of TR.
	10	Intercept writes of IDTR.
	11	Intercept writes of GDTR.
	12	Intercept writes of LDTR.
	13	Intercept writes of TR.
	14	Intercept RDTSC instruction.
	15	Intercept RDPMC instruction.

Table B-1. VMCB Layout, Control Area (continued)

Byte Offset	Bit(s)	Function
00Ch (continued)	16	Intercept PUSHF instruction.
	17	Intercept POPF instruction.
	18	Intercept CPUID instruction.
	19	Intercept RSM instruction.
	20	Intercept IRET instruction.
	21	Intercept INT*n* instruction.
	22	Intercept INVD instruction.
	23	Intercept PAUSE instruction.
	24	Intercept HLT instruction.
	25	Intercept INVLPG instruction.
	26	Intercept INVLPGA instruction.
	27	IOIO_PROT—Intercept IN/OUT accesses to selected ports.
	28	MSR_PROT—intercept RDMSR or WRMSR accesses to selected MSRs.
	29	Intercept task switches.
	30	FERR_FREEZE: intercept processor “freezing” during legacy FERR handling.
	31	Intercept shutdown events.

Table B-1. VMCB Layout, Control Area (continued)

Byte Offset	Bit(s)	Function
010h (vector 4)	0	Intercept VMRUN instruction.
	1	Intercept VMMCALL instruction.
	2	Intercept VMLOAD instruction.
	3	Intercept VMSAVE instruction.
	4	Intercept STGI instruction.
	5	Intercept CLGI instruction.
	6	Intercept SKINIT instruction.
	7	Intercept RDTSCP instruction.
	8	Intercept ICEBP instruction.
	9	Intercept WBINVD and WBNOINVD instructions.
	10	Intercept MONITOR/MONITORX instruction.
	11	Intercept MWAIT/MWAITX instruction unconditionally.
	12	Intercept MWAIT/MWAITX instruction if monitor hardware is armed.
	13	Intercept XSETBV instruction.
	14	Intercept RDPRU instruction.
	15	Intercept writes of EFER (occurs after guest instruction finishes).
	31:16	Intercept writes of CR0-15 (occurs after guest instruction finishes).
014h (vector 5)	0	Intercept all INVLPGB instructions.
	1	Intercept only illegally specified INVLPGB instructions.
	2	Intercept INVPCID instruction.
	3	Intercept MCOMMIT instruction.
	4	Intercept TLBSYNC instruction. Presence of this bit is indicated by CPUID Fn8000_000A, EDX[24] = 1.
	31:5	RESERVED, SBZ
018h–03Bh	RESERVED, SBZ	
03Ch	15:0	PAUSE Filter Threshold.
03Eh	15:0	PAUSE Filter Count.
040h	63:0	IOPM_BASE_PA—Physical base address of IOPM (bits 11:0 are ignored.)
048h	63:0	MSRPM_BASE_PA—Physical base address of MSRPM (bits 11:0 are ignored.)
050h	63:0	TSC_OFFSET—To be added in RDTSC and RDTSCP.

Table B-1. VMCB Layout, Control Area (continued)

Byte Offset	Bit(s)	Function
058h	31:0	Guest ASID
	39:32	TLB_CONTROL 00h—Do nothing. 01h—Flush entire TLB (all entries, all ASIDs) on VMRUN. Should only be used by legacy hypervisors. 03h—Flush this guest's TLB entries. 07h—Flush this guest's non-global TLB entries. ***NOTE:*** *All other encodings are reserved.*
	63:40	RESERVED, SBZ
060h	7:0	V_TPR—The virtual TPR for the guest. Bits 3:0 are used for a 4-bit virtual TPR value; bits 7:4 are SBZ. *NOTE: This value is written back to the VMCB at #VMEXIT.*
	8	V_IRQ—If nonzero, virtual INTR is pending. ***NOTE:*** *This value is written back to the VMCB at #VMEXIT. This field is ignored on VMRUN when AVIC is enabled.*
	9	VGIF value (0 – Virtual interrupts are masked, 1 – Virtual Interrupts are unmasked)
	15:10	RESERVED, SBZ
	19:16	V_INTR_PRIO—Priority for virtual interrupt ***NOTE:*** *This field is ignored on VMRUN when AVIC is enabled.*
	20	V_IGN_TPR—If nonzero, the current virtual interrupt ignores the (virtual) TPR. ***NOTE:*** *This field is ignored on VMRUN when AVIC is enabled.*
	23:21	RESERVED, SBZ
	24	V_INTR_MASKING—Virtualize masking of INTR interrupts ("Virtualizing APIC.TPR" on page 518).
	25	AMD Virtual GIF enabled for this guest (0 - Disabled, 1 - Enabled)
	30:26	Reserved, SBZ
	31	AVIC Enable
	39:32	V_INTR_VECTOR—Vector to use for this interrupt. *NOTE: This field is ignored on VMRUN when AVIC is enabled.*
	63:40	RESERVED, SBZ
068h	0	INTERRUPT_SHADOW - Guest is in an interrupt shadow
	1	GUEST_INTERRUPT_MASK - Value of the RFLAGS.IF bit for the guest. ***Note:*** *This value is written back to the VMCB on #VMEXIT. It is not used during VMRUN*
	63:2	RESERVED, SBZ
070h	63:0	EXITCODE
078h	63:0	EXITINFO1

Table B-1. VMCB Layout, Control Area (continued)

Byte Offset	Bit(s)	Function
080h	63:0	EXITINFO2
088h	63:0	EXITINTINFO
090h	0	NP_ENABLE—Enable nested paging.
	1	Enable Secure Encrypted Virtualization
	2	Enable Encrypted State for Secure Encrypted Virtualization
	3	Guest Mode Execute Trap
	4	SSSCheckEn - Enable supervisor shadow stack restrictions in nested page tables. Support for this feature is indicated by CPUID Fn8000_000A_EDX[19] (SSSCheck).
	5	Virtual Transparent Encryption
	6	RESERVED, SBZ
	7	Enable INVLPGB/TLBSYNC. 0 - INVLPGB and TLBSYNC will result in #UD. 1 - INVLPGB and TLBSYNC can be executed in guest. Presence of this bit is indicated by CPUID bit 8000_000A, EDX[24] = 1. When in SEV-ES guest or this bit is not present, INVLPGB/TLBSYNC is always enabled in guest if supported by processor.
	63:8	RESERVED, SBZ
098h	63:52	RESERVED, SBZ
	51:0	AVIC APIC_BAR
0A0h	63:0	Guest physical address of GHCB
0A8h	63:0	EVENTINJ—Event injection (“Event Injection” on page 516 for details.)
0B0h	63:0	N_CR3—Nested page table CR3 to use for nested paging
0B8h	0	LBR_VIRTUALIZATION_ENABLE 0—Do nothing. 1—Enable LBR virtualization hardware acceleration.
	1	Virtualized VMSAVE/VMLOAD (0 –Disabled, 1- Enabled)
	63:2	RESERVED, SBZ
0C0h	31:0	VMCB Clean Bits. See “Layout of VMCB Clean Field” on page 512.
	63:32	RESERVED, SBZ
0C8h	63:0	nRIP—Next sequential instruction pointer
0D0h	7:0	Number of bytes fetched
	127:8	Guest instruction bytes
0E0h	63:52	RESERVED, SBZ
	51:0	AVIC APIC_BACKING_PAGE Pointer

Table B-1. VMCB Layout, Control Area (continued)

Byte Offset	Bit(s)	Function
0E8h–0EFh	Reserved, SBZ	
0F0h	63:52	RESERVED, SBZ
	51:12	AVIC LOGICAL_TABLE Pointer
	11:0	Reserved, SBZ
0F8h	63:52	RESERVED, SBZ
	51:12	AVIC PHYSICAL_TABLE Pointer[51:12]
	11:8	RESERVED, SBZ
	7:0	AVIC_PHYSICAL_MAX_INDEX
100h – 107h	RESERVED, SBZ	
108h	63:52	RESERVED, SBZ
	51:12	VMSA Pointer[51:12]
	11:0	RESERVED, SBZ
All other fields up to 3DFh	RESERVED, SBZ	
3E0h – 3FFh	Reserved for Host usage	

When SEV-ES is not enabled, the state-save area within the VMCB starts at offset 400h into the VMCB page; Table B-2 describes the fields within the state-save area; note that the table lists offsets *relative to the state-save area* (not the VMCB as a whole).

Table B-2. VMCB Layout, State Save Area

Offset	Size	Contents		Notes
000h	word	ES	selector	
002h	word		attrib	
004h	dword		limit	
008h	qword		base	Only lower 32 bits are implemented.
010h	word	CS	selector	
012h	word		attrib	
014h	dword		limit	
018h	qword		base	Only lower 32 bits are implemented.
020h	word	SS	selector	
022h	word		attrib	
024h	dword		limit	
028h	qword		base	Only lower 32 bits are implemented.
030h	word	DS	selector	
032h	word		attrib	
034h	dword		limit	
038h	qword		base	Only lower 32 bits are implemented.

Table B-2. VMCB Layout, State Save Area (continued)

Offset	Size	Contents		Notes
040h	word	FS	selector	
042h	word		attrib	
044h	dword		limit	
048h	qword		base	
050h	word	GS	selector	
052h	word		attrib	
054h	dword		limit	
058h	qword		base	
060h	word	GDTR	selector	reserved
062h	word		attrib	reserved
064h	dword		limit	Only lower 16 bits are implemented.
068h	qword		base	
070h	word	LDTR	selector	
072h	word		attrib	
074h	dword		limit	
078h	qword		base	
080h	word	IDTR	selector	reserved
082h	word		attrib	reserved
084h	dword		limit	Only lower 16 bits are implemented.
088h	qword		base	
090h	word	TR	selector	
092h	word		attrib	
094h	dword		limit	
098h	qword		base	
0A0h–0CAh		RESERVED		
0CBh	byte	CPL		If the guest is real-mode then the CPL is forced to 0; if the guest is virtual-mode then the CPL is forced to 3.
0CCh	dword	RESERVED		
0D0h	qword	EFER		
0D8h–147h		RESERVED		
148h	qword	CR4		
150h	qword	CR3		
158h	qword	CR0		
160h	qword	DR7		
168h	qword	DR6		
170h	qword	RFLAGS		

Table B-2. VMCB Layout, State Save Area (continued)

Offset	Size	Contents	Notes
178h	qword	RIP	
180h–1D7h		RESERVED	
1D8h	qword	RSP	
1E0h	qword	S_CET	
1E8h	qword	SSP	
1F0h	qword	ISST_ADDR	
1F8h	qword	RAX	
200h	qword	STAR	
208h	qword	LSTAR	
210h	qword	CSTAR	
218h	qword	SFMASK	
220h	qword	KernelGsBase	
228h	qword	SYSENTER_CS	
230h	qword	SYSENTER_ESP	
238h	qword	SYSENTER_EIP	
240h	qword	CR2	
248h–267h		RESERVED	
268h	qword	G_PAT	Guest PAT—only used if nested paging enabled.
270h	qword	DBGCTL	Guest DebugCtl MSR—only used if hardware acceleration of LBR virtualization is supported and enabled by setting the LBR_VIRTUALIZATION_ENABLE bit of the VMCB control area.
278h	qword	BR_FROM	Guest LastBranchFromIP MSR—only used if hardware acceleration of LBR virtualization is supported and enabled.
280h	qword	BR_TO	Guest LastBranchToIP MSR—only used if hardware acceleration of LBR virtualization is supported and enabled.
288h	qword	LASTEXCPFROM	Guest LastIntFromIP MSR—Only used if hardware acceleration of LBR virtualization is supported and enabled.
298h–2DFh	72 bytes	RESERVED	
2E0h	qword	SPEC_CTRL	
2E8h to end of VMCB		RESERVED	

When SEV-ES is enabled (Section 15.35 “Encrypted State (SEV-ES)” on page 578), the VMSA structure starts at offset 0h in the page indicated by the VMSA Pointer. The format of the VM save state for SEV-ES guests is described in the table below.

All state is categorized into 3 swap types based on how it is handled by hardware during a world switch:

Table B-3. Swap Types

Swap Type	Behavior in VMRUN	Behavior in AE VMEXIT
A	Host state saved to host save area Guest state loaded from VMSA	Guest state saved to VMSA Host state loaded from host save area
B	Guest state loaded from VMSA (Host state **not** saved to host save area)	Guest state saved to VMSA Host state loaded from host save area
C	Guest state loaded from VMSA (Host state **not** saved to host save area)	Guest state saved to VMSA Host state initialized to default (reset) values

The format of the host save area is identical to the guest save area described in the table below, except that it begins at offset 400h in the host save page (e.g., the host TR value is stored at offset 490h relative to the start of the host save page).

Table B-4. VMSA Layout, State Save Area for SEV-ES

Offset	Size	Content	Swap Type	Notes
000h	16 bytes	ES	A	
010h	16 bytes	CS	A	
020h	16 bytes	SS	A	
030h	16 bytes	DS	A	
040h	16 bytes	FS	B	
050h	16 bytes	GS	B	
060h	16 bytes	GDTR	A	
070h	16 bytes	LDTR	B	
080h	16 bytes	IDTR	A	
090h	16 bytes	TR	B	
0A0h	qword	PL0_SSP	B	
0A8h	qword	PL1_SSP	B	
0B0h	qword	PL2_SSP	B	
0B8h	qword	PL3_SSP	B	
0C0h	qword	U_CET	B	
0C8h	dword	Reserved	–	
0CAh	byte	VMPL	–	Swapped for guest. Not used in host mode
0CBh	byte	CPL	A	
0CCh	dword	Reserved	–	

Table B-4. VMSA Layout, State Save Area for SEV-ES (continued)

Offset	Size	Content	Swap Type	Notes
0D0h	qword	EFER	A	
0D8h-13Fh	104 bytes	Reserved	–	
140h	qword	XSS	B	
148h	qword	CR4	A	
150h	qword	CR3	A	
158h	qword	CR0	A	
160h	qword	DR7	C	
168h	qword	DR6	C	
170h	qword	RFLAGS	A	
178h	qword	RIP	A	
180h	qword	DR0	B	
188h	qword	DR1	B	
190h	qword	DR2	B	
198h	qword	DR3	B	
1A0h	qword	DR0_ADDR_MASK	B	
1A8h	qword	DR1_ADDR_MASK	B	
1B0h	qword	DR2_ADDR_MASK	B	
1B8h	qword	DR3_ADDR_MASK	B	
1C0h-1D7h	24 bytes	Reserved	–	
1D8h	qword	RSP	A	
1E0h	qword	S_CET	A	
1E8h	qword	SSP	A	
1F0h	qword	ISST_ADDR	A	
1F8h	qword	RAX	A	
200h	qword	STAR	B	
208h	qword	LSTAR	B	
210h	qword	CSTAR	B	
218h	qword	SFMASK	B	
220h	qword	KernelGsBase	B	
228h	qword	SYSENTER_CS	B	
230h	qword	SYSENTER_ESP	B	
238h	qword	SYSENTER_EIP	B	
240h	qword	CR2	C	
248h-267h	32 bytes	Reserved	–	
268h	qword	G_PAT	–	Swapped for guest, not used in host mode

Table B-4. VMSA Layout, State Save Area for SEV-ES (continued)

Offset	Size	Content	Swap Type	Notes
270h	qword	DBGCTL	A	
278h	qword	BR_FROM	A	
280h	qword	BR_TO	A	
288h	qword	LASTEXCPFROM	A	
290h	qword	LASTEXCPTO	A	
298h-2DFh	72 bytes	Reserved	–	
2E0h	qword	Reserved	–	
2E8h	dword	PKRU	B	
2ECh	dword	Reserved	–	
2F0h	qword	GUEST_TSC_SCALE	–	
2F8h	qword	GUEST_TSC_OFFSET	–	
300h	qword	REG_PROT_NONCE	–	
308h	qword	RCX	B	
310h	qword	RDX	B	
318h	qword	RBX	B	
320h	qword	Reserved	–	
328h	qword	RBP	B	
330h	qword	RSI	B	
338h	qword	RDI	B	
340h	qword	R8	B	
348h	qword	R9	B	
350h	qword	R10	B	
358h	qword	R11	B	
360h	qword	R12	B	
368h	qword	R13	B	
370h	qword	R14	B	
378h	qword	R15	B	
380h	16 bytes	Reserved	–	
390h	qword	GUEST_EXITINFO1	–	EXITINFO1 for AE exits.
398h	qword	GUEST_EXITINFO2	–	EXITINFO2 for AE exits.
3A0h	qword	GUEST_EXITINTINFO	–	EXITINTINFO for AE exits.
3A8h	qword	GUEST_NRIP	–	Next sequential instruction pointer for AE exits.

Table B-4. VMSA Layout, State Save Area for SEV-ES (continued)

Offset	Size	Content	Swap Type	Notes
3B0h	qword	SEV_FEATURES	–	Guest-controlled SEV feature selection. • Bit 0: SNPActive • Bit 1: vTOM • Bit 2: ReflectVC • Bit 3: RestrictedInjection • Bit 4: AlternateInjection • Bit 5: DebugSwap • Bit 6: PreventHostIBS • Bit 7: BTBIsolation • Bit 8: Reserved, SBZ • Bit 9: SecureTSC • Bit 13:10: Reserved, SBZ • Bit 14: VmsaRegProt • Bits 63:15: Reserved, SBZ
3B8h	qword	VINTR_CTRL	–	Guest-controlled injection control. • Bits 7:0: V_TPR • Bit 8: V_IRQ • Bit 9: VGIF • Bit 10: INT_SHADOW • Bits 15:11: Reserved, SBZ • Bits 19:16: V_INTR_PRIO • Bit 20: V_IGN_TPR • Bit 31:21: Reserved, SBZ • Bit 39:32: V_INTR_VECTOR • Bit 62:40: Reserved, SBZ • Bit 63: BUSY
3C0h	qword	GUEST_EXITCODE	–	EXITCODE for AE exits.
3C8h	qword	VIRTUAL_TOM	–	Swapped for guest, not used in host mode. Only bits 51:21 are observed.
3D0h	qword	TLB_ID	–	
3D8h	qword	PCPU_ID	–	
3E0h	qword	EVENTINJ	–	Same as the EVENTINJ field in the VMCB (Table B-1) at offset 0A8h.
3E8h	qword	XCR0	B	
3F0h-3FFh	16 bytes	Reserved	–	
400h	qword	X87_DP	C	FP x87 data pointer
408h	dword	MXCSR	C	FP MXCSR

Table B-4. VMSA Layout, State Save Area for SEV-ES (continued)

Offset	Size	Content	Swap Type	Notes
40Ch	word	X87_FTW	C	FP x87 tag word
40Eh	word	X87_FSW	C	FP x87 status word
410h	word	X87_FCW	C	FP control word
412h	word	X87_FOP	C	FP x87 opcode
414h	word	X87_DS	C	FP x87 DS
416h	word	X87_CS	C	FP x87 CS
418h	qword	X87_RIP	C	FP x87 RIP
420h-46Fh	80 bytes	FPREG_X87	C	X87 register state (stack order)
470h-56Fh	256 bytes	FPREG_XMM	C	XMM register state
570h-66Fh	256 bytes	FPREG_YMM	C	YMM_HI register state

Appendix C SVM Intercept Exit Codes

When the VMRUN instruction exits (back to the host), an exit/reason code is stored in the EXITCODE field in the VMCB. The exit codes are defined in Table C-1. Intercept exit codes 0h–8Dh equal the bit position of the corresponding flag in the VMCB's intercept vector.

Table C-1. SVM Intercept Codes

Code	Name	Cause
0h–Fh	VMEXIT_CR[0–15]_READ	read of CR 0 through 15, respectively
10h–1Fh	VMEXIT_CR[0–15]_WRITE	write of CR 0 through 15, respectively
20h–2Fh	VMEXIT_DR[0–15]_READ	read of DR 0 through 15, respectively
30h–3Fh	VMEXIT_DR[0–15]_WRITE	write of DR 0 through 15, respectively
40h–5Fh	VMEXIT_EXCP[0–31]	exception vector 0–31, respectively
60h	VMEXIT_INTR	physical INTR (maskable interrupt)
61h	VMEXIT_NMI	physical NMI
62h	VMEXIT_SMI	physical SMI (the EXITINFO1 field provides more information)
63h	VMEXIT_INIT	physical INIT
64h	VMEXIT_VINTR	virtual INTR
65h	VMEXIT_CR0_SEL_WRITE	write of CR0 that changed any bits other than CR0.TS or CR0.MP
66h	VMEXIT_IDTR_READ	read of IDTR
67h	VMEXIT_GDTR_READ	read of GDTR
68h	VMEXIT_LDTR_READ	read of LDTR
69h	VMEXIT_TR_READ	read of TR
6Ah	VMEXIT_IDTR_WRITE	write of IDTR
6Bh	VMEXIT_GDTR_WRITE	write of GDTR
6Ch	VMEXIT_LDTR_WRITE	write of LDTR
6Dh	VMEXIT_TR_WRITE	write of TR
6Eh	VMEXIT_RDTSC	RDTSC instruction
6Fh	VMEXIT_RDPMC	RDPMC instruction
70h	VMEXIT_PUSHF	PUSHF instruction
71h	VMEXIT_POPF	POPF instruction
72h	VMEXIT_CPUID	CPUID instruction
73h	VMEXIT_RSM	RSM instruction
74h	VMEXIT_IRET	IRET instruction
75h	VMEXIT_SWINT	software interrupt (INT*n* instructions)
76h	VMEXIT_INVD	INVD instruction
77h	VMEXIT_PAUSE	PAUSE instruction
78h	VMEXIT_HLT	HLT instruction

Table C-1. SVM Intercept Codes (continued)

Code	Name	Cause
79h	VMEXIT_INVLPG	INVLPG instructions
7Ah	VMEXIT_INVLPGA	INVLPGA instruction
7Bh	VMEXIT_IOIO	IN or OUT accessing protected port (the EXITINFO1 field provides more information)
7Ch	VMEXIT_MSR	RDMSR or WRMSR access to protected MSR
7Dh	VMEXIT_TASK_SWITCH	task switch
7Eh	VMEXIT_FERR_FREEZE	FP legacy handling enabled, and processor is frozen in an x87/mmx instruction waiting for an interrupt
7Fh	VMEXIT_SHUTDOWN	Shutdown
80h	VMEXIT_VMRUN	VMRUN instruction
81h	VMEXIT_VMMCALL	VMMCALL instruction
82h	VMEXIT_VMLOAD	VMLOAD instruction
83h	VMEXIT_VMSAVE	VMSAVE instruction
84h	VMEXIT_STGI	STGI instruction
85h	VMEXIT_CLGI	CLGI instruction
86h	VMEXIT_SKINIT	SKINIT instruction
87h	VMEXIT_RDTSCP	RDTSCP instruction
88h	VMEXIT_ICEBP	ICEBP instruction
89h	VMEXIT_WBINVD	WBINVD or WBNOINVD instruction
8Ah	VMEXIT_MONITOR	MONITOR or MONITORX instruction
8Bh	VMEXIT_MWAIT	MWAIT or MWAITX instruction
8Ch	VMEXIT_MWAIT_CONDITIONAL	MWAIT or MWAITX instruction, if monitor hardware is armed.
8Eh	VMEXIT_RDPRU	RDPRU instruction
8Dh	VMEXIT_XSETBV	XSETBV instruction
8Fh	VMEXIT_EFER_WRITE_TRAP	Write of EFER MSR (occurs after guest instruction finishes)
90h-9Fh	VMEXIT_CR[0-15]_WRITE_TRAP	Write of CR0-15, respectively (occurs after guest instruction finishes)
A0h	VMEXIT_INVLPGB	INVLPGB instruction
A1h	VMEXIT_INVLPGB_ILLEGAL	Illegal INVLPGB instruction
A2h	VMEXIT_INVPCID	INVPCID instruction
A3h	VMEXIT_MCOMMIT	MCOMMIT instruction
A4h	VMEXIT_TLBSYNC	TLBSYNC instruction
400h	VMEXIT_NPF	Nested paging: host-level page fault occurred (EXITINFO1 contains fault error code; EXITINFO2 contains the guest physical address causing the fault.)
401h	AVIC_INCOMPLETE_IPI	AVIC—Virtual IPI delivery not completed. See "AVIC IPI Delivery Not Completed" for EXITINFO1–2 definitions.

Table C-1. SVM Intercept Codes (continued)

Code	Name	Cause
402h	AVIC_NOACCEL	AVIC—Attemped access by guest to vAPIC register not handled by AVIC hardware. See "AVIC Access to un-accelerated vAPIC register" for EXITINFO1–2 definitions.
403h	VMEXIT_VMGEXIT	VMGEXIT instruction
F000_000h	Unused	Reserved for Host
–1	VMEXIT_INVALID	Invalid guest state in VMCB
–2	VMEXIT_BUSY	BUSY bit was set in the encrypted VMSA (see "Interrupt Injection Restrictions")

Appendix D SMM Containerization

To minimally participate in SMM activity, the VMM can implement simple containerization. This appendix provides example pseudocode to perform this simple containerization. VMMs that do not trust SMM code should implement secure containerization, which requires further extension of the code provided here.

D.1 SMM Containerization Pseudocode

This code emulates transitions to and from SMM:

- The process of entering SMM mode as a result of a system management interrupt (SMI)
- The RSM instruction, which returns the processor from SMM.

A hypervisor that containerizes SMM must set the SMM intercept bit in all guest VMCBs. When the hypervisor encounters a #VMEXIT(SMI), it should then emulate SMM entry and execute the SMM handler by means of VMRUN with the RSM intercept bit set. When the RSM instruction is intercepted, the hypervisor should emulate the RSM instruction and then resume normal execution.

In this code, the hypervisor sets up the `smm_vmcb` from scratch and assigns it the supplied address space identifier (ASID).

This example code sets up a container VMCB for the SMM handler and copies appropriate state information into the SMM save area. After calling `emulate_smm()`, the hypervisor should repeatedly VMRUN the SMM handler VMCB until the hypervisor encounters a #VMEXIT(RSM). Finally, the hypervisor should call `emulate_rsm()`.

```
//emulate_smm( ):
// Inputs:
//   smm_vmcb:   the _virtual address_ of a VMCB that will be configured
//               as an SMM container
//   asid:       the asid to use for the SMM handler; the hypervisor should
//               ensure that no TLB entries for this ASID are present in the TLB
//   smm_regs:   an array of 64-bit values that will be filled with the
//               GPRs (except RSP and RAX) for the SMM handler
//   guest_vmcb:  the _virtual address_ of the VMCB of the guest
//                that was running when the intercepted SMI occurred
//   guest_regs: an array of 64-bit values that contains the GPRs (except RSP
//               and RAX) for the guest that was running when the intercepted
//               SMI occurred

void
emulate_smm(VMCB *smm_vmcb, uint32 asid, uint64 smm_regs[16],
            VMCB *guest_vmcb, uint64 guest_regs[16])
{
   setup_smm_container(*smm_vmcb, asid, smm_regs, *guest_vmcb, guest_regs)
```

```
   //Enter SMM mode:
   wrmsr(SMM_CTL_MSR, ENTER+DISMISS+SMI_CYCLE)
   setup_smm_save_state(*guest_vmcb, guest_regs)


   do { VMRUN(smm_vmcb) } until we see #VMEXIT(RSM).
       Shadow EFER reads and writes to protect the SVME bit.


   //Emulate RSM:
   copy_smm_save_to_guest_vmcb(guest_vmcb, guest_regs)
   //Leave SMM mode:
   wrmsr(SMM_CTL_MSR, EXIT+RSM_CYCLE)
}

void
setup_smm_container(VMCB &smm_vmcb, uint32 asid, uint64 smm_regs[16],
                    VMCB &g_vmcb, uint64 guest_regs[16])
{
   clear smm_vmcb to all zeros
   set intercepts in smm_vmcb:
      RSM
      VMRUN
      MSR
   smm_vmcb.msrpm = (physical address of msr protection map with
                     efer read and efer write set)
   // Note that the hypervisor should shadow the SVME bit of EFER and
   // return EFER.SVME=0 on reads of EFER.
   //
   // Note also that the IOPM (unused in this example code) and MSRPM for the SMM
   // container can be statically set up and reused on subsequent SMM entries,
   // and can be shared between multiple cores' SMM container VMCBs. Each core
   // must have a separate VMCB for the SMM container, but those cores’ VMCBs may
   // be statically or dynamically allocated.

   smm_vmcb.asid = asid

   smmbase = rdmsr(smmbase_msr) // Note: smmbase is a 32 bit value

   Set up the smm handler's segment information: {Selector, Attrib, Limit, Base}

   smm_vmcb.CS = {(smmbase & 0x00ffff00) >> 4, 0x089B, 0xffff_ffff, smmbase}
   smm_vmcb.{ES, SS, DS, FS, GS} = {0x0000, 0x0893, 0xffff_ffff, 0x0000_0000}
   smm_vmcb.GDTR = {unused, unused, g_vmcb.gdtr_limit, g_vmcb.gdtr_base}
   smm_vmcb.LDTR = (copy all from g_vmcb.LDTR)
   smm_vmcb.IDTR = {unused, unused, g_vmcb.idtr_limit, g_vmcb.idtr_base}
   smm_vmcb.TR = (copy all from g_vmcb.TR)

   smm_vmcb.CPL = 0
   smm_vmcb.EFER = 0x1000 (SVME = 1)
```

```
   smm_vmcb.CR4 = 0
   smm_vmcb.DR7 = 0x0000_0400
   smm_vmcb.RFLAGS = 0x0000_0002
   smm_vmcb.RIP = 0x0000_8000


   Copy the following values from g_vmcb to smm_vmcb
   CR3
   DR6
   RSP
   RAX
   STAR
   LSTAR
   CSTAR
   SFMASK
   KERNELGSBASE
   SYSENTER_CS
   SYSENTER_ESP
   SYSENTER_EIP
   CR2
   CR0: clear bits 0, 2, 3, 31

   copy 14 guest GPRs from guest_regs (all except RAX, RSP) to smm_regs
}


void
setup_smm_save_state(struct VMCB &g_vmcb, uint64 guest_regs[16])
{
   smmbase = rdmsr(smmbase_msr) // Note: smmbase is a 32 bit value
   smmsave_physical_addr = smmbase + 0xfe00
   // smmsave is the physical address of the SMM save area;
   // the hypervisor will need to map this into its virtual memory space.
   smmsave = virtual_to_physical_map(smmsave_physical_addr)

   Copy the following values from g_vmcb to smmsave:
   all defined portions of ES, CS, SS, DS, FS, GS, GDTR, LDTR, IDTR, TR
        (all bytes of each 16-byte segment save area)
   CPL
   EFER
   CR4
   CR3
   CR0
   DR7
   DR6
   RFLAGS
   RIP
   RSP
   RAX

   copy 14 guest GPRs (other than RAX and RSP) from guest_regs
```

```
                                                to GPR entries in smmsave

   iorestart_dword[31:0] = g_vmcb.exitinfo1[63:32]
   if ((iorestart_dword & IO_RESTART_VALID) != 0)
   {
      Copy iorestart_dword to smmsave.iorestart_dword,
            masking out address size bits
      Copy g_vmcb.exitinfo2 to smmsave.iorestart_rip

      uint64 *guest_indexreg // Point to the index register in the guest context
                             // that is changed by the string instruction...
      uint64 *smm_indexreg   // ...similarly, for the smm save area
      if (iorestart_dword & IO_RESTART_IN != 0) {
         guest_indexreg = &guest_regs[RDI] // type=IN, indexreg=RDI
         smm_indexreg = &smmsave.iorestart_rdi
         smmsave.iorestart_rsi = guest_regs[RSI]
      } else {
         guest_indexreg = &guest_regs[RSI] // type=OUT, indexreg=RSI
         smm_indexreg = &smmsave.iorestart_rsi
         smmsave.iorestart_rdi = guest_regs[RDI]
      }

      // Reconstruct the IORestart values
      if (iorstart_dword & IO_RESTART_STR != 0)
      {
          uint64 mask
          uint64 ecxfix

          operand_size = (iorestart_dword >> 4) & 0x7)

          address_size = (iorestart_dword >> 7) & 0x7)
          if (address_size == 0) // Some SVM implementations do not provide
                                 // these bits; we must decode on those CPUs
             address_size = decode_io_size(guest_vmcb)

          mask = (1<<address_size) - 1
          if (g->RFLAGS D-bit is set)
             operand_size = -operand_size

          if (iorestart_dword & IO_RESTART_RIP != 0)
             ecxfix = 1
          else ecxfix = 0

          *smm_indexreg = *guest_indexreg & ~mask  |  (*guest_indexreg -
                                                          operand_size) & mask
          smmsave.iorestart_rcx = mask & (guest_regs[RCX] + ecxfix)
      } else { // not string
          *smm_indexreg = *guest_indexreg
          smmsave.iorestart_rcx = guest_regs[RCX]
      }
   } else { // iorestart isn't valid: Put the same values into the restart values.
```

```
      smmsave.iorestart_dword = 0
      smmsave.iorestart_rip = g.rip
      smmsave.iorestart_rcx = guest_regs[RCX]
      smmsave.iorestart_rsi = guest_regs[RSI]
      smmsave.iorestart_rdi= guest_regs[RDI]
   }

   smmsave.iorestart = 0
   smmsave.hltrestart = 0
   smmsave.nmimask = 0
   smmsave.smm_revision = 0x30064
   smmsave.smm_base = smmbase
}

void
copy_smm_save_to_guest_vmcb(struct VMCB &g_vmcb, uint64 guest_regs[16])
{
   smmbase = rdmsr(smmbase_msr) // Note: smmbase is a 32 bit value
   smmsave_physical_addr = smmbase + 0xfe00
   // smmsave is the physical address of the SMM save area;
   // the hypervisor will need to map this into its virtual memory space.
   smmsave = virtual_to_physical_map(smmsave_physical_addr)

   Copy the following values from smmsave to g_vmcb
   all defined portions of ES, CS, SS, DS, FS, GS, GDTR, LDTR, IDTR, TR
   CPL
   EFER
   CR4
   CR3
   CR0
   DR7
   DR6
   RFLAGS
   RSP
   RAX

   Copy the other 14 GPRs from smmsave into guest_regs.

   If smmsave.iorestart is set, copy RDI,
   RSI, RCX from the smmsave.iorestart_{RDI, RSI, RCX} fields
                instead of the regular {RDI, RSI, RCX} fields.

   if (smmsave.iorestart is zero and smmsave.iorestart_dword is valid)
   {
      modify g_vmcb.DR6:
         clear g_vmcb.DR6[3:0] and copy BRP bits from
                smmsave.iorestart_dword[15:12] into g_vmcb.DR6[3:0]
         // this preserves AMD's behavior that dr6[3:0] is not sticky,
         // but the other bits are sticky
         g_vmcb.DR6.BS |= smmsave.iorestart_dword.TF
      if any bit of smmsave.iorestart_dword.{BRP[3:0], TF} is nonzero,
```

```
         we have a pending #DB exception,
         so set up a #DB event injection for the guest.
   }


   if (smmsave.iorestart is set) {
      set g_vmcb.RIP = smmsave.iorestart_rip
   } else if (smmsave.hltrestart is set) {
      // (In the event that the guest is allowed to execute HLT and
      // the SMM code wants to use the auto-halt restart function,
      // we need to re-execute the HLT instruction in the guest context.
      // Even if the HLT has prefixes (all of which would be ignored),
      // we know that RIP-1 is the F4 opcode itself.)

      Subtract 1 from the guest RIP under a mask that masks out bits
             above the current default address size:

      mask = (1 << current_address_size) - 1
      g_vmcb.RIP = mask & (g_vmcb.RIP-1)
   } else {
      set g_vmcb.RIP = smmsave.RIP
   }
// Note that it is undefined to have both iorestart and hltrestart set at
// the same time.

// Perform the RSM consistency checks listed in volume 3 of the
// AMD64 Architecture Programmer's manual, except the check that
// disallows CR0.PG = 1 when CR0.PE = 0.  Note that the expected
// value for the SMM revision field is 0x0003_0064.  If any of the
// checks fail, the native RSM instruction would have caused a
// processor shutdown (which commonly results in a reboot
// triggered by the chipset). The hypervisor may wish to destroy
// the guest or cause its own shutdown.
}
```

D.1.1 Converting Simple Containerization into Secure Containerization

To convert this simple containerization example into secure containerization, the hypervisor must limit the SMM handler's access to I/O ports, MSRs, and memory. Based on security policy decisions, the hypervisor should set appropriate bits in the I/O Protection Map and the MSR Protection map and emulate any accesses the SMM handler makes to those protected resources. The hypervisor should run the SMM handler in paged real mode, with a page table that appropriately limits memory accessible to SMM code. Additionally, the hypervisor may wish to conceal some or all of the contents of a guest's general purpose and floating-point registers from the SMM handler.

Appendix E OS-Visible Workarounds

Operating system software may provide a workaround for a hardware erratum. These operating system-visible workarounds are provisional and should be removed or disabled when the erratum is corrected in a subsequent hardware release.

The OS-Visible Workaround (OSVW) architecture provides a means by which operating system software may determining the status of a known erratum for the hardware on which the software is running. Support for the OSVW mechanism is indicated by CPUID Fn8000_0001_ECX[OSVW] = 1.

See Section 3.3, "Processor Feature Identification," on page 71 for information on using the CPUID instruction.

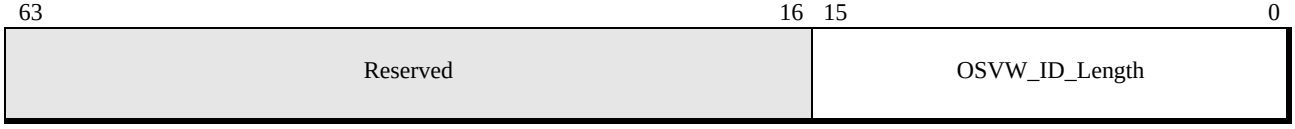
Each hardware erratum is assigned a unique OSVW ID number. OSVW ID numbers start at 0 and are assigned sequentially up to the most recently identified erratum which is assigned the number *m*–1. The OSVW mechanism encodes the status of each erratum for a given hardware system in a bit vector of length *m* accessed through OSVM MSRs 1–N. The state of bit *n* of the vector indicates the status of the erratum with the OSVW ID number *n*. The OSVW ID number for the erratum and the bit position within the erratum status bit vector, once assigned, are global across all AMD processors; the OSVW ID and bit position will not be re-used.

The OSVW MSRs are defined as follows:

- OSVW MSR0 contains the *OSVW_ID_Length* field, used to indicate the total number of valid OSVW ID bits (*m*). The format of this MSR is shown in Figure E-1 below.
- OSVW MSR 1 and following contain the erratum status bit vector of length *m*. Each bit *n* of this vector encodes the status of erratum *n* (OSVW ID = *n*). The format of these MSRs is shown in Figure E-2 on page 704.

The bank of OSVW MSRs is located at address C001_0140h, starting with OSVW MSR0.

The OSVW MSRs should be treated as read-only registers for the OS. The OS should never write into these registers. Hardware allows platform firmware writes to these registers.



Bits	Mnemonic	Description	R/W[1]
63:16	Reserved		
15:0	OSVW_ID_Length	Total length of the status vector OSVW_E in bits.	R/W

Note 1: MSR should be treated as read-only by operating system software.

Figure E-1. OSVW MSR0: OSVW_ID_Length

OSVW_ID_Length—Bits [15:0]. The number of valid bits in the OSVW erratum status vector OSVW_E. If a specific erratum has an OSVW ID that is greater than or equal to the *OSVW_ID_Length*, the erratum is unknown to the latest release. Otherwise, the erratum status bit in the appropriate OSVW MSR can be checked to see if a workaround is required.

The erratum status bit vector (OSVW_E) is accessed through OSVW MSR 1 and following. For MSR *N*, the 64-bit MSR holds erratum status bits (*N*–1)*64+63:(*N*–1)*64. To access the erratum status for OSVW ID number *n* (E[*n*] in the diagram), read MSR *N*, where *N* = *n*/64 + 1, and test bit *i*, where *i* = *n* modulo 64.

Figure E-2 below gives the format of the OSVW MSRs 1–N.

Bit	Mnemonic	Description	R/W[1]
i	OSVW_E[*n*]	OS-visible workaround status bit *n*	R/W

Note 1: MSR should be treated as read-only by operating system software.

Figure E-2. OSVW MSRs 1–N: OSVW Erratum Status Registers

OS-Visible Workaround Erratum Status (OSVW_E[n])—Bits 63:0. Each bit indicates whether platform hardware is affected by OS-visible erratum *n* and whether the OS needs to apply a workaround.

For the status bit:

1 = Hardware contains the erratum; an OS software workaround is required.

0 = Hardware has corrected the erratum; an OS software workaround is unnecessary. If one is installed, it must be disabled.

The location of an OSVW ID status bit within a bank of OSVW MSRs is determined as follows:

- MSR address = OSVW_MSR0 + 1 + floor (OSVW_ID /64)
- Bit offset in MSR = OSVW_ID modulo 64

If a specific erratum has an OSVW_ID that is greater than or equal to the OSVW_ID_LENGTH, hardware does not know about the erratum and the processor model must be used to determine whether the workaround must be applied.

OSVW MSR bits beyond the end of the OSVW_E bit vector are reserved.

E.1 Erratum Process Overview

Following is an overview of the AMD erratum process:

1. When an OS-visible erratum is discovered, AMD assigns a unique OSVW ID to the erratum and publishes to OS vendors the starting range of affected processor models and suggested workarounds.
2. AMD works with platform firmware vendors and OEMs in parallel to develop a firmware update to add the new erratum status bit to the OSVW_E erratum status bit vector for affected silicon revisions to report the new OSVW ID as requiring a workaround. The OSVW_ID_Length field in OSVW MSR0 is incremented by one.
3. OS vendors schedule the workaround into their release schedules and eventually release it.
4. The OS detection logic for the workaround first checks whether the processor OSVW MSRs 1–N record the erratum by comparing the OSVW ID of the erratum with the OSVW_ID_Length field in OSVW MSR0.
5. If the erratum OSVW ID is greater than or equal to the OSVW_ID_Length, the current firmware does not know about this erratum. In this case, the OS software compares the processor model ID with the starting model ID that AMD supplied with the erratum to determine if the workaround should be applied.
6. If the erratum OSVW ID is less than the OSVW_ID_Length, the firmare is aware of the erratum. In this case, the OS uses the state of the associated OSVW_E status bit to conditionally apply the workaround. If the associated status bit is set, the workaround is applied.
7. Once AMD fixes the erratum in a future release, updated firmware ensures that the OSVW_E status bit associated with the erratum is cleared. When OS workaround detection logic runs on the new hardware, it will see that the bit corresponding to the OSVW ID is cleared and not apply the OS workaround for that erratum.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

AMD64 Technology

AMD64 Architecture Programmer's Manual

Volume 3: General-Purpose and System Instructions

Publication No.	Revision	Date
24594	3.33	November 2021

Advanced Micro Devices

Trademarks

Contents

Figures

Tables

Revision History

Date	Revision	Description
November 2021	3.33	Added CET exception case to RET FAR pseudo code. Added missing INCSSP exception information. Clarified INVLPGB description. Added details to RDPMC description. Corrected exception information for PSMASH, RMPADJUST, RMPUPDATE. Added clarification to TLBSYNC. Added CPUID information for various new features. Corrected opcode map note for PREFETCH encodings.
March 2021	3.32	Chapter 1: Updated Instruction Encoding Syntax and An Instruction as Stored in Memory figures. Added content to Summary of Legacy Prefixes section. Chapter 3: Added content Instruction Support Indicated by CPUID Feature Bits table. Added content to LFENCE Updated note 1 in the Legacy Instruction Prefixes table. Chapter 4: Added content to the System Instruction Support Indicated by CPUID Feature Bits table. Added VMGEXIT instruction. Added content to WRMSR instruction. Appendix D: Added content to the Feature Flags for Instruction / Instruction Subset Support table. Appendix E: Updated instructions and added instructions to sections E.3 and E.4: See bold line items.

Date	Revision	Description
October 2020	3.31	Chapter 2: Added to pseudocode Definition section. Table 2-1: Added content. Chapter 3: Added pseudocode updates. Chapter 4: Added pseudocode updates. Added 8 new instructions. Added INVLPGB, TLBSYNC to System Instruction Support Indicated by CPUID Feature Bits table. Updated INVLPGB and TLBSYNC description. Appendix A: Instructions encoding clarifications. Appendix D: Added new instructions to Feature Flags for Instruction / Instruction Subset Support table. Appendix E: Added content to CPUID Fn0000_0007_ECX_x0 Structured Extended Feature Identifiers (ECX=0) table and to Function Dh—Processor Extended State Enumeration section. Added content to CPUID Fn8000_0008_EBX Extended Feature Identifiers, CPUID Fn8000_000A_EDX SVM Feature Identification, and CPUID Fn8000_001F_EAX tables.
April 2020	3.30	Chapter 4: Updated INVLPG, MOV CRn, and RSM sections. Chapter 4: Added INVLPGB, INVPCID, RDPKRU, TLBSYNC, and WRPKRU instructions. Appendix D: Table D-1. Updated table. Appendix E: Updated E.3.6, E.4.7, and E.4.9 sections.
April 2020	3.29	Table 2-1: Added content. Chapter 4: Added PSMASH, PVALIDATE, RMPADJUST, and RMPUPDATE instructions. Appendix A: Table A-6, A-7, and A-8: Updated table. Appendix D: Table D-1: Added content. Removed D.3 section. Appendix E: Material for new features plus clarifications. Appendix F: Table F-1: Added content.
September 2019	3.28	Added MCOMMIT instruction. Corrected CPUID function 8000_001Dh description.
July 2019	3.27	Added CLWB, RDPID, RDPRU, and WBNOINVD instructions. Corrected functional details of BZHI instruction. Corrected SAHF and LAHF #UD fault details. Corrected RSM reserved-bit behavioral details.

Date	Revision	Description
May 2018	3.26	Modified description of CLFLUSH. Added clarification that MOVD is referred to in some forms as MOVQ. Corrected the operands for VMOVNTDQA . Updated L2/L3 Cache and Associativity tables with new encodings over old reserved encodings Updated CPUID with Nested Virtualization and Virtual GIF indication bits.
December 2017	3.25	Updated Appendix E.
November 2017	3.24	Modified Mem16int in Section 2.5.1 Mnemonic Syntax Corrected Opcode for ADCX and ADOX. Clarified the explanation for Load Far Pointer Modified the Description for CLAC and STAC Added clarification to MWAITX. Added clarifying footnote to Table A-6. Added CPUID flags for new SVM features. Added Bit descriptions for CPUID Fn8000_0008_EBX Reserved Modified SAL1 and SAL count in Appendix F, Table F-1.
March 2017	3.23	Added CR0.PE, CR0.PE=1, EFER.LME=0 to Conventions and Definitions in the Preface. Modified Note 4 in Table 1-10. Chapter 3: Added ADCX, ADOX, CLFLUSHOPT, CLZERO, RDSEED, UD0 and UD1. Modified CALL (Far). Moved UD2 and MONITORX, MWAITX, from Chapter 4. Chapter 4: Modified RDTSC and RDTSCP. Added CLAC and STAC. Appendix A: Modified Table A-7, Group 11. Appendix D: Modified Table D-1 and Added new Feature Flags.
June 2015	3.22	Added MONITORX and MWAITX to Chapter 4.

Date	Revision	Description
October 2013	3.21	Added BMI2 instructions to Chapter 3. Added BZHI to Table F-1 on page 635. Changed CPUID Fn8000_0001_ECX[25] to reserved. Changed CPUID Fn8000_0007_EAX and _EDX[11] to reserved. Added CPUID Fn0000_0006_EDX[ARAT] (bit 2).
May 2013	3.20	Updated Appendix D "Instruction Subsets and CPUID Feature Flags" on page 587 to make instruction list comprehensive. Added a new Appendix E "Obtaining Processor Information Via the CPUID Instruction" on page 593 which describes all defined processor feature bits. Supersedes and replaces the *CPUID Specification* (PID # 25481). Previous Appendix E "Instruction Effects on RFLAGS" renumbered as Appendix F.
September 2012	3.19	Corrected the value specified for the most significant nibble of the encoding for the VPSHAx instructions in Table A-28 on page 544.
March 2012	3.18	Added MOVBE instruction reference page to Chapter 3 "General-Purpose Instruction Reference" on page 71. Added instruction reference pages for the RDFSBASE/RDGSBASE and WRFSBASE/WRGSBASE instructions to Chapter 3. Added opcodes for the instructions to the opcode maps in Appendix A.

Date	Revision	Description
December 2011	3.17	Corrected second byte of VEX C5 escape sequence in Figure 1-2 on page 5. Made multiple corrections to the description of register-indirect addressing in Section 1.4 on page 17. Corrected *mod* field value in third row of Figure 1-16 on page 25. Updated pseudocode definition (see Section 2.5.3 on page 57). Corrected exception tables for LZCNT and TZCNT instructions. Added discussion of UD opcodes to introduction of Appendix A. Provided ommitted definition of “B” used in the specification of operand types in opcode maps of Appendix A. Provided numerous corrections to instruction entries in opcode maps of Appendix A. Added ymm register mnemonic to Table A-32 on page 546 and Table A-34 on page 548. Changed notational convention for indicating addressing modes in Table A-33 on page 546, Table A-35 on page 549, Table A-36 on page 551, and Table A-37 on page 552; edited footnotes.
September 2011	3.16	Reworked “Instruction Byte Order” section of Chapter 1. See “Instruction Encoding Overview” on page 1. Added clarification: Execution of VMRUN is disallowed while in System Management Mode. Made wording for BMI and TBM feature flag indication consistent with other instructions. Moved BMI and TBM instructions to this volume from Volume 4. Added instruction reference page for CRC32 Instruction. Removed one cause of #GP fault from exception table for LAR and LSL instructions. Added three-byte, VEX, and XOP opcode maps to Appendix A. Revised description of RDPMC instruction. Corrected errors in description of CLFLUSH instruction. Corrected footnote of Table A-35 on page 549.
November 2009	3.15	Clarified MFENCE serializing behavior. Added multibyte variant to “NOP” on page 237. Corrected descriptive text to “CMPXCHG8B CMPXCHG16B” on page 151.
September 2007	3.14	Added minor clarifications and corrected typographical and formatting errors.

Date	Revision	Description
July 2007	3.13	Added the following instructions: LZCNT, POPCNT, MONITOR, and MWAIT. Reformatted information on instruction support indicated by CPUID feature bits into a table. Added minor clarifications and corrected typographical and formatting errors.
September 2006	3.12	Added minor clarifications and corrected typographical and formatting errors.
December 2005	3.11	Added SVM instructions; added PAUSE instructions; made factual changes.
January 2005	3.10	Clarified CPUID information in exception tables on instruction pages. Added information under “CPUID” on page 153. Made numerous small corrections.
September 2003	3.09	Corrected table of valid descriptor types for LAR and LSL instructions and made several minor formatting, stylistic and factual corrections. Clarified several technical definitions.
April 2003	3.08	Corrected description of the operation of flags for RCL, RCR, ROL, and ROR instructions. Clarified description of the MOVSXD and IMUL instructions. Corrected operand specification for the STOS instruction. Corrected opcode of SET*cc*, J*cc*, instructions. Added thermal control and thermal monitoring bits to CPUID instruction. Corrected exception tables for POPF, SFENCE, SUB, XLAT, IRET, LSL, MOV(CR*n*), SGDT/SIDT, SMSW, and STI instructions. Corrected many small typos and incorporated branding terminology.

Preface

About This Book

This book is part of a multivolume work entitled the *AMD64 Architecture Programmer's Manual*. This table lists each volume and its order number.

Title	Order No.
Volume 1: Application Programming	24592
Volume 2: System Programming	24593
Volume 3: General-Purpose and System Instructions	24594
Volume 4: 128-Bit and 256-Bit Media Instructions	26568
Volume 5: 64-Bit Media and x87 Floating-Point Instructions	26569

Audience

This volume (Volume 3) is intended for all programmers writing application or system software for a processor that implements the AMD64 architecture. Descriptions of general-purpose instructions assume an understanding of the application-level programming topics described in Volume 1. Descriptions of system instructions assume an understanding of the system-level programming topics described in Volume 2.

Organization

Volumes 3, 4, and 5 describe the AMD64 architecture's instruction set in detail. Together, they cover each instruction's mnemonic syntax, opcodes, functions, affected flags, and possible exceptions.

The AMD64 instruction set is divided into five subsets:

- General-purpose instructions
- System instructions
- Streaming SIMD Extensions–SSE (includes 128-bit and 256-bit media instructions)
- 64-bit media instructions (MMX™)
- x87 floating-point instructions

Several instructions belong to—and are described identically in—multiple instruction subsets.

This volume describes the general-purpose and system instructions. The index at the end cross-references topics within this volume. For other topics relating to the AMD64 architecture, and for

information on instructions in other subsets, see the tables of contents and indexes of the other volumes.

Conventions and Definitions

The following section **Notational Conventions** describes notational conventions used in this volume and in the remaining volumes of this *AMD64 Architecture Programmer's Manual*. This is followed by a **Definitions** section which lists a number of terms used in the manual along with their technical definitions. Finally, the **Registers** section lists the registers which are a part of the application programming model.

Notational Conventions

#GP(0)

An instruction exception—in this example, a general-protection exception with error code of 0.

1011b

A binary value—in this example, a 4-bit value.

F0EA_0B02h

A hexadecimal value. Underscore characters may be inserted to improve readability.

128

Numbers without an alpha suffix are decimal unless the context indicates otherwise.

7:4

A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first. Commas may be inserted to indicate gaps.

CPUID Fn*XXXX_XXXX_RRR*[*FieldName*]

Support for optional features or the value of an implementation-specific parameter of a processor can be discovered by executing the CPUID instruction on that processor. To obtain this value, software must execute the CPUID instruction with the function code *XXXX_XXXX*h in EAX and then examine the field *FieldName* returned in register *RRR*. If the "*_RRR*" notation is followed by "*_xYYY*", register ECX must be set to the value *YYY*h before executing CPUID. When *FieldName* is not given, the entire contents of register *RRR* contains the desired value. When determining optional feature support, if the bit identified by *FieldName* is set to a one, the feature is supported on that processor.

CR0–CR4

A register range, from register CR0 through CR4, inclusive, with the low-order register first.

CR0[PE], CR0.PE

Notation for referring to a field within a register—in this case, the PE field of the CR0 register.

CR0[PE] = 1, CR0.PE = 1

Notation indicating that the PE bit of the CR0 register has a value of 1.

DS:rSI

The contents of a memory location whose segment address is in the DS register and whose offset relative to that segment is in the rSI register.

EFER[LME] = 0, EFER.LME = 0

Notation indicating that the LME bit of the EFER register has a value of 0.

RFLAGS[13:12]

A field within a register identified by its bit range. In this example, corresponding to the IOPL field.

Definitions

Many of the following definitions assume an in-depth knowledge of the legacy x86 architecture. See "Related Documents" on page xxxvi for descriptions of the legacy x86 architecture.

128-bit media instructions

Instructions that operate on the various 128-bit vector data types. Supported within both the legacy SSE and extended SSE instruction sets.

256-bit media instructions

Instructions that operate on the various 256-bit vector data types. Supported within the extended SSE instruction set.

64-bit media instructions

Instructions that operate on the 64-bit vector data types. These are primarily a combination of MMX™ and 3DNow!™ instruction sets, with some additional instructions from the SSE1 and SSE2 instruction sets.

16-bit mode

Legacy mode or compatibility mode in which a 16-bit address size is active. See *legacy mode* and *compatibility mode*.

32-bit mode

Legacy mode or compatibility mode in which a 32-bit address size is active. See *legacy mode* and *compatibility mode*.

64-bit mode

A submode of *long mode*. In 64-bit mode, the default address size is 64 bits and new features, such as register extensions, are supported for system and application software.

absolute

Said of a displacement that references the base of a code segment rather than an instruction pointer. Contrast with *relative*.

biased exponent

The sum of a floating-point value's exponent and a constant bias for a particular floating-point data type. The bias makes the range of the biased exponent always positive, which allows reciprocation without overflow.

byte

Eight bits.

clear

To write a bit value of 0. Compare *set*.

compatibility mode

A submode of *long mode*. In compatibility mode, the default address size is 32 bits, and legacy 16-bit and 32-bit applications run without modification.

commit

To irreversibly write, in program order, an instruction's result to software-visible storage, such as a register (including flags), the data cache, an internal write buffer, or memory.

CPL

Current privilege level.

direct

Referencing a memory location whose address is included in the instruction's syntax as an immediate operand. The address may be an absolute or relative address. Compare *indirect*.

dirty data

Data held in the processor's caches or internal buffers that is more recent than the copy held in main memory.

displacement

A signed value that is added to the base of a segment (absolute addressing) or an instruction pointer (relative addressing). Same as *offset*.

doubleword

Two words, or four bytes, or 32 bits.

double quadword

Eight words, or 16 bytes, or 128 bits. Also called *octword*.

effective address size

The address size for the current instruction after accounting for the default address size and any address-size override prefix.

effective operand size

The operand size for the current instruction after accounting for the default operand size and any operand-size override prefix.

element

See *vector*.

exception

An abnormal condition that occurs as the result of executing an instruction. The processor's response to an exception depends on the type of the exception. For all exceptions except 128-bit media SIMD floating-point exceptions and x87 floating-point exceptions, control is transferred to the handler (or service routine) for that exception, as defined by the exception's vector. For floating-point exceptions defined by the IEEE 754 standard, there are both masked and unmasked responses. When unmasked, the exception handler is called, and when masked, a default response is provided instead of calling the handler.

flush

An often ambiguous term meaning (1) writeback, if modified, and invalidate, as in "flush the cache line," or (2) invalidate, as in "flush the pipeline," or (3) change a value, as in "flush to zero."

GDT

Global descriptor table.

IDT

Interrupt descriptor table.

IGN

Ignored. Value written is ignored by hardware. Value returned on a read is indeterminate. See *reserved*.

indirect

Referencing a memory location whose address is in a register or other memory location. The address may be an absolute or relative address. Compare *direct*.

IRB

The virtual-8086 mode interrupt-redirection bitmap.

IST

The long-mode interrupt-stack table.

IVT

The real-address mode interrupt-vector table.

LDT

Local descriptor table.

legacy x86

The legacy x86 architecture. See “Related Documents” on page xxxvi for descriptions of the legacy x86 architecture.

legacy mode

An operating mode of the AMD64 architecture in which existing 16-bit and 32-bit applications and operating systems run without modification. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Legacy mode has three submodes, *real mode*, *protected mode*, and *virtual-8086 mode*.

LIP

Linear Instruction Pointer. LIP = (CS.base + rIP).

long mode

An operating mode unique to the AMD64 architecture. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Long mode has two submodes, *64-bit mode* and *compatibility mode*.

lsb

Least-significant bit.

LSB

Least-significant byte.

main memory

Physical memory, such as RAM and ROM (but not cache memory) that is installed in a particular computer system.

mask

(1) A control bit that prevents the occurrence of a floating-point exception from invoking an exception-handling routine. (2) A field of bits used for a control purpose.

MBZ

Must be zero. If software attempts to set an MBZ bit to 1, a general-protection exception (#GP) occurs.

memory

Unless otherwise specified, *main memory*.

ModRM
: A byte following an instruction opcode that specifies address calculation based on mode (Mod), register (R), and memory (M) variables.

moffset
: A 16, 32, or 64-bit offset that specifies a memory operand directly, without using a ModRM or SIB byte.

msb
: Most-significant bit.

MSB
: Most-significant byte.

multimedia instructions
: A combination of *128-bit media instructions* and *64-bit media instructions*.

octword
: Same as *double quadword.*

offset
: Same as *displacement*.

overflow
: The condition in which a floating-point number is larger in magnitude than the largest, finite, positive or negative number that can be represented in the data-type format being used.

packed
: See *vector*.

PAE
: Physical-address extensions.

physical memory
: Actual memory, consisting of *main memory* and cache.

probe
: A check for an address in a processor's caches or internal buffers. *External probes* originate outside the processor, and *internal probes* originate within the processor.

procedure stack
: A portion of a stack segment in memory that is used to link procedures. Also known as a program

stack.

program stack
: See procedure stack.

protected mode
: A submode of *legacy mode*.

quadword
: Four words, or eight bytes, or 64 bits.

RAZ
: Read as zero. Value returned on a read is always zero (0) regardless of what was previously written. See *reserved*.

real-address mode
: See *real mode*.

real mode
: A short name for *real-address mode*, a submode of *legacy mode*.

relative
: Referencing with a displacement (also called offset) from an instruction pointer rather than the base of a code segment. Contrast with *absolute*.

reserved
: Fields marked as reserved may be used at some future time.

  To preserve compatibility with future processors, reserved fields require special handling when read or written by software. Software must not depend on the state of a reserved field (unless qualified as RAZ), nor upon the ability of such fields to return a previously written state.

  If a field is marked reserved without qualification, software must not change the state of that field; it must reload that field with the same value returned from a prior read.

  Reserved fields may be qualified as IGN, MBZ, RAZ, or SBZ (see definitions).

REX
: An instruction prefix that specifies a 64-bit operand size and provides access to additional registers.

RIP-relative addressing
: Addressing relative to the 64-bit RIP instruction pointer.

SBZ
: Should be zero. An attempt by software to set an SBZ bit to 1 results in undefined behavior.

shadow stack

A shadow stack is a separate, protected stack that is conceptually parallel to the procedure stack and used only by the shadow stack feature.

set

To write a bit value of 1. Compare *clear*.

SIB

A byte following an instruction opcode that specifies address calculation based on scale (S), index (I), and base (B).

SIMD

Single instruction, multiple data. See *vector*.

SSE

Streaming SIMD extensions instruction set. See *128-bit media instructions* and *64-bit media instructions*.

SSE2

Extensions to the SSE instruction set. See *128-bit media instructions* and *64-bit media instructions*.

SSE3

Further extensions to the SSE instruction set. See *128-bit media instructions.*

sticky bit

A bit that is set or cleared by hardware and that remains in that state until explicitly changed by software.

TOP

The x87 top-of-stack pointer.

TPR

Task-priority register (CR8).

TSS

Task-state segment.

underflow

The condition in which a floating-point number is smaller in magnitude than the smallest nonzero, positive or negative number that can be represented in the data-type format being used.

vector

(1) A set of integer or floating-point values, called *elements*, that are packed into a single operand. Most of the 128-bit and 64-bit media instructions use vectors as operands. Vectors are also called *packed* or *SIMD* (single-instruction multiple-data) operands.

(2) An index into an interrupt descriptor table (IDT), used to access exception handlers. Compare *exception*.

virtual-8086 mode

A submode of *legacy mode*.

word

Two bytes, or 16 bits.

x86

See *legacy x86*.

Registers

In the following list of registers, the names are used to refer either to a given register or to the contents of that register:

AH–DH

The high 8-bit AH, BH, CH, and DH registers. Compare *AL–DL*.

AL–DL

The low 8-bit AL, BL, CL, and DL registers. Compare *AH–DH*.

AL–r15B

The low 8-bit AL, BL, CL, DL, SIL, DIL, BPL, SPL, and R8B–R15B registers, available in 64-bit mode.

BP

Base pointer register.

CR*n*

Control register number *n*.

CS

Code segment register.

eAX–eSP

The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers. Compare *rAX–rSP*.

EFER

Extended features enable register.

eFLAGS

16-bit or 32-bit flags register. Compare *rFLAGS*.

EFLAGS

32-bit (extended) flags register.

eIP

16-bit or 32-bit instruction-pointer register. Compare *rIP*.

EIP

32-bit (extended) instruction-pointer register.

FLAGS

16-bit flags register.

GDTR

Global descriptor table register.

GPRs

General-purpose registers. For the 16-bit data size, these are AX, BX, CX, DX, DI, SI, BP, and SP. For the 32-bit data size, these are EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP. For the 64-bit data size, these include RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and R8–R15.

IDTR

Interrupt descriptor table register.

IP

16-bit instruction-pointer register.

LDTR

Local descriptor table register.

MSR

Model-specific register.

r8–r15

The 8-bit R8B–R15B registers, or the 16-bit R8W–R15W registers, or the 32-bit R8D–R15D registers, or the 64-bit R8–R15 registers.

rAX–rSP

The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers, or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers, or the 64-bit RAX, RBX, RCX, RDX, RDI, RSI, RBP, and RSP registers. Replace the placeholder *r* with nothing for 16-bit size, “E” for 32-bit size, or “R” for 64-bit size.

RAX

64-bit version of the EAX register.

RBP
64-bit version of the EBP register.

RBX
64-bit version of the EBX register.

RCX
64-bit version of the ECX register.

RDI
64-bit version of the EDI register.

RDX
64-bit version of the EDX register.

rFLAGS
16-bit, 32-bit, or 64-bit flags register. Compare *RFLAGS*.

RFLAGS
64-bit flags register. Compare *rFLAGS*.

rIP
16-bit, 32-bit, or 64-bit instruction-pointer register. Compare *RIP*.

RIP
64-bit instruction-pointer register.

RSI
64-bit version of the ESI register.

RSP
64-bit version of the ESP register.

SP
Stack pointer register.

SS
Stack segment register.

SSP
Shadow-stack pointer register.

TPR
Task priority register, a new register introduced in the AMD64 architecture to speed interrupt management.

TR

Task register.

Endian Order

The x86 and AMD64 architectures address memory using little-endian byte-ordering. Multibyte values are stored with their least-significant byte at the lowest byte address, and they are illustrated with their least significant byte at the right side. Strings are illustrated in reverse order, because the addresses of their bytes increase from right to left.

Related Documents

- Peter Abel, *IBM PC Assembly Language and Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Rakesh Agarwal, *80x86 Architecture & Programming: Volume II*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- AMD, *Software Optimization Guide for AMD Family 15h Processors*, order number 47414.
- AMD, *BIOS and Kernel Developer's Guide (BKDG)* for particular hardware implementations of older families of the AMD64 architecture.
- AMD, *Processor Programming Reference (PPR)* for particular hardware implementations of newer families of the AMD64 architecture.
- Don Anderson and Tom Shanley, *Pentium Processor System Architecture*, Addison-Wesley, New York, 1995.
- Nabajyoti Barkakati and Randall Hyde, *Microsoft Macro Assembler Bible*, Sams, Carmel, Indiana, 1992.
- Barry B. Brey, *8086/8088, 80286, 80386, and 80486 Assembly Language Programming*, Macmillan Publishing Co., New York, 1994.
- Barry B. Brey, *Programming the 80286, 80386, 80486, and Pentium Based Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Ralf Brown and Jim Kyle, *PC Interrupts*, Addison-Wesley, New York, 1994.
- Penn Brumm and Don Brumm, *80386/80486 Assembly Language Programming*, Windcrest McGraw-Hill, 1993.
- Geoff Chappell, *DOS Internals*, Addison-Wesley, New York, 1994.
- Chips and Technologies, Inc. *Super386 DX Programmer's Reference Manual*, Chips and Technologies, Inc., San Jose, 1992.
- John Crawford and Patrick Gelsinger, *Programming the 80386*, Sybex, San Francisco, 1987.
- Cyrix Corporation, *5x86 Processor BIOS Writer's Guide*, Cyrix Corporation, Richardson, TX, 1995.
- Cyrix Corporation, *M1 Processor Data Book*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor MMX Extension Opcode Table*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor Data Book*, Cyrix Corporation, Richardson, TX, 1997.
- Ray Duncan, *Extending DOS: A Programmer's Guide to Protected-Mode DOS*, Addison Wesley, NY, 1991.
- William B. Giles, *Assembly Language Programming for the Intel 80xxx Family*, Macmillan, New York, 1991.
- Frank van Gilluwe, *The Undocumented PC*, Addison-Wesley, New York, 1994.

- John L. Hennessy and David A. Patterson, *Computer Architecture*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.
- Thom Hogan, *The Programmer's PC Sourcebook*, Microsoft Press, Redmond, WA, 1991.
- Hal Katircioglu, *Inside the 486, Pentium, and Pentium Pro*, Peer-to-Peer Communications, Menlo Park, CA, 1997.
- IBM Corporation, *486SLC Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *486SLC2 Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *80486DX2 Processor Floating Point Instructions*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *80486DX2 Processor BIOS Writer's Guide*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *Blue Lightning 486DX2 Data Book*, IBM Corporation, Essex Junction, VT, 1994.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, ANSI/IEEE Std 854-1987.
- Muhammad Ali Mazidi and Janice Gillispie Mazidi, *80X86 IBM PC and Compatible Computers*, Prentice-Hall, Englewood Cliffs, NJ, 1997.
- Hans-Peter Messmer, *The Indispensable Pentium Book*, Addison-Wesley, New York, 1995.
- Karen Miller, *An Assembly Language Introduction to Computer Architecture: Using the Intel Pentium*, Oxford University Press, New York, 1999.
- Stephen Morse, Eric Isaacson, and Douglas Albert, *The 80386/387 Architecture*, John Wiley & Sons, New York, 1987.
- NexGen Inc., *Nx586 Processor Data Book*, NexGen Inc., Milpitas, CA, 1993.
- NexGen Inc., *Nx686 Processor Data Book*, NexGen Inc., Milpitas, CA, 1994.
- Bipin Patwardhan, *Introduction to the Streaming SIMD Extensions in the Pentium III*, www.x86.org/articles/sse_pt1/ simd1.htm, June, 2000.
- Peter Norton, Peter Aitken, and Richard Wilton, *PC Programmer's Bible*, Microsoft Press, Redmond, WA, 1993.
- *PharLap 386|ASM Reference Manual*, Pharlap, Cambridge MA, 1993.
- *PharLap TNT DOS-Extender Reference Manual*, Pharlap, Cambridge MA, 1995.
- Sen-Cuo Ro and Sheau-Chuen Her, *i386/i486 Advanced Programming*, Van Nostrand Reinhold, New York, 1993.
- Jeffrey P. Royer, *Introduction to Protected Mode Programming*, course materials for an onsite class, 1992.

- Tom Shanley, *Protected Mode System Architecture*, Addison Wesley, NY, 1996.
- SGS-Thomson Corporation, *80486DX Processor SMM Programming Manual*, SGS-Thomson Corporation, 1995.
- Walter A. Triebel, *The 80386DX Microprocessor*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- John Wharton, *The Complete x86*, MicroDesign Resources, Sebastopol, California, 1994.

1 Instruction Encoding

AMD64 technology instructions are encoded as byte strings of variable length. The order and meaning of each byte of an instruction's encoding is specified by the architecture. Fields within the encoding specify the instruction's basic operation, the location of the one or more source operands, and the destination of the result of the operation. Data to be used in the execution of the instruction or the computation of addresses for memory-based operands may also be included. This section describes the general format and parameters used by all instructions.

For information on the specific encoding(s) for each instruction, see:

- Chapter 3, "General-Purpose Instruction Reference."
- Chapter 4, "System Instruction Reference."
- "SSE Instruction Reference" in Volume 4.
- "64-Bit Media Instruction Reference" in Volume 5.
- "x87 Floating-Point Instruction Reference" in Volume 5.

For information on determining the instruction form and operands specified by a given binary encoding, see Appendix A.

1.1 Instruction Encoding Overview

An instruction is encoded as a string between one and 15 bytes in length. The entire sequence of bytes that represents an instruction, including the basic operation, the location of source and destination operands, any operation modifiers, and any immediate and/or displacement values, is called the instruction encoding.The following sections discuss instruction encoding syntax and representation in memory.

1.1.1 Encoding Syntax

Figure 1-1 provides a schematic representation of the encoding syntax of an instruction.

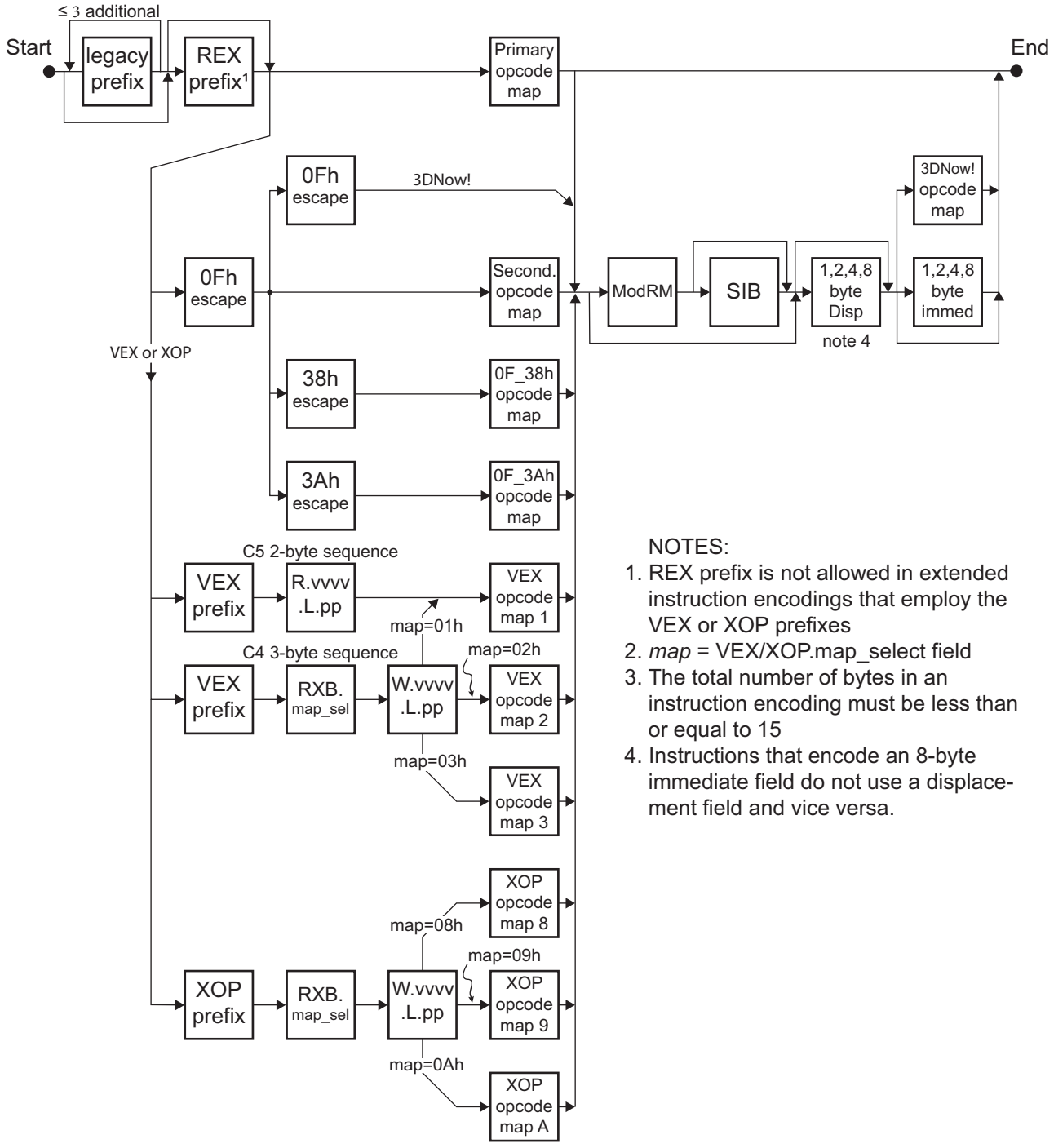


Figure 1-1. Instruction Encoding Syntax

Each square in this diagram represents an instruction byte of a particular type and function. To understand the diagram, follow the connecting paths in the direction indicated by the arrows from "Start" to "End." The squares passed through as the graph is traversed indicate the order and number of

bytes used to encode the instruction. Note that the path shown above the legacy prefix byte loops back indicating that up to four additional prefix bytes may be used in the encoding of a single instruction. Branches indicate points in the syntax where alternate semantics are employed based on the instruction being encoded. The “VEX or XOP” gate across the path leading down to the VEX prefix and XOP prefix blocks means that only extended instructions employing the VEX or XOP prefixes use this particular branch of the syntax diagram. This diagram will be further explained in the sections that follow.

1.1.1.1 Legacy Prefixes

As shown in the figure, an instruction optionally begins with up to five *legacy prefixes*. These prefixes are described in “Summary of Legacy Prefixes” on page 6. The legacy prefixes modify an instruction’s default address size, operand size, or segment, or they invoke a special function such as modification of the opcode, atomic bus-locking, or repetition.

In the encoding of most SSE instructions, a legacy operand-size or repeat prefix is repurposed to modify the opcode. For the extended encodings utilizing the XOP or VEX prefixes, these prefixes are not allowed.

1.1.1.2 REX Prefix

Following the optional legacy prefix or prefixes, the REX prefix can be used in 64-bit mode to access the AMD64 register number and size extensions. Refer to the diagram in “Application-Programming Register Set” in Volume 1 for an illustration of these facilities. If a REX prefix is used, it must immediately precede the opcode byte or the first byte of a legacy *escape sequence*. The REX prefix is not allowed in extended instruction encodings using the VEX or XOP encoding escape prefixes. Violating this restriction results in an #UD exception.

1.1.1.3 Opcode

The *opcode* is a single byte that specifies the basic operation of an instruction. Every instruction requires an opcode. The correspondence between the binary value of an opcode and the operation it represents is presented in a table called an *opcode map*. Because it is indexed by an 8-bit value, an opcode map has 256 entries. Since there are more than 256 instructions defined by the architecture, multiple different opcode maps must be defined and the selection of these alternate opcode maps must be encoded in the instruction. Escape sequences provide this access to alternate opcode maps.

If there are no opcode escapes, the primary (“one-byte”) opcode map is used. In the figure this is the path pointing from the REX Prefix block to the Primary opcode map block.

Section , “Primary Opcode Map” of Appendix A provides details concerning this opcode map.

1.1.1.4 Escape Sequences

Escape sequences allow access to alternate opcode maps that are distinct from the primary opcode map. Escape sequences may be one, two, or three bytes in length and begin with a unique byte value designated for this purpose in the primary opcode map. Escape sequences are of two distinct types:

legacy escape sequences and extended escape sequences. The legacy escape sequences will be covered here. For more details on the extended escape sequences, see “VEX and XOP Prefixes” on page 16.

Legacy Escape Sequences

The legacy syntax allows one 1-byte escape sequence (0Fh), and three 2-byte escape sequences (0Fh, 0Fh; 0Fh, 38h; and 0Fh, 3Ah). The 1-byte legacy escape sequence 0Fh selects the secondary (“two-byte”) opcode map. In legacy terminology, the sequence [0Fh, *opcode*] is called a two-byte opcode. See Section , “Secondary Opcode Map” of Appendix A for details concerning this opcode map.

The 2-byte escape sequence 0F, 0Fh selects the 3DNow! opcode map which is indexed using an immediate byte rather than an opcode byte. In this case, the byte following the escape sequence is the ModRM byte instead of the opcode byte. In Figure 1-1 this is indicated by the path labeled “3DNow!” leaving the second 0Fh escape block. Details concerning the 3DNow! opcode map are presented in Section A.1.2, “3DNow!™ Opcodes” of Appendix A.

The 2-byte escape sequences [0Fh, 38h] and [0Fh, 3Ah] respectively select the 0F_38h opcode map and the 0F_3Ah opcode map. These are used primarily to encode SSE instructions and are described in Section , “0F_38h and 0F_3Ah Opcode Maps” of Appendix A.

1.1.1.5 ModRM and SIB Bytes

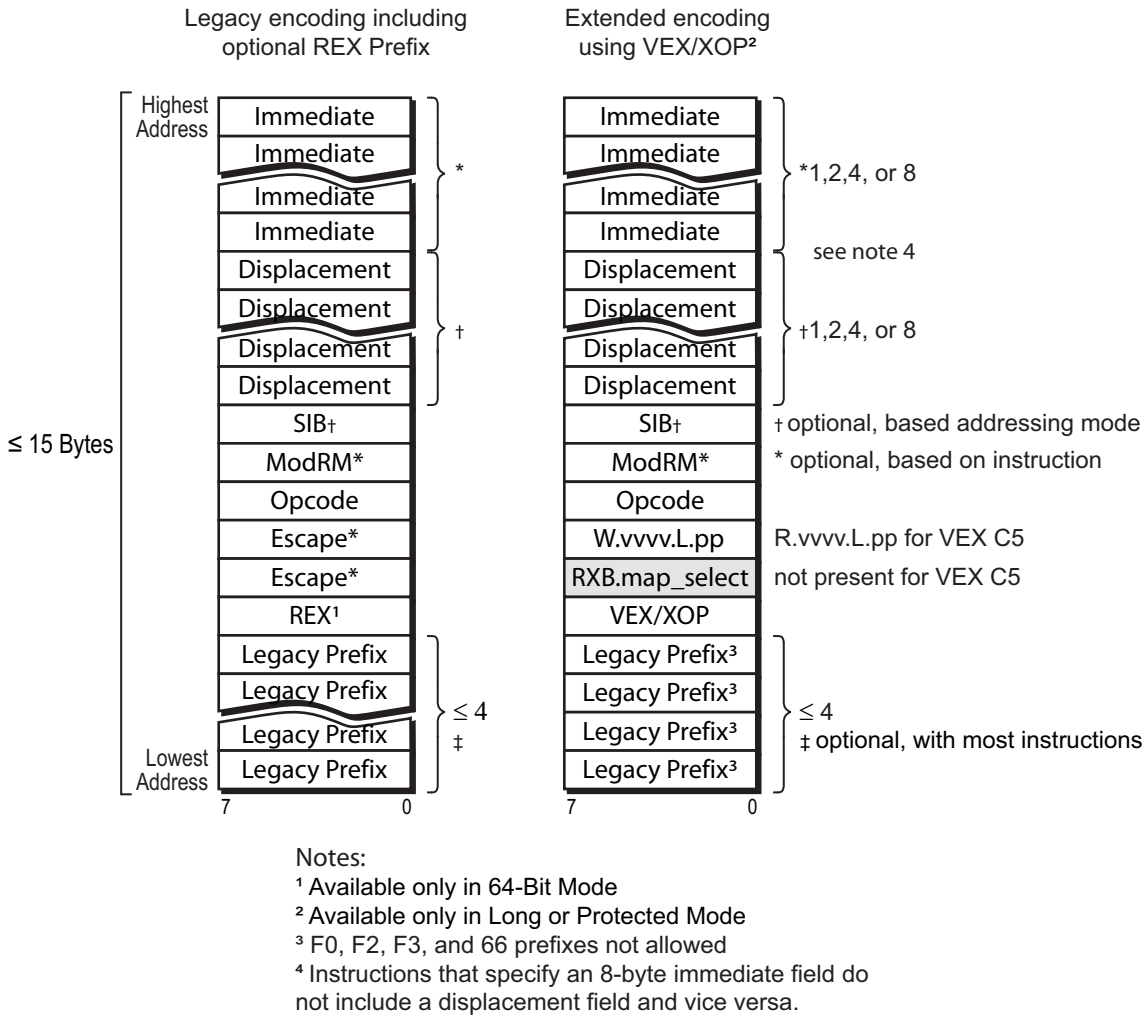
The opcode can be followed by a *mode-register-memory* (ModRM) byte, which further describes the operation and/or operands. The ModRM byte may also be followed by a *scale-index-base* (SIB) byte, which is used to specify indexed register-indirect forms of memory addressing. The ModRM and SIB bytes are described in “ModRM Byte Format” on page 17. Their legacy functions can be augmented by the REX prefix (see “REX Prefix” on page 14) or the VEX and XOP escape sequences (See “VEX and XOP Prefixes” on page 16).

1.1.1.6 Displacement and Immediate Fields

The instruction encoding may end with a 1-, 2-, or 4-byte *displacement* field and/or a 1-, 2-, or 4-byte *immediate* field depending on the instruction and/or the addressing mode. Specific instructions also allow either an 8-byte immediate field or an 8-byte displacement field.

1.1.2 Representation in Memory

Instructions are stored in memory in little-endian order. The first byte of an instruction is stored at the lowest memory address, as shown in Figure 1-2 below. Since instructions are strings of bytes, they may start at any memory address. The total instruction length must be less than or equal to 15. If this limit is exceeded, a general-protection exception results.



Notes:
¹ Available only in 64-Bit Mode
² Available only in Long or Protected Mode
³ F0, F2, F3, and 66 prefixes not allowed
⁴ Instructions that specify an 8-byte immediate field do not include a displacement field and vice versa.

Figure 1-2. An Instruction as Stored in Memory

1.2 Instruction Prefixes

Instruction prefixes are of two types: *instruction modifier* prefixes and *encoding escape* prefixes. Instruction modifier prefixes can change the operation of the instruction (including causing its execution to repeat), change its operand types, specify an alternate operand size, augment register specification, or even change the interpretation of the opcode byte.

The instruction modifier prefixes comprise the legacy prefixes and the REX prefix. The legacy prefixes are discussed in the next section. The REX prefix is discussed in “REX Prefix” on page 14.

Encoding escape prefixes, on the other hand, signal that the two or three bytes that follow obey a different encoding syntax. As a group, the encoding escape prefix and its subsequent bytes constitute a multi-byte escape sequence. These multi-byte escape sequences perform functions similar to that of

the instruction modifier prefixes, but they also provide a means to directly specify alternate opcode maps.

The currently defined encoding escape prefixes are the VEX and XOP prefixes. They are discussed further in the section entitled “VEX and XOP Prefixes” on page 16.

1.2.1 Summary of Legacy Prefixes

Table 1-1 on page 7 shows the legacy prefixes. The legacy prefixes are organized into five groups, as shown in the left-most column of Table 1-1. An instruction encoding may include a maximum of one prefix from each of the five groups. The legacy prefixes can appear in any order within the position shown in Figure 1-1 for legacy prefixes. The result of using multiple prefixes from a single group is undefined.

Some of the restrictions on legacy prefixes are:

- *Operand-Size Override*—This prefix only affects the operand size for general-purpose instructions or for other instructions whose source or destination is a general-pupose register. When used in the encoding of SIMD and some other instructions, this prefix is repurposed to modify the opcode.
- *Address-Size Override*—This prefix only affects the address size of memory operands.
- *Segment Override*—In 64-bit mode, the CS, DS, ES, and SS segment override prefixes are ignored.
- *LOCK Prefix*—This prefix is allowed only with certain instructions that modify memory.
- *Repeat Prefixes*—These prefixes affect only certain string instructions. When used in the encoding of SIMD and some other instructions, these prefixes are repurposed to modify the opcode.

Note that Lock and Repeat prefixes are in effect mutually exclusive when used as instruction modifiers, in that there are no instructions for which both are meaningful.

Table 1-1. Legacy Instruction Prefixes

Prefix Group[1]	Mnemonic	Prefix Byte (Hex)	Description
Operand-Size Override	none	66[2]	Changes the default operand size of a memory or register operand, as shown in Table 1-2 on page 8.
Address-Size Override	none	67[3]	Changes the default address size of a memory operand, as shown in Table 1-3 on page 9.
Segment Override	CS	2E[4]	Forces use of the current CS segment for memory operands.
	DS	3E[4]	Forces use of the current DS segment for memory operands.
	ES	26[4]	Forces use of the current ES segment for memory operands.
	FS	64	Forces use of the current FS segment for memory operands.
	GS	65	Forces use of the current GS segment for memory operands.
	SS	36[4]	Forces use of the current SS segment for memory operands.
Lock	LOCK	F0[5]	Causes certain kinds of memory read-modify-write instructions to occur atomically.
Repeat	REP	F3[6]	Repeats a string operation (INS, MOVS, OUTS, LODS, and STOS) until the rCX register equals 0.
	REPE or REPZ		Repeats a compare-string or scan-string operation (CMPSx and SCASx) until the rCX register equals 0 or the zero flag (ZF) is cleared to 0.
	REPNE or REPNZ	F2[6]	Repeats a compare-string or scan-string operation (CMPSx and SCASx) until the rCX register equals 0 or the zero flag (ZF) is set to 1.

Notes:

1. *A single instruction should include no more than one prefix from each of the Override prefix groups plus either a Lock or Repeat prefix, when used as instruction modifiers.*
2. *When used in the encoding of SIMD and some other instructions, this prefix is repurposed to extend the opcode. The prefix is ignored by 64-bit media floating-point (3DNow!™) instructions. See "Instructions that Cannot Use the Operand-Size Prefix" on page 8.*
3. *This prefix also changes the size of the RCX register when used as an implied count register.*
4. *In 64-bit mode, the CS, DS, ES, and SS segment overrides are ignored.*
5. *The LOCK prefix should not be used for instructions other than those listed in "Lock Prefix" on page 11.*
6. *This prefix should be used only with compare-string and scan-string instructions. When used in the encoding of SIMD and some other instructions, the prefix is repurposed to extend the opcode.*

1.2.2 Operand-Size Override Prefix

The default operand size for an instruction is determined by a combination of its opcode, the D (default) bit in the current code-segment descriptor, and the current operating mode, as shown in Table 1-2. The operand-size override prefix (66h) selects the non-default operand size. The prefix can

be used with any general-purpose instruction that accesses non-fixed-size operands in memory or general-purpose registers (GPRs), and it can also be used with the x87 FLDENV, FNSTENV, FNSAVE, and FRSTOR instructions.

In 64-bit mode, the prefix allows mixing of 16-bit, 32-bit, and 64-bit data on an instruction-by-instruction basis. In compatibility and legacy modes, the prefix allows mixing of 16-bit and 32-bit operands on an instruction-by-instruction basis.

Table 1-2. Operand-Size Overrides

Operating Mode		Default Operand Size (Bits)	Effective Operand Size (Bits)	Instruction Prefix[1]	
				66h	REX.W[3]
Long Mode	64-Bit Mode	32[2]	64	don't care	yes
			32	no	no
			16	yes	no
	Compatibility Mode	32	32	no	Not Appli-cable
			16	yes	
		16	32	yes	
			16	no	
Legacy Mode (Protected, Virtual-8086, or Real Mode)		32	32	no	
			16	yes	
		16	32	yes	
			16	no	

Notes:
1. *A "no' indicates that the default operand size is used.*
2. *This is the typical default, although some instructions default to other operand sizes. See Appendix B, "General-Purpose Instructions in 64-Bit Mode," for details.*
3. *See "REX Prefix" on page 14.*

In 64-bit mode, most instructions default to a 32-bit operand size. For these instructions, a REX prefix (page 14) can specify a 64-bit operand size, and a 66h prefix specifies a 16-bit operand size. The REX prefix takes precedence over the 66h prefix. However, if an instruction defaults to a 64-bit operand size, it does not need a REX prefix and it can only be overridden to a 16-bit operand size. It cannot be overridden to a 32-bit operand size, because there is no 32-bit operand-size override prefix in 64-bit mode. Two groups of instructions have a default 64-bit operand size in 64-bit mode:

- Near branches. For details, see "Near Branches in 64-Bit Mode" in Volume 1.
- All instructions, except far branches, that implicitly reference the RSP. For details, see "Stack Operation" in Volume 1.

Instructions that Cannot Use the Operand-Size Prefix. The operand-size prefix should be used only with general-purpose instructions and the x87 FLDENV, FNSTENV, FNSAVE, and FRSTOR

instructions, in which the prefix selects between 16-bit and 32-bit operand size. The prefix is ignored by all other x87 instructions and by 64-bit media floating-point (3DNow!™) instructions.

For other instructions (mostly SIMD instructions) the 66h, F2h, and F3h prefixes are used as opcode extensions to extend the instruction encoding space in the 0Fh, 0F_38h, and 0F_3Ah opcode maps.

Operand-Size and REX Prefixes. The W bit field of the REX prefix takes precedence over the 66h prefix. See “REX.W: Operand width (Bit 3)” on page 23 for details.

1.2.3 Address-Size Override Prefix

The default address size for instructions that access non-stack memory is determined by the current operating mode, as shown in Table 1-3. The address-size override prefix (67h) selects the non-default address size. Depending on the operating mode, this prefix allows mixing of 16-bit and 32-bit, or of 32-bit and 64-bit addresses, on an instruction-by-instruction basis. The prefix changes the address size for memory operands. It also changes the size of the RCX register for instructions that use RCX implicitly.

For instructions that implicitly access the stack segment (SS), the address size for stack accesses is determined by the D (default) bit in the stack-segment descriptor. In 64-bit mode, the D bit is ignored, and all stack references have a 64-bit address size. However, if an instruction accesses both stack and non-stack memory, the address size of the non-stack access is determined as shown in Table 1-3.

Table 1-3. Address-Size Overrides

Operating Mode		Default Address Size (Bits)	Effective Address Size (Bits)	Address-Size Prefix (67h)[1] Required?
Long Mode	64-Bit Mode	64	64	no
			32	yes
	Compatibility Mode	32	32	no
			16	yes
		16	32	yes
			16	no
Legacy Mode (Protected, Virtual-8086, or Real Mode)		32	32	no
			16	yes
		16	32	yes
			16	no

Notes:
1. *A “no” indicates that the default address size is used.*

As Table 1-3 shows, the default address size is 64 bits in 64-bit mode. The size can be overridden to 32 bits, but 16-bit addresses are not supported in 64-bit mode. In compatibility and legacy modes, the default address size is 16 bits or 32 bits, depending on the operating mode (see “Processor

Initialization and Long Mode Activation" in Volume 2 for details). In these modes, the address-size prefix selects the non-default size, but the 64-bit address size is not available.

Certain instructions reference pointer registers or count registers implicitly, rather than explicitly. In such instructions, the address-size prefix affects the size of such addressing and count registers, just as it does when such registers are explicitly referenced. Table 1-4 lists all such instructions and the registers referenced using the three possible address sizes.

Table 1-4. Pointer and Count Registers and the Address-Size Prefix

Instruction	Pointer or Count Register		
	16-Bit Address Size	32-Bit Address Size	64-Bit Address Size
CMPS, **CMPSB**, **CMPSW**, **CMPSD, CMPSQ**—Compare Strings	SI, DI, CX	ESI, EDI, ECX	RSI, RDI, RCX
INS, INSB, INSW, INSD—Input String	DI, CX	EDI, ECX	RDI, RCX
JCXZ, **JECXZ, JRCXZ**—Jump on CX/ECX/RCX Zero	CX	ECX	RCX
LODS, **LODSB**, **LODSW**, **LODSD, LODSQ**—Load String	SI, CX	ESI, ECX	RSI, RCX
LOOP, LOOPE, LOOPNZ, **LOOPNE, LOOPZ**—Loop	CX	ECX	RCX
MOVS, **MOVSB**, **MOVSW**, **MOVSD, MOVSQ**—Move String	SI, DI, CX	ESI, EDI, ECX	RSI, RDI, RCX
OUTS, OUTSB, OUTSW, OUTSD—Output String	SI, CX	ESI, ECX	RSI, RCX
REP, REPE, REPNE, REPNZ, REPZ—Repeat Prefixes	CX	ECX	RCX
SCAS, **SCASB**, **SCASW**, **SCASD, SCASQ**—Scan String	DI, CX	EDI, ECX	RDI, RCX
STOS, **STOSB**, **STOSW**, **STOSD, STOSQ**—Store String	DI, CX	EDI, ECX	RDI, RCX
XLAT, **XLATB**—Table Look-up Translation	BX	EBX	RBX

1.2.4 Segment-Override Prefixes

Segment overrides can be used only with instructions that reference non-stack memory. Most instructions that reference memory are encoded with a ModRM byte (page 17). The default segment

for such memory-referencing instructions is implied by the base register indicated in its ModRM byte, as follows:

- *Instructions that Reference a Non-Stack Segment*—If an instruction encoding references any base register other than rBP or rSP, or if an instruction contains an immediate offset, the default segment is the data segment (DS). These instructions can use the segment-override prefix to select one of the non-default segments, as shown in Table 1-5.
- *String Instructions*—String instructions reference two memory operands. By default, they reference both the DS and ES segments (DS:rSI and ES:rDI). These instructions can override their DS-segment reference, as shown in Table 1-5, but they cannot override their ES-segment reference.
- *Instructions that Reference the Stack Segment*—If an instruction's encoding references the rBP or rSP base register, the default segment is the stack segment (SS). All instructions that reference the stack (push, pop, call, interrupt, return from interrupt) use SS by default. These instructions cannot use the segment-override prefix.

Table 1-5. Segment-Override Prefixes

Mnemonic	Prefix Byte (Hex)	Description
CS[1]	2E	Forces use of current CS segment for memory operands.
DS[1]	3E	Forces use of current DS segment for memory operands.
ES[1]	26	Forces use of current ES segment for memory operands.
FS	64	Forces use of current FS segment for memory operands.
GS	65	Forces use of current GS segment for memory operands.
SS[1]	36	Forces use of current SS segment for memory operands.

Notes:
1. In 64-bit mode, the CS, DS, ES, and SS segment overrides are ignored.

Segment Overrides in 64-Bit Mode. In 64-bit mode, the CS, DS, ES, and SS segment-override prefixes have no effect. These four prefixes are not treated as segment-override prefixes for the purposes of multiple-prefix rules. Instead, they are treated as null prefixes.

The FS and GS segment-override prefixes are treated as true segment-override prefixes in 64-bit mode. Use of the FS or GS prefix causes their respective segment bases to be added to the effective address calculation. See "FS and GS Registers in 64-Bit Mode" in Volume 2 for details.

1.2.5 Lock Prefix

The LOCK prefix causes certain kinds of memory read-modify-write instructions to occur atomically. The mechanism for doing so is implementation-dependent (for example, the mechanism may involve bus signaling or packet messaging between the processor and a memory controller). The prefix is intended to give the processor exclusive use of shared memory in a multiprocessor system.

The LOCK prefix can only be used with forms of the following instructions that write a memory operand: ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, and XOR. An invalid-opcode exception occurs if the LOCK prefix is used with any other instruction.

1.2.6 Repeat Prefixes

The repeat prefixes cause repetition of certain instructions that load, store, move, input, or output strings. The prefixes should only be used with such string instructions. Two pairs of repeat prefixes, REPE/REPZ and REPNE/REPNZ, perform the same repeat functions for certain compare-string and scan-string instructions. The repeat function uses rCX as a count register. The size of rCX is based on address size, as shown in Table 1-4 on page 10.

REP. The REP prefix repeats its associated string instruction the number of times specified in the counter register (rCX). It terminates the repetition when the value in rCX reaches 0. The prefix can be used with the INS, LODS, MOVS, OUTS, and STOS instructions. Table 1-6 shows the valid REP prefix opcodes.

Table 1-6. REP Prefix Opcodes

Mnemonic	Opcode
REP INS *reg/mem8*, DX REP INSB	F3 6C
REP INS *reg/mem16/32*, DX REP INSW REP INSD	F3 6D
REP LODS *mem8* REP LODSB	F3 AC
REP LODS *mem16/32/64* REP LODSW REP LODSD REP LODSQ	F3 AD
REP MOVS *mem8*, *mem8* REP MOVSB	F3 A4
REP MOVS *mem16/32/64*, *mem16/32/64* REP MOVSW REP MOVSD REP MOVSQ	F3 A5
REP OUTS DX, *reg/mem8* REP OUTSB	F3 6E

Table 1-6. REP Prefix Opcodes (continued)

Mnemonic	Opcode
REP OUTS DX, *reg/mem16/32* REP OUTSW REP OUTSD	F3 6F
REP STOS *mem8* REP STOSB	F3 AA
REP STOS *mem16/32/64* REP STOSW REP STOSD REP STOSQ	F3 AB

REPE and REPZ. REPE and REPZ are synonyms and have identical opcodes. These prefixes repeat their associated string instruction the number of times specified in the counter register (rCX). The repetition terminates when the value in rCX reaches 0 or when the zero flag (ZF) is cleared to 0. The REPE and REPZ prefixes can be used with the CMPS, CMPSB, CMPSD, CMPSW, SCAS, SCASB, SCASD, and SCASW instructions. Table 1-7 shows the valid REPE and REPZ prefix opcodes.

Table 1-7. REPE and REPZ Prefix Opcodes

Mnemonic	Opcode
REPx CMPS *mem8*, *mem8* REPx CMPSB	F3 A6
REPx CMPS *mem16/32/64*, *mem16/32/64* REPx CMPSW REPx CMPSD REPx CMPSQ	F3 A7
REPx SCAS *mem8* REPx SCASB	F3 AE
REPx SCAS *mem16/32/64* REPx SCASW REPx SCASD REPx SCASQ	F3 AF

REPNE and REPNZ. REPNE and REPNZ are synonyms and have identical opcodes. These prefixes repeat their associated string instruction the number of times specified in the counter register (rCX). The repetition terminates when the value in rCX reaches 0 or when the zero flag (ZF) is set to 1. The REPNE and REPNZ prefixes can be used with the CMPS, CMPSB, CMPSD, CMPSW, SCAS, SCASB, SCASD, and SCASW instructions. Table 1-8 on page 14 shows the valid REPNE and REPNZ prefix opcodes.

Table 1-8. REPNE and REPNZ Prefix Opcodes

Mnemonic	Opcode
REPNx CMPS *mem8*, *mem8* REPNx CMPSB	F2 A6
REPNx CMPS *mem16/32/64*, *mem16/32/64* REPNx CMPSW REPNx CMPSD REPNx CMPSQ	F2 A7
REPNx SCAS *mem8* REPNx SCASB	F2 AE
REPNx SCAS *mem16/32/64* REPNx SCASW REPNx SCASD REPNx SCASQ	F2 AF

Instructions that Cannot Use Repeat Prefixes. In general, the repeat prefixes should only be used in the string instructions listed in tables 1-6, 1-7, and 1-8 above. For other instructions (mostly SIMD instructions) the 66h, F2h, and F3h prefixes are used as instruction modifiers to extend the instruction encoding space in the 0Fh, 0F_38h, and 0F_3Ah opcode maps.

Optimization of Repeats. Depending on the hardware implementation, the repeat prefixes can have a setup overhead. If the repeated count is variable, the overhead can sometimes be avoided by substituting a simple loop to move or store the data. Repeated string instructions can be expanded into equivalent sequences of inline loads and stores or a sequence of stores can be used to emulate a REP STOS.

For repeated string moves, performance can be maximized by moving the largest possible operand size. For example, use REP MOVSD rather than REP MOVSW and REP MOVSW rather than REP MOVSB. Use REP STOSD rather than REP STOSW and REP STOSW rather than REP MOVSB.

Depending on the hardware implementation, string moves with the direction flag (DF) cleared to 0 (up) may be faster than string moves with DF set to 1 (down). DF = 1 is only needed for certain cases of overlapping REP MOVS, such as when the source and the destination overlap.

1.2.7 REX Prefix

The REX prefix, available in 64-bit mode, enables use of the AMD64 register and operand size extensions. Unlike the legacy instruction modification prefixes, REX is not a single unique value, but occupies a range (40h to 4Fh). Figure 1-1 on page 2 shows how the REX prefix fits within the encoding syntax of instructions.

The REX prefix enables the following features in 64-bit mode:

- Use of the extended GPR (Figure 2-3 on page 39) and YMM/XMM registers (Figure 2-8 on page 44).

- Use of the 64-bit operand size when accessing GPRs.
- Use of the extended control and debug registers, as described in Section 2.4 “Registers” in Volume 2.
- Use of the uniform byte registers (AL–R15).

REX contains five fields. The upper nibble is unique to the REX prefix and identifies it is as such. The lower nibble is divided into four 1-bit fields (W, R, X, and B). See below for a discussion of these fields.Figure 1-3 below shows the format of the REX prefix. Since each bit of the lower nibble can be a 1 or a 0, REX spans one full row of the primary opcode map occupying entries 40h through 4Fh.

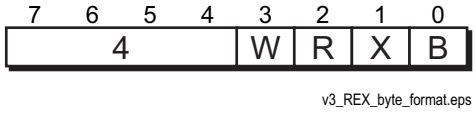


Figure 1-3. REX Prefix Format

A REX prefix is normally required with an instruction that accesses a 64-bit GPR or one of the extended GPR or YMM/XMM registers. A few instructions have an operand size that defaults to (or is fixed at) 64 bits in 64-bit mode, and thus do not need a REX prefix. These instructions are listed in Table 1-9 below.

Table 1-9. Instructions Not Requiring REX Prefix in 64-Bit Mode

CALL (Near)	POP reg/mem
ENTER	POP reg
J*cc*	POP FS
JrCXZ	POP GS
JMP (Near)	POPF, POPFD, POPFQ
LEAVE	PUSH imm8
LGDT	PUSH imm32
LIDT	PUSH reg/mem
LLDT	PUSH reg
LOOP	PUSH FS
LOOP*cc*	PUSH GS
LTR	PUSHF, PUSHFD, PUSHFQ
MOV CR*n*	RET (Near)
MOV DR*n*	

An instruction may have only one REX prefix which must immediately precede the opcode or first escape byte in the instruction encoding. The use of a REX prefix in an instruction that does not access an extended register is ignored. The instruction-size limit of 15 bytes applies to instructions that contain a REX prefix.

Implications for INC and DEC Instructions

The REX prefix values are taken from the 16 single-byte INC and DEC instructions, one for each of the eight legacy GPRs. Therefore, these single-byte opcodes for INC and DEC are not available in 64-bit mode, although they are available in legacy and compatibility modes. The functionality of these INC and DEC instructions is still available in 64-bit mode, however, using the ModRM forms of those instructions (opcodes FF /0 and FF /1).

1.2.8 VEX and XOP Prefixes

The extended instruction encoding syntax, available in protected and long modes, provides one 2-byte and three 3-byte escape sequences introduced by either the VEX or XOP prefixes. These multi-byte sequences not only select opcode maps, they also provide instruction modifiers similar to, but in lieu of, the REX prefix.

The 2-byte escape sequence initiated by the VEX C5h prefix implies a map_select encoding of 1. The three-byte escape sequences, initiated by the VEX C4h prefix or the XOP (8Fh) prefix, select the target opcode map explicitly via the VEX/XOP.map_select field. The five-bit VEX.map_select field allows the selection of one of 31 different opcode maps (opcode map 00h is reserved). The XOP.map_select field is restricted to the range 08h – 1Fh and thus can only select one of 24 different opcode maps.

The VEX and XOP escape sequences contain fields that extend register addressing to a total of 16, increase the operand specification capability to four operands, and modify the instruction operation.

The extended SSE instruction subsets AVX, AES, CLMU, FMA, FMA4, and XOP and a few non-SSE instructions utilize the extended encoding syntax. See “Encoding Using the VEX and XOP Prefixes” on page 29 for details on the encoding of the two- and three-byte extended escape sequences.

1.3 Opcode

The *opcode* is a single byte that specifies the basic operation of an instruction. In some cases, it also specifies the operands for the instruction. Every instruction requires an opcode. The correspondence between the binary value of the opcode and the operation it represents is defined by a table called an *opcode map*. As discussed in the previous sections, the legacy prefixes 66h, F2h, and F3h and other fields within the instruction encoding may be used to modify the operation encoded by the opcode.

The affect of the presence of a 66h, F2h, or F3h prefix on the operation performed by the opcode is represented in the opcode map by additional rows in the table indexed by the applicable prefix. The 3-bit reg and r/m fields of the ModRM byte (“ModRM Byte Format” on page 17 and “SIB Byte Format” on page 18) are used as well in the encoding of certain instructions. This is represented in the opcode maps via instruction group tables that detail the modifications represented via the extra encoding bits. See Section A.1, “Opcode Maps” of Appendix A for examples.

Even though each instruction has a unique opcode map and opcode, assemblers often support multiple alternate mnemonics for the same instruction to improve the readability of assembly language code.

The 64-bit floating-point 3DNow! instructions utilize the two-byte escape sequence 0Fh, 0Fh to select the 3DNow! opcode map. For these instructions the opcode is encoded in the immediate field at the end of the instruction encoding.

For details on how the opcode byte encodes the basic operation for specifc instructions, see Section A.1, “Opcode Maps” of Appendix A

1.4 ModRM and SIB Bytes

The ModRM byte is optional depending on the instruction. When present, it follows the opcode and is used to specify:

- two register-based operands, or
- one register-based operand and a second memory-based operand and an addressing mode.

In the encoding of some instructions, fields within the ModRM byte are repurposed to provide additional opcode bits used to define the instruction’s function.

The ModRM byte is partitioned into three fields—*mod*, *reg*, and *r/m*. Normally the reg field specifies a register-based operand and the mod and r/m fields used together specify a second operand that is either register-based or memory-based. The addressing mode is also specified when the operand is memory-based.

In 64-bit mode, the REX.R and REX.B bits augment the reg and r/m fields respectively allowing the specification of twice the number of registers.

1.4.1 ModRM Byte Format

Figure 1-4 below shows the format of a ModRM byte.

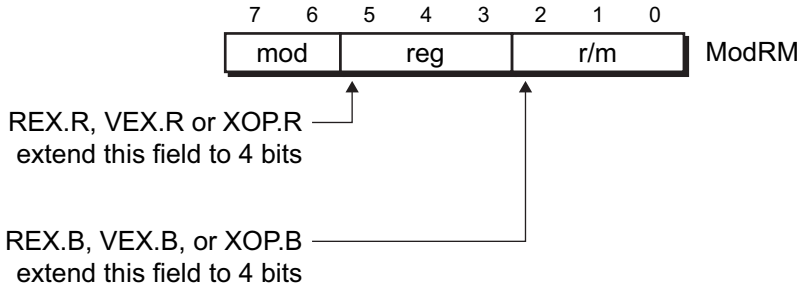


Figure 1-4. ModRM-Byte Format

Depending on the addressing mode, the SIB byte may appear after the ModRM byte. SIB is used in the specification of various forms of indexed register-indirect addressing. See the following section for details.

ModRM.mod (Bits[7:6]). The mod field is used with the r/m field to specify the addressing mode for an operand. ModRM.mod = 11b specifies the register-direct addressing mode. In the register-direct mode, the operand is held in the specified register. ModRM.mod values less than 11b specify register-indirect addressing modes. In register-indirect addressing modes, values held in registers along with an optional displacement specified in the instruction encoding are used to calculate the address of a memory-based operand. Other encodings of the 5 bits {mod, r/m} are discussed below.

ModRM.reg (Bits[5:3]). The reg field is used to specify a register-based operand, although for some instructions, this field is used to extend the operation encoding. The encodings for this field are shown in Table 1-10 below.

ModRM.r/m (Bits[2:0]). As stated above, the r/m field is used in combination with the mod field to encode 32 different operand specifications (See Table 1-14 on page 21). The encodings for this field are shown in Table 1-10 below.

Table 1-10. ModRM.reg and .r/m Field Encodings

Encoded value (binary)	ModRM.reg[1]	ModRM.r/m (mod = 11b)[1]	ModRM.r/m (mod ≠ 11b)[2]
000	rAX, MMX0, XMM0, YMM0	rAX, MMX0, XMM0, YMM0	[rAX]
001	rCX, MMX1, XMM1, YMM1	rCX, MMX1, XMM1, YMM1	[rCX]
010	rDX, MMX2, XMM2, YMM2	rDX, MMX2, XMM2, YMM2	[rDX]
011	rBX, MMX3, XMM3, YMM3	rBX, MMX3, XMM3, YMM3	[rBX]
100	AH, rSP, MMX4, XMM4, YMM4	AH, rSP, MMX4, XMM4, YMM4	SIB[3]
101	CH, rBP, MMX5, XMM5, YMM5	CH, rBP, MMX5, XMM5, YMM5	[rBP][4]
110	DH, rSI, MMX6, XMM6, YMM6	DH, rSI, MMX6, XMM6, YMM6	[rSI]
111	BH, rDI, MMX7, XMM7, YMM7	BH, rDI, MMX7, XMM7, YMM7	[rDI]

Notes:

1. *Specific register used is instruction-dependent.*
2. *mod = 01 and mod = 10 include an offset specified by the instruction displacement field. The notation [*] signifies that the specified register holds the address of the operand.*
3. *Indexed register-indirect addressing. SIB byte follows ModRM byte. See following section for SIB encoding.*
4. *For mod = 00b , r/m = 101b signifies absolute (displacement-only) addressing in 32-bit mode or RIP-relative addressing in 64-bit mode, where the rBP register is not used. For mod = [01b, 10b], r/m = 101b specifies the base + offset addressing mode with [rBP] as the base.*

Similar to the reg field, r/m is used in some instructions to extend the operation encoding.

1.4.2 SIB Byte Format

The SIB byte has three fields—*scale*, *index*, and *base*—that define the scale factor, index-register number, and base-register number for the 32-bit and 64-bit indexed register-indirect addressing modes.

The basic formula for computing the effective address of a memory-based operand using the indexed register-indirect address modes is:

```
effective_address = scale * index + base + offset
```

Specific variants of this addressing mode set one or more elements of the sum to zero.

Figure 1-5 below shows the format of the SIB byte.

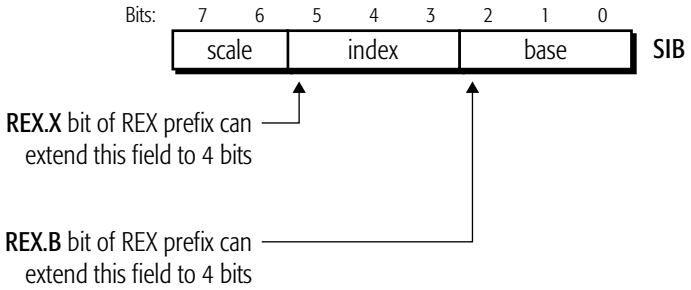


Figure 1-5. SIB Byte Format

SIB.scale (Bits[7:6]). The scale field is used to specify the scale factor used in computing the scale*index portion of the effective address. In normal usage scale represents the size of data elements in an array expressed in number of bytes. SIB.scale is encoded as shown in Table 1-11 below.

Table 1-11. SIB.scale Field Encodings

Encoded value (binary)	scale factor
00	1
01	2
10	4
11	8

SIB.index (Bits[5:3]). The index field is used to specify the register containing the index portion of the indexed register-indirect effective address. SIB.index is encoded as shown in Table 1-12 below.

SIB.base (Bits[2:0]). The base field is used to specify the register containing the base address portion of the indexed register-indirect effective address. SIB.base is encoded as shown in Table 1-12 below.

Table 1-12. SIB.index and .base Field Encodings

Encoded value (binary)	SIB.index	SIB.base
000	[rAX]	[rAX]
001	[rCX]	[rCX]
010	[rDX]	[rDX]
011	[rBX]	[rBX]
100	(none)[1]	[rSP]
101	[rBP]	[rBP], (none)[2]
110	[rSI]	DH, [rSI]
111	[rDI]	BH, [rDI]

Notes:

*1. Register specification is null. The scale*index portion of the indexed register-indirect effective address is set to 0.*

2. If ModRM.mod = 00b, the register specification is null. The base portion of the indexed register-indirect effective address is set to 0. Otherwise, base encodes the rBP register as the source of the base address used in the effective address calculation.

Table 1-13. SIB.base encodings for ModRM.r/m = 100b

	SIB base Field							
mod	000	001	010	011	100	101	110	111
00	[rAX]	[rCX]	[rDX]	[rBX]	[rSP]	disp32	[rSI]	[rDI]
01						[rBP]+disp8		
10						[rBP]+disp32		
11	(not applicable)							

More discussion of operand addressing follows in the next two sections.

1.4.3 Operand Addressing in Legacy 32-bit and Compatibility Modes

The mod and r/m fields of the ModRM byte provide a total of five bits used to encode 32 operand specification and memory addressing modes. Table 1-14 below shows these encodings.

Table 1-14. Operand Addressing Using ModRM and SIB Bytes

ModRM.mod	ModRM.r/m	Register / Effective Address
00	000	[rAX]
	001	[rCX]
	010	[rDX]
	011	[rBX]
	100	SIB[1]
	101	*disp32*
	110	[rSI]
	111	[rDI]
01	000	[rAX]+*disp8*
	001	[rCX]+*disp8*
	010	[rDX]+*disp8*
	011	[rBX]+*disp8*
	100	SIB+*disp8*[2]
	101	[rBP]+*disp8*
	110	[rSI]+*disp8*
	111	[rDI]+*disp8*
10	000	[rAX]+*disp32*
	001	[rCX]+*disp32*
	010	[rDX]+*disp32*
	011	[rBX]+*disp32*
	100	SIB+*disp32*[3]
	101	[rBP]+*disp32*
	110	[rSI]+*disp32*
	111	[rDI]+*disp32*

Notes:

*0. In the following notes, scaled_index = SIB.index * (1 << SIB.scale).*

1. SIB byte follows ModRM byte. Effective address is calculated using scaled_index+base. When SIB.base = 101b, addressing mode depends on ModRM.mod. See Table 1-13 above.

2. SIB byte follows ModRM byte. Effective address is calculated using scaled_index+base+8-bit_offset. One-byte Displacement field provides the offset.

3. SIB byte follows ModRM byte. Effective address is calculated using scaled_index+base+32-bit_offset. Four-byte Displacement field provides the offset.

Table 1-14. Operand Addressing Using ModRM and SIB Bytes (continued)

ModRM.mod	ModRM.r/m	Register / Effective Address
11	000	AL/rAX/MMX0/XMM0/YMM0
	001	CL/rCX/MMX1/XMM1/YMM1
	010	DL/rDX/MMX2/XMM2/YMM2
	011	BL/rBX/MMX3/XMM3/YMM3
	100	AH/SPL/rSP/MMX4/XMM4/YMM4
	101	CH/BPL/rBP/MMX5/XMM5/YMM5
	110	DH/SIL/rSI/MMX6/XMM6/YMM6
	111	BH/DIL/rDI/MMX7/XMM7/YMM7

Notes:

*0. In the following notes, scaled_index = SIB.index * (1 << SIB.scale).*

1. SIB byte follows ModRM byte. Effective address is calculated using scaled_index+base. When SIB.base = 101b, addressing mode depends on ModRM.mod. See Table 1-13 above.

2. SIB byte follows ModRM byte. Effective address is calculated using scaled_index+base+8-bit_offset. One-byte Displacement field provides the offset.

3. SIB byte follows ModRM byte. Effective address is calculated using scaled_index+base+32-bit_offset. Four-byte Displacement field provides the offset.

Note that the addressing mode mod = 11b is a register-direct mode, that is, the operand is contained in the specified register, while the modes mod = [00b:10b] specify different addressing modes for a memory-based operand.

For mod = 11b, the register containing the operand is specified by the r/m field. For the other modes (mod = [00b:10b]), the mod and r/m fields are combined to specify the addressing mode for the memory-based operand. Most are register-indirect addressing modes meaning that the address of the memory-based operand is contained in the register specified by r/m. For these register-indirect modes, mod = 01b and mod = 10b include an offset encoded in the displacement field of the instruction.

The encodings {mod ≠ 11b, r/m = 100b} specify the *indexed register-indirect* addressing mode in which the target address is computed using a combination of values stored in registers and a scale factor encoded directly in the SIB byte. For these addressing modes the effective address is given by the formula:

```
effective_address = scale * index + base + offset
```

Scale is encoded in SIB.scale field. Index is contained in the register specified by SIB.index field and base is contained in the register specified by SIB.base field. Offset is encoded in the displacement field of the instruction using either one or four bytes.

If {mod, r/m} = 00100b, the offset portion of the formula is set to 0. For {mod, r/m} = 01100b and {mod, r/m} =10100b, offset is encoded in the one- or 4-byte displacement field of the instruction.

Finally, the encoding {mod, r/m} = 00101b specifies an absolute addressing mode. In this mode, the address is provided directly in the instruction encoding using a 4-byte displacement field. In 64-bit mode this addressing mode is changed to RIP-relative (see “RIP-Relative Addressing” on page 24).

1.4.4 Operand Addressing in 64-bit Mode

AMD64 architecture doubles the number of GPRs and increases their width to 64-bits. It also doubles the number of YMM/XMM registers. In order to support the specification of register operands contained in the eight additional GPRs or YMM/XMM registers and to make the additional GPRs available to hold addresses to be used in the addressing modes, the REX prefix provides the R, X, and B bit fields to extend the reg, r/m, index, and base fields of the ModRM and SIB bytes in the various operand addressing modes to four bits. A fourth REX bit field (W) allows instruction encodings to specify a 64-bit operand size.

Table 1-15 below and the sections that follow describe each of these bit fields.

Table 1-15. REX Prefix-Byte Fields

Mnemonic	Bit Position(s)	Definition
—	7:4	0100 (4h)
REX.W	3	0 = Default operand size 1 = 64-bit operand size
REX.R	2	1-bit (msb) extension of the ModRM *reg* field[1], permitting access to 16 registers.
REX.X	1	1-bit (msb) extension of the SIB *index* field[1], permitting access to 16 registers.
REX.B	0	1-bit (msb) extension of the ModRM *r/m* field[1], SIB *base* field[1], or opcode *reg* field, permitting access to 16 registers.

Notes:
1. *For a description of the ModRM and SIB bytes, see "ModRM and SIB Bytes" on page 17.*

REX.W: Operand width (Bit 3). Setting the REX.W bit to 1 specifies a 64-bit operand size. Like the existing 66h operand-size override prefix, the REX 64-bit operand-size override has no effect on byte operations. For non-byte operations, the REX operand-size override takes precedence over the 66h prefix. If a 66h prefix is used together with a REX prefix that has the W bit set to 1, the 66h prefix is ignored. However, if a 66h prefix is used together with a REX prefix that has the W bit cleared to 0, the 66h prefix is not ignored and the operand size becomes 16 bits.

REX.R: Register field extension (Bit 2). The REX.R bit adds a 1-bit extension (in the most significant bit position) to the ModRM.reg field when that field encodes a GPR, YMM/XMM, control, or debug register. REX.R does not modify ModRM.reg when that field specifies other registers or is used to extend the opcode. REX.R is ignored in such cases.

REX.X: Index field extension (Bit 1). The REX.X bit adds a 1-bit (msb) extension to the SIB.index field. See "ModRM and SIB Bytes" on page 17.

REX.B: Base field extension (Bit 0). The REX.B bit adds a 1-bit (msb) extension to either the ModRM.r/m field to specify a GPR or XMM register, or to the SIB.base field to specify a GPR. (See Table 2-2 on page 56 for more about the B bit.)

1.5 Displacement Bytes

A *displacement* (also called an *offset)* is a signed value that is added to the base of a code segment (absolute addressing) or to an instruction pointer (relative addressing), depending on the addressing mode. The size of a displacement is 1, 2, or 4 bytes. If an addressing mode requires a displacement, the bytes (1, 2, or 4) for the displacement follow the opcode, ModRM, or SIB byte (whichever comes last) in the instruction encoding.

In 64-bit mode, the same ModRM and SIB encodings are used to specify displacement sizes as those used in legacy and compatibility modes. However, the displacement is sign-extended to 64 bits during effective-address calculations. Also, in 64-bit mode, support is provided for some 64-bit displacement and immediate forms of the MOV instruction. See “Immediate Operand Size” in Volume 1 for more information on this.

1.6 Immediate Bytes

An *immediate* is a value—typically an operand value—encoded directly into the instruction. Depending on the opcode and the operating mode, the size of an immediate operand can be 1, 2, 4, or 8 bytes. 64-bit immediates are allowed in 64-bit mode on MOV instructions that load GPRs, otherwise they are limited to 4 bytes. See “Immediate Operand Size” in Volume 1 for more information.

If an instruction takes an immediate operand, the bytes (1, 2, 4, or 8) for the immediate follow the opcode, ModRM, SIB, or displacement bytes (whichever come last) in the instruction encoding. Some 128-bit media instructions use the immediate byte as a condition code.

1.7 RIP-Relative Addressing

In 64-bit mode, addressing relative to the contents of the 64-bit instruction pointer (program counter)—called RIP-relative addressing or PC-relative addressing—is implemented for certain instructions. In such cases, the effective address is formed by adding the displacement to the 64-bit RIP of the next instruction.

In the legacy x86 architecture, addressing relative to the instruction pointer is available only in control-transfer instructions. In the 64-bit mode, any instruction that uses ModRM addressing can use RIP-relative addressing. This feature is particularly useful for addressing data in position-independent code and for code that addresses global data.

Without RIP-relative addressing, ModRM instructions address memory relative to zero. With RIP-relative addressing, ModRM instructions can address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of ±2 Gbytes from the RIP.

Programs usually have many references to data, especially global data, that are not register-based. To load such a program, the loader typically selects a location for the program in memory and then adjusts program references to global data based on the load location. RIP-relative addressing of data makes this adjustment unnecessary.

1.7.1 Encoding

Table 1-16 shows the ModRM and SIB encodings for RIP-relative addressing. Redundant forms of 32-bit displacement-only addressing exist in the current ModRM and SIB encodings. There is one ModRM encoding with several SIB encodings. RIP-relative addressing is encoded using one of the redundant forms. In 64-bit mode, the ModRM *disp32* (32-bit displacement) encoding ({mod,r/m} = 00101b) is redefined to be *RIP + disp32* rather than displacement-only.

Table 1-16. Encoding for RIP-Relative Addressing

ModRM	SIB	Legacy and Compatibility Modes	64-bit Mode	Additional 64-bit Implications
• mod = 00 • r/m = 101	not present	disp32	RIP + disp32	Zero-based (normal) displacement addressing must use SIB form (see next row).
• mod = 00 • r/m = 100[1]	• base = 101[2] • index = 100[3] • scale = xx	disp32	Same as Legacy	None

Notes:
1. Encodes the indexed register-indirect addressing mode with 32-bit offset.
2. Base register specification is null (base portion of effective address calculation is set to 0)
*3. index register specification is null (scale*index portion of effective address calculation is set to 0)*

1.7.2 REX Prefix and RIP-Relative Addressing

ModRM encoding for RIP-relative addressing does not depend on a REX prefix. In particular, the *r/m* encoding of 101, used to select RIP-relative addressing, is not affected by the REX prefix. For example, selecting R13 (REX.B = 1, r/m = 101) with mod = 00 still results in RIP-relative addressing.

The four-bit r/m field of ModRM is not fully decoded. Therefore, in order to address R13 with no displacement, software must encode it as R13 + 0 using a one-byte displacement of zero.

1.7.3 Address-Size Prefix and RIP-Relative Addressing

RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. Conversely, use of the address-size prefix (“Address-Size Override Prefix” on page 9) does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits, like any other addressing mode.

1.8 Encoding Considerations Using REX

Figure 1-6 on page 28 shows four examples of how the R, X, and B bits of the REX prefix are concatenated with fields from the ModRM byte, SIB byte, and opcode to specify register and memory addressing.

1.8.1 Byte-Register Addressing

In the legacy architecture, the byte registers (AH, AL, BH, BL, CH, CL, DH, and DL, shown in Figure 2-2 on page 38) are encoded in the ModRM reg or r/m field or in the opcode *reg* field as registers 0 through 7. The REX prefix provides an additional byte-register addressing capability that makes the least-significant byte of any GPR available for byte operations (Figure 2-3 on page 39). This provides a uniform set of byte, word, doubleword, and quadword registers better suited for register allocation by compilers.

1.8.2 Special Encodings for Registers

Readers who need to know the details of instruction encodings should be aware that certain combinations of the ModRM and SIB fields have special meaning for register encodings. For some of these combinations, the instruction fields expanded by the REX prefix are not decoded (treated as don’t cares), thereby creating aliases of these encodings in the extended registers. Table 1-17 on page 27 describes how each of these cases behaves.

Table 1-17. Special REX Encodings for Registers

ModRM and SIB Encodings[2]	Meaning in Legacy and Compatibility Modes	Implications in Legacy and Compatibility Modes	Additional REX Implications
ModRM Byte: • mod ≠ 11 • r/m[1] = 100 (ESP)	SIB byte is present.	SIB byte is required for ESP-based addressing.	REX prefix adds a fourth bit (b), which is decoded and modifies the base register in the SIB byte. Therefore, the SIB byte is also required for R12-based addressing.
ModRM Byte: • mod = 00 • r/m[1] = x101 (EBP)	Base register is not used.	Using EBP without a displacement must be done by setting mod = 01 with a displacement of 0 (with or without an index register).	REX prefix adds a fourth bit (x), which is not decoded (don't care). Therefore, using RBP or R13 without a displacement must be done via mod = 01 with a displacement of 0.
SIB Byte: • index[1] = x100 (ESP)	Index register is not used.	ESP cannot be used as an index register.	REX prefix adds a fourth bit (x), which is decoded. Therefore, there are no additional implications. The expanded index field is used to distinguish RSP from R12, allowing R12 to be used as an index.
SIB Byte: • base = b101 (EBP) • ModRM.mod = 00	Base register is not used if ModRM.mod = 00.	Base register depends on mod encoding. Using EBP with a scaled index and without a displacement must be done by setting mod = 01 with a displacement of 0.	REX prefix adds a fourth bit (b), which is not decoded (don't care). Therefore, using RBP or R13 without a displacement must be done via mod = 01 with a displacement of 0 (with or without an index register).

Notes:

1. *The REX-prefix bit is shown in the fourth (most-significant) bit position of the encodings for the ModRM r/m, SIB index, and SIB base fields. The lower-case "x" for ModRM r/m (rather than the upper-case "B" shown in Figure 1-6 on page 28) indicates that the REX-prefix bit is not decoded (don't care).*
2. *For a description of the ModRM and SIB bytes, see "ModRM and SIB Bytes" on page 17.*

Examples of Operand Addressing Extension Using REX

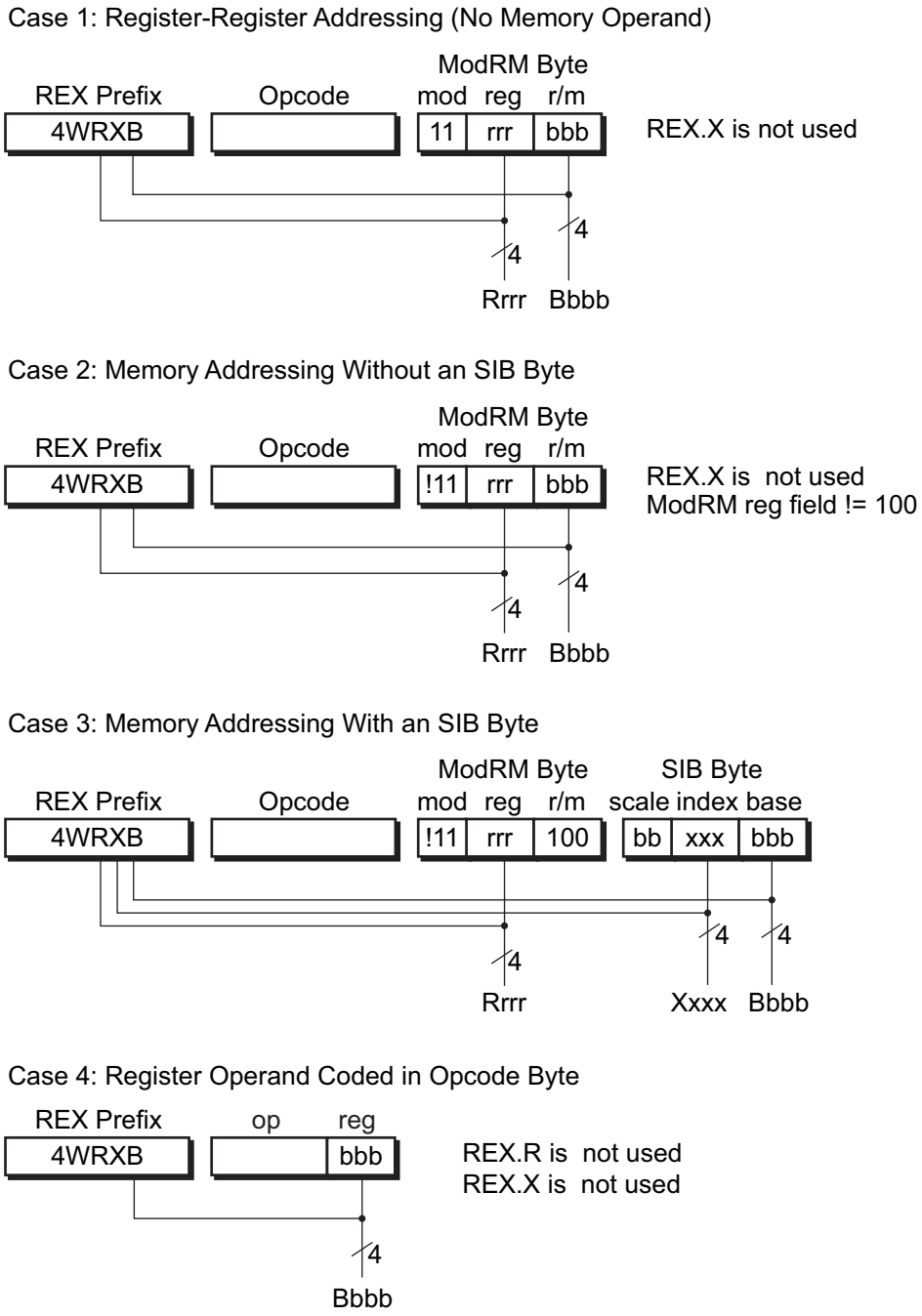


Figure 1-6. Encoding Examples Using REX R, X, and B Bits

1.9 Encoding Using the VEX and XOP Prefixes

An extended escape sequence is introduced by an encoding escape prefix which establishes the context and the format of the bytes that follow. The currently defined prefixes fall in two classes: the XOP and the VEX prefixes (of which there are two). The XOP prefix and the VEX C4h prefix introduce a three byte sequence with identical syntax, while the VEX C5h prefix introduces a two-byte escape sequence with a different syntax.

These escape sequences supply fields used to extend operand specification as well as provide for the selection of alternate opcode maps. Encodings support up to two additional operands and the addressing of the extended (beyond 7) registers. The specification of two of the operands is accomplished using the legacy ModRM and optional SIB bytes with the reg, r/m, index, and base fields extended by one bit in a manner analogous to the REX prefix.

The encoding of the extended SSE instructions utilize extended escape sequences. XOP instructions use three-byte escape sequences introduced by the XOP prefix. The AVX, FMA, FMA4, and CLMUL instruction subsets use three-byte or two-byte escape sequences introduced by the VEX prefixes.

1.9.1 Three-Byte Escape Sequences

All the extended instructions can be encoded using a three-byte escape sequence, but certain VEX-encoded instructions that comply with the constraints described below in Section 1.9.2, “Two-Byte Escape Sequence” can also utilize a two-byte escape sequence. Figure 1-7 below shows the format of the three-byte escape sequence which is common to the XOP and VEX-based encodings.

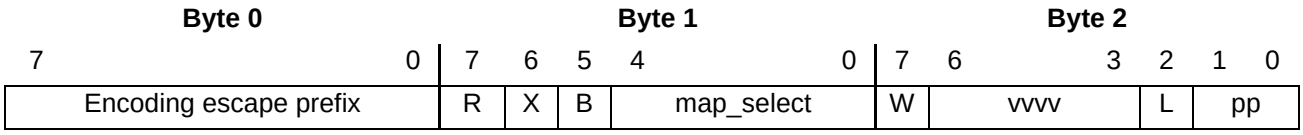


Figure 1-7. VEX/XOP Three-byte Escape Sequence Format

Byte	Bit	Mnemonic	Description
0	[7:0]	VEX, XOP	Value specific to the extended instruction set
1	[7]	R	Inverted one-bit extension of ModRM *reg* field
	[6]	X	Inverted one-bit extension of SIB *index* field
	[5]	B	Inverted one-bit extension, *r/m* field or SIB *base* field
	[4:0]	map_select	Opcode map select

Byte	Bit	Mnemonic	Description
2	[7]	W	Default operand size override for a general purpose register to 64-bit size in 64-bit mode; operand configuration specifier for certain YMM/XMM-based operations.
	[6:3]	vvvv	Source or destination register selector, in ones' complement format
	[2]	L	Vector length specifier
	[1:0]	pp	Implied 66, F2, or F3 opcode extension

Table 1-18. Three-byte Escape Sequence Field Definitions

Byte 0 (VEX/XOP Prefix)

Byte 0 is the encoding escape prefix byte which introduces the encoding escape sequence and establishes the context for the bytes that follow. The VEX and XOP prefixes have the following encodings:

- VEX prefix is encoded as C4h
- XOP prefix is encoded as 8Fh

Byte 1

VEX/XOP.R (Bit 7). The bit-inverted equivalent of the REX.R bit. A one-bit extension of the ModRM.reg field in 64-bit mode, permitting access to 16 YMM/XMM and GPR registers. In 32-bit protected and compatibility modes, the value must be 1.

VEX/XOP.X (Bit 6). The bit-inverted equivalent of the REX.X bit. A one-bit extension of the SIB.index field in 64-bit mode, permitting access to 16 YMM/XMM and GPR registers. In 32-bit protected and compatibility modes, this value must be 1.

VEX/XOP.B (Bit 5). The bit-inverted equivalent of the REX.B bit, available only in the 3-byte prefix format. A one-bit extension of either the ModRM.r/m field, to specify a GPR or XMM register, or of the SIB base field, to specify a GPR. This permits access to all 16 GPR and YMM/XMM registers. In 32-bit protected and compatibility modes, this bit is ignored.

VEX/XOP.map_select (Bits [4:0]). The five-bit map_select field is used to select an alternate opcode map. The map select encoding spaces for VEX and XOP are disjoint. Table 1-19 below lists the encodings for VEX.map_select and Table 1-20 lists the encodings for XOP.map_select.

Table 1-19. VEX.map_select Encoding

Binary Value	Opcode Map	Analogous Legacy Opcode Map
00000	Reserved	–
00001	VEX opcode map 1	Secondary ("two-byte") opcode map

Table 1-19. VEX.map_select Encoding

Binary Value	Opcode Map	Analogous Legacy Opcode Map
00010	VEX opcode map 2	0F_38h (“three-byte”) opcode map
00011	VEX opcode map 3	0F_3Ah (“three-byte”) opcode map
00100 – 11111	Reserved	–

Table 1-20. XOP.map_select Encoding

Binary Value	Opcode Map
00000 – 00111	Reserved
01000	XOP opcode map 8
01001	XOP opcode map 9
01010	XOP opcode map 10 (Ah)
01011 – 11111	Reserved

AVX instructions are encoded using the VEX opcode maps 1–3. The AVX instruction set includes instructions that provide operations similar to most legacy SSE instructions. For those AVX instructions that have an analogous legacy SSE instruction, the VEX opcode maps use the same binary opcode value and modifiers as the legacy version. The correspondence between the VEX opcode maps and the legacy opcode maps are shown in Table 1-19 above.

VEX opcode maps 1–3 are also used to encode the FMA4 and FMA instructions. In addition, not all legacy SSE instructions have AVX equivalents. Therefore, the VEX opcode maps are not the same as the legacy opcode maps.

The XOP opcode maps are unique to the XOP instructions. The XOP.map_select value is restricted to the range [08h:1Fh]. If the value of the XOP.map_select field is less than 8, the first two bytes of the three-byte XOP escape sequence are interpreted as a form of the POP instruction.

Both legacy and extended opcode maps are covered in detail in Appendix A.

Byte 2

VEX/XOP.W (Bit 7). Function is instruction-specific. The bit is often used to configure source operand order.

VEX/XOP.vvvv (Bits [6:3]). Used to specify an additional operand for three and four operand instructions. Encodes an XMM or YMM register in inverted ones’ complement form, as shown in Table 1-21.

Table 1-21. VEX/XOP.vvvv Encoding

Binary Value	Register	Binary Value	Register
0000	XMM15/YMM15	1000	XMM07/YMM07
0001	XMM14/YMM14	1001	XMM06/YMM06
0010	XMM13/YMM13	1010	XMM05/YMM05
0011	XMM12/YMM12	1011	XMM04/YMM04
0100	XMM11/YMM11	1100	XMM03/YMM03
0101	XMM10/YMM10	1101	XMM02/YMM02
0110	XMM09/YMM09	1110	XMM01/YMM01
0111	XMM08/YMM08	1111	XMM00/YMM00

Values 0000h to 0111h are not valid in 32-bit modes. vvvv is typically used to encode the first source operand, but for the VPSLLDQ, VPSRLDQ, VPSRLW, VPSRLD, VPSRLQ, VPSRAW, VPSRAD, VPSLLW, VPSLLD, and VPSLLQ shift instructions, the field specifies the destination register.

VEX/XOP.L (Bit 2). L = 0 specifies 128-bit vector length (XMM registers/128-bit memory locations). L=1 specifies 256-bit vector length (YMM registers/256-bit memory locations). For SSE or XOP instructions with scalar operands, the L bit is ignored. Some vector SSE instructions support only the 128 bit vector size. For these instructions, L is cleared to 0.

VEX/XOP.pp (Bits [1:0]). Specifies an implied 66h, F2h, or F3h opcode extension which is used in a way analogous to the legacy instruction encodings to extend the opcode encoding space. The correspondence between the encoding of the VEX/XOP.pp field and its function as an opcode modifier is shown in Table 1-22. The legacy prefixes 66h, F2h, and F3h are not allowed in the encoding of extended instructions.

Table 1-22. VEX/XOP.pp Encoding

Binary Value	Implied Prefix
00	None
01	66h
10	F3h
11	F2h

1.9.2 Two-Byte Escape Sequence

All VEX-encoded instructions can be encoded using the three-byte escape sequence, but certain instructions can also be encoded utilizing a more compact, two-byte VEX escape sequence. The format of the two-byte escape sequence is shown in Figure 1-8 below.

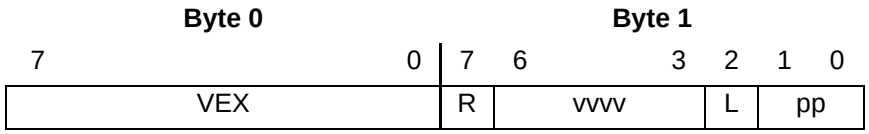


Figure 1-8. VEX Two-byte Escape Sequence Format

Prefix Byte	Bit	Mnemonic	Description
0	[7:0]	VEX	VEX 2-byte encoding escape prefix
1	[7]	R	Inverted one-bit extension of ModRM.reg field
	[6:3]	vvvv	Source or destination register selector, in ones' complement format.
	[2]	L	Vector length specifier
	[1:0]	pp	Implied 66, F2, or F3 opcode extension.

Table 1-23. VEX Two-byte Escape Sequence Field Definitions

Byte 0 (VEX Prefix)

The VEX prefix for the two-byte escape sequence is encoded as C5h.

Byte 1

Note that the bit 7 of this byte is used to encode VEX.R instead of VEX.W as in the three-byte escape sequence form. The R, vvvv, L, and pp fields are defined as in the three-byte escape sequence.

When the two-byte escape sequence is used, specific fields from the three-byte format take on fixed values as shown in Table 1-24 below.

Table 1-24. Fixed Field Values for VEX 2-Byte Format

VEX Field	Value
X	1
B	1
W	0
map_select	00001b

Although they may be encoded using the VEX three-byte escape sequence, all instructions that conform with the constraints listed in Table 1-24 may be encoded using the two-byte escape sequence. Note that the implied value of map_select is 00001b, which means that only instructions included in the VEX opcode map 1 may be encoded using this format.

VEX-encoded instructions that use the other defined values of map_select (00010b and 00011b) cannot be encoded using this a two-byte escape sequence format. Note that the VEX.pp field value is

explicitly encoded in this form and can be used to specify any of the implied legacy prefixes as defined in Table 1-22.

2 Instruction Overview

2.1 Instruction Groups

For easier reference, the instruction descriptions are divided into five groups based on usage. The following sections describe the function, mnemonic syntax, opcodes, affected flags, and possible exceptions generated by all instructions in the AMD64 architecture:

- *Chapter 3, "General-Purpose Instruction Reference"*—The general-purpose instructions are used in basic software execution. Most of these load, store, or operate on data in the general-purpose registers (GPRs), in memory, or in both. Other instructions are used to alter sequential program flow by branching to other locations within the program or to entirely different programs.
- *Chapter 4, "System Instruction Reference"*—The system instructions establish the processor operating mode, access processor resources, handle program and system errors, and manage memory.
- *"SSE Instruction Reference" in Volume 4*—The Streaming SIMD Extensions (SSE) instructions load, store, or operate on data located in the YMM/XMM registers. These instructions define both vector and scalar operations on floating-point and integer data types. They include the SSE and SSE2 instructions that operate on the YMM/XMM registers. Some of these instructions convert source operands in YMM/XMM registers to destination operands in GPR, MMX, or x87 registers or otherwise affect YMM/XMM state.
- *"64-Bit Media Instruction Reference" in Volume 5*—The 64-bit media instructions load, store, or operate on data located in the 64-bit MMX registers. These instructions define both vector and scalar operations on integer and floating-point data types. They include the legacy MMX™ instructions, the 3DNow!™ instructions, and the AMD extensions to the MMX and 3DNow! instruction sets. Some of these instructions convert source operands in MMX registers to destination operands in GPR, YMM/XMM, or x87 registers or otherwise affect MMX state.
- *"x87 Floating-Point Instruction Reference" in Volume 5*—The x87 instructions are used in legacy floating-point applications. Most of these instructions load, store, or operate on data located in the x87 ST(0)–ST(7) stack registers (the FPR0–FPR7 physical registers). The remaining instructions within this category are used to manage the x87 floating-point environment.

The description of each instruction covers its behavior in all operating modes, including legacy mode (real, virtual-8086, and protected modes) and long mode (compatibility and 64-bit modes). Details of certain kinds of complex behavior—such as control-flow changes in CALL, INT, or FXSAVE instructions—have cross-references in the instruction-detail pages to detailed descriptions in volumes 1 and 2.

Two instructions—CMPSD and MOVSD—use the same mnemonic for different instructions. Assemblers can distinguish them on the basis of the number and type of operands with which they are used.

2.2 Reference-Page Format

Figure 2-1 on page 37 shows the format of an instruction-detail page. The instruction mnemonic is shown in bold at the top-left, along with its name. In this example, *POPFD* is the mnemonic and *POP to EFLAGS Doubleword* is the name. Next, there is a general description of the instruction's operation. Many descriptions have cross-references to more detail in other parts of the manual.

Beneath the general description, the mnemonic is shown again, together with the related opcode(s) and a description summary. Related instructions are listed below this, followed by a table showing the flags that the instruction can affect. Finally, each instruction has a summary of the possible exceptions that can occur when executing the instruction. The columns labeled "Real" and "Virtual-8086" apply only to execution in legacy mode. The column labeled "Protected" applies both to legacy mode and long mode, because long mode is a superset of legacy protected mode.

The 128-bit and 64-bit media instructions also have diagrams illustrating the operation. A few instructions have examples or pseudocode describing the action.

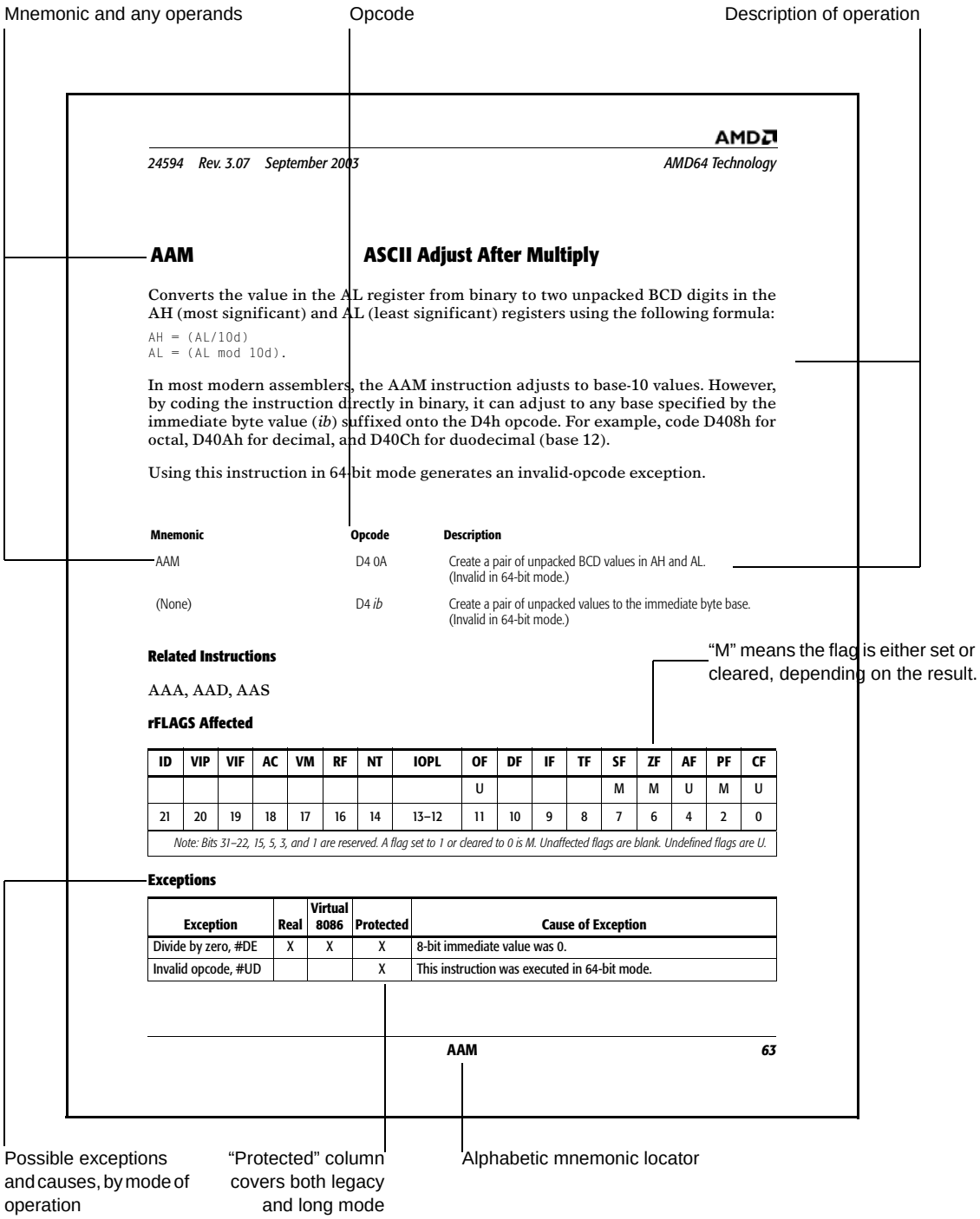


Figure 2-1. Format of Instruction-Detail Pages

2.3 Summary of Registers and Data Types

This section summarizes the registers available to software using the five instruction subsets described in “Instruction Groups” on page 35. For details on the organization and use of these registers, see their respective chapters in volumes 1 and 2.

2.3.1 General-Purpose Instructions

Registers. The size and number of general-purpose registers (GPRs) depends on the operating mode, as do the size of the flags and instruction-pointer registers. Figure 2-2 shows the registers available in legacy and compatibility modes.

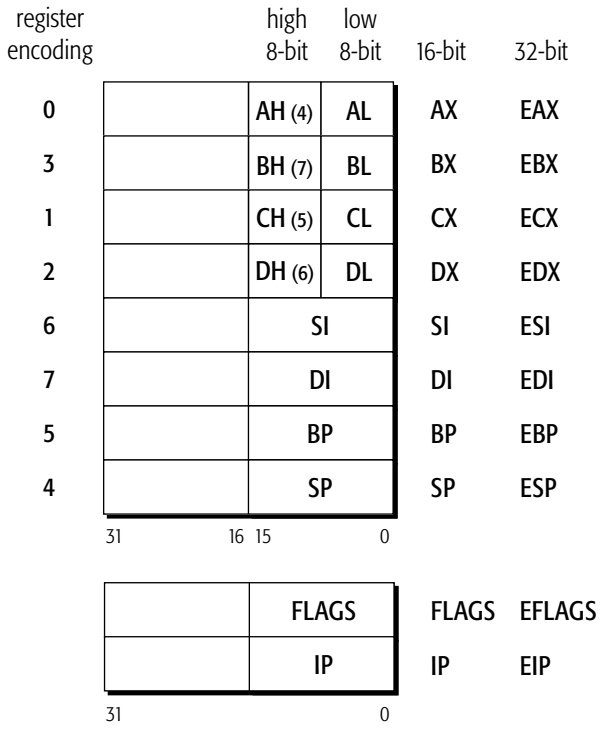


Figure 2-2. General Registers in Legacy and Compatibility Modes

Figure 2-3 on page 39 shows the registers accessible in 64-bit mode. Compared with legacy mode, registers become 64 bits wide, eight new data registers (R8–R15) are added and the low byte of all 16 GPRs is available for byte operations, and the four high-byte registers of legacy mode (AH, BH, CH, and DH) are not available if the REX prefix is used. The high 32 bits of doubleword operands are zero-extended to 64 bits, but the high bits of word and byte operands are not modified by operations in 64-

bit mode. The RFLAGS register is 64 bits wide, but the high 32 bits are reserved. They can be written with anything but they read as zeros (RAZ).

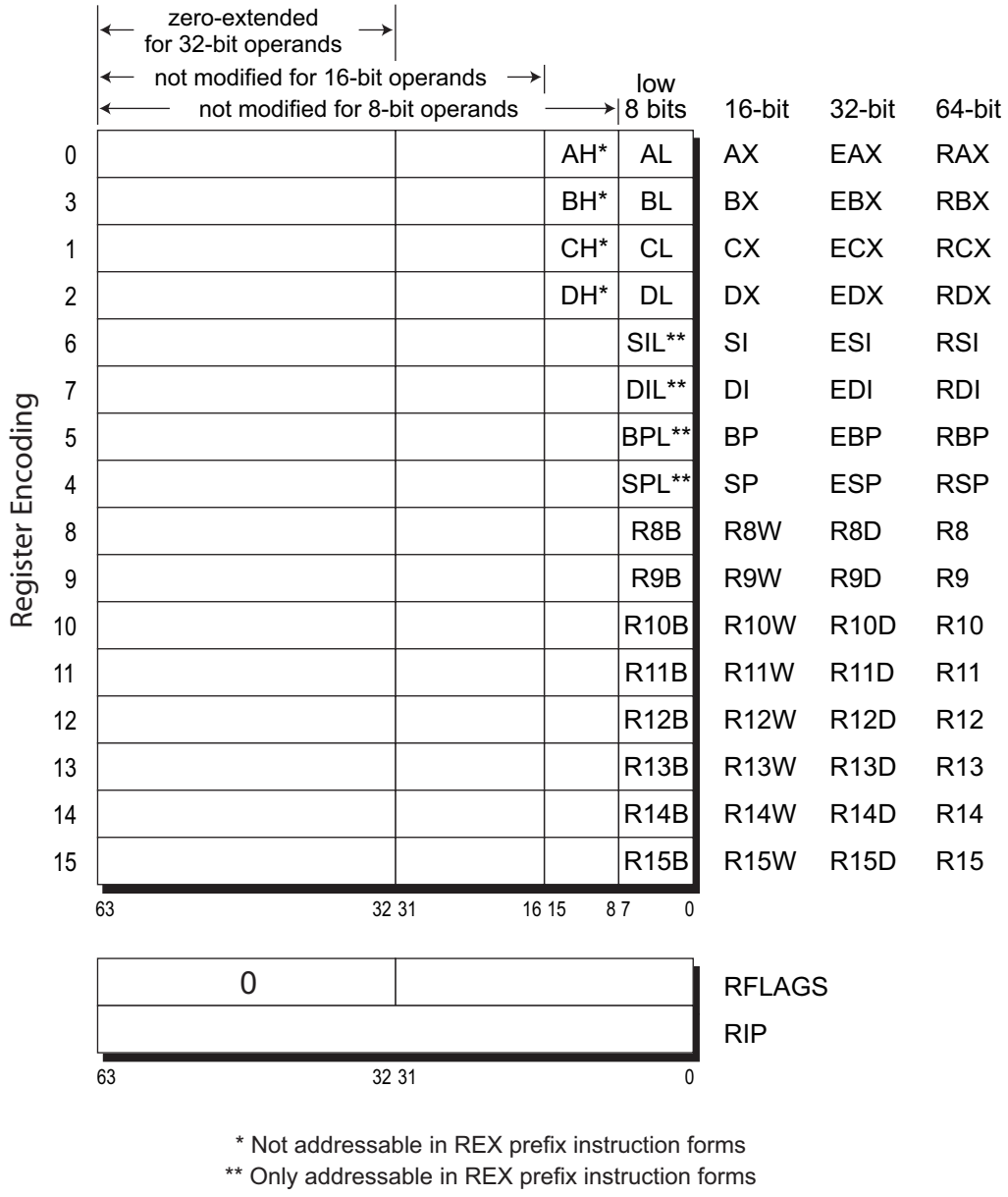


Figure 2-3. General Registers in 64-Bit Mode

For most instructions running in 64-bit mode, access to the extended GPRs requires a either a REX instruction modification prefix or extended encoding encoding using the VEX or XOP sequences (page 16).

Figure 2-4 shows the segment registers which, like the instruction pointer, are used by all instructions. In legacy and compatibility modes, all segments are accessible. In 64-bit mode, which uses the flat (non-segmented) memory model, only the CS, FS, and GS segments are recognized, whereas the contents of the DS, ES, and SS segment registers are ignored (the base for each of these segments is assumed to be zero, and neither their segment limit nor attributes are checked). For details, see "Segmented Virtual Memory" in Volume 2.

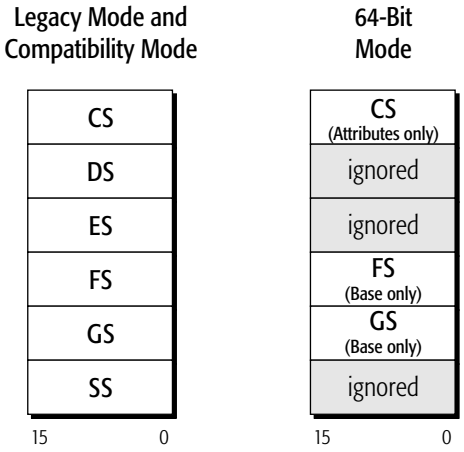


Figure 2-4. Segment Registers

Data Types. Figure 2-5 on page 41 shows the general-purpose data types. They are all scalar, integer data types. The 64-bit (quadword) data types are only available in 64-bit mode, and for most instructions they require a REX instruction prefix.

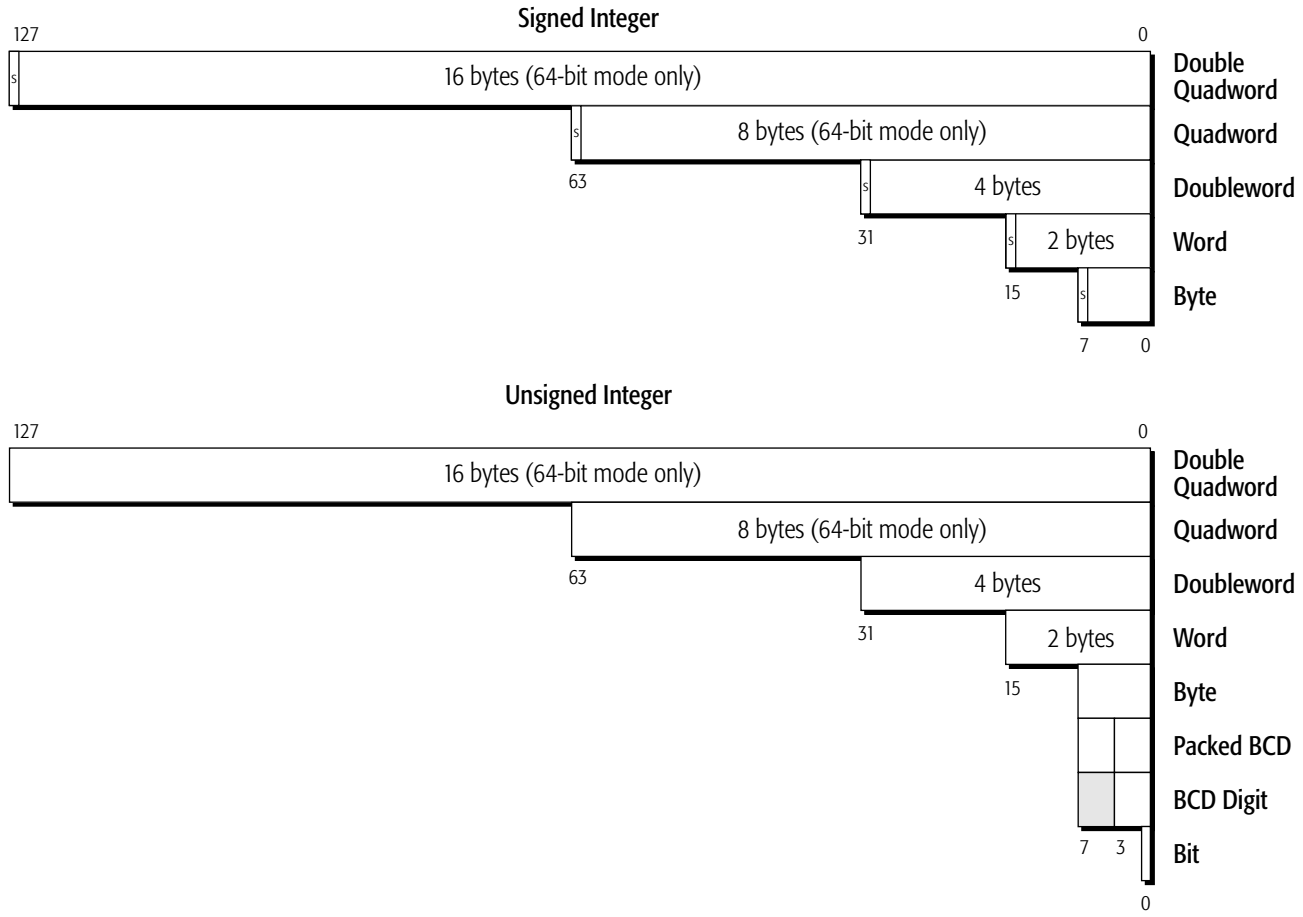


Figure 2-5. General-Purpose Data Types

2.3.2 System Instructions

Registers. The system instructions use several specialized registers shown in Figure 2-6 on page 42. System software uses these registers to, among other things, manage the processor's operating environment, define system resource characteristics, and monitor software execution. With the exception of the RFLAGS register, system registers can be read and written only from privileged software.

All system registers are 64 bits wide, except for the descriptor-table registers and the task register, which include 64-bit base-address fields and other fields.

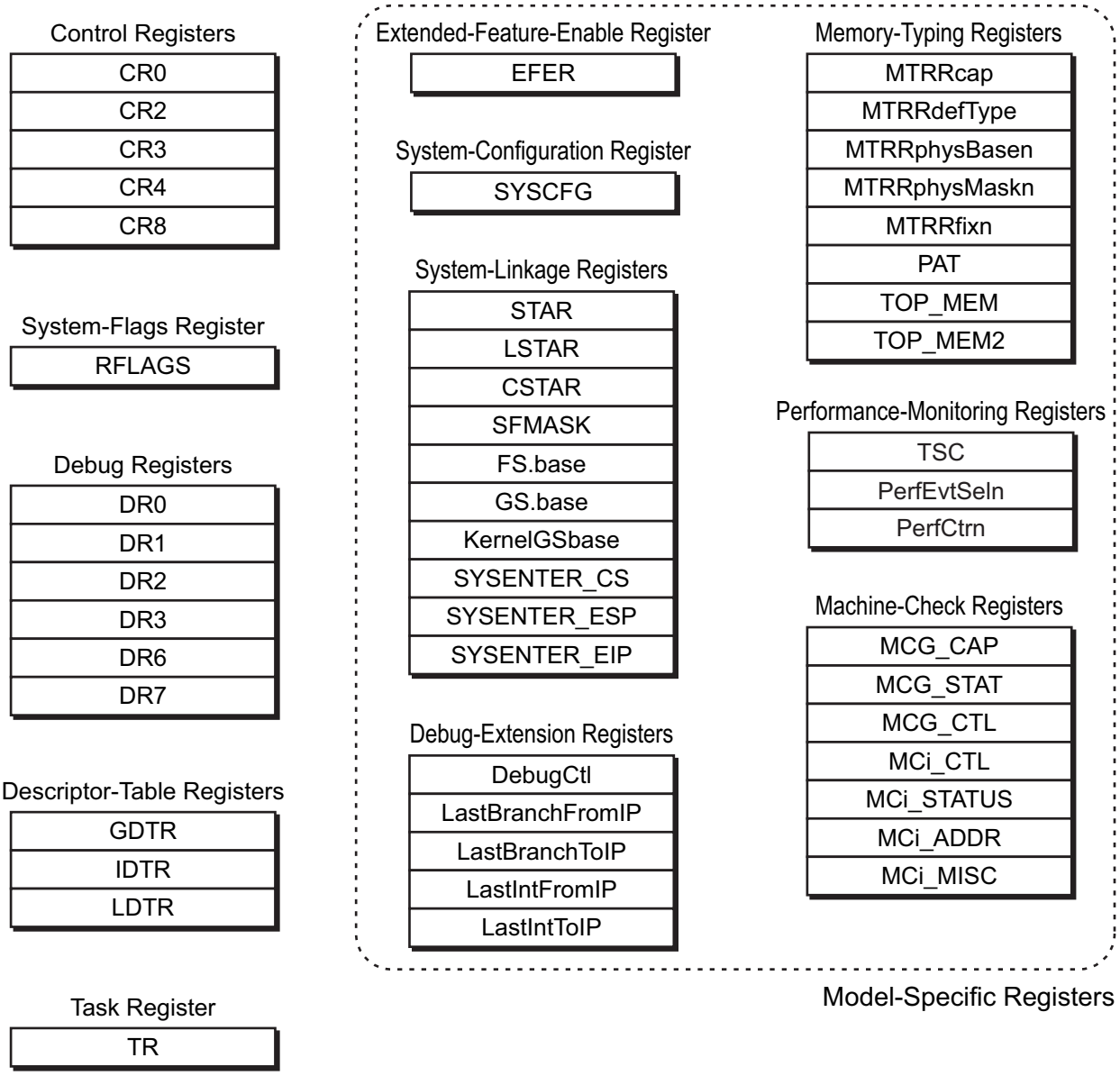


Figure 2-6. System Registers

Data Structures. Figure 2-7 on page 43 shows the system data structures. These are created and maintained by system software for use in protected mode. A processor running in protected mode uses these data structures to manage memory and protection, and to store program-state information when an interrupt or task switch occurs.

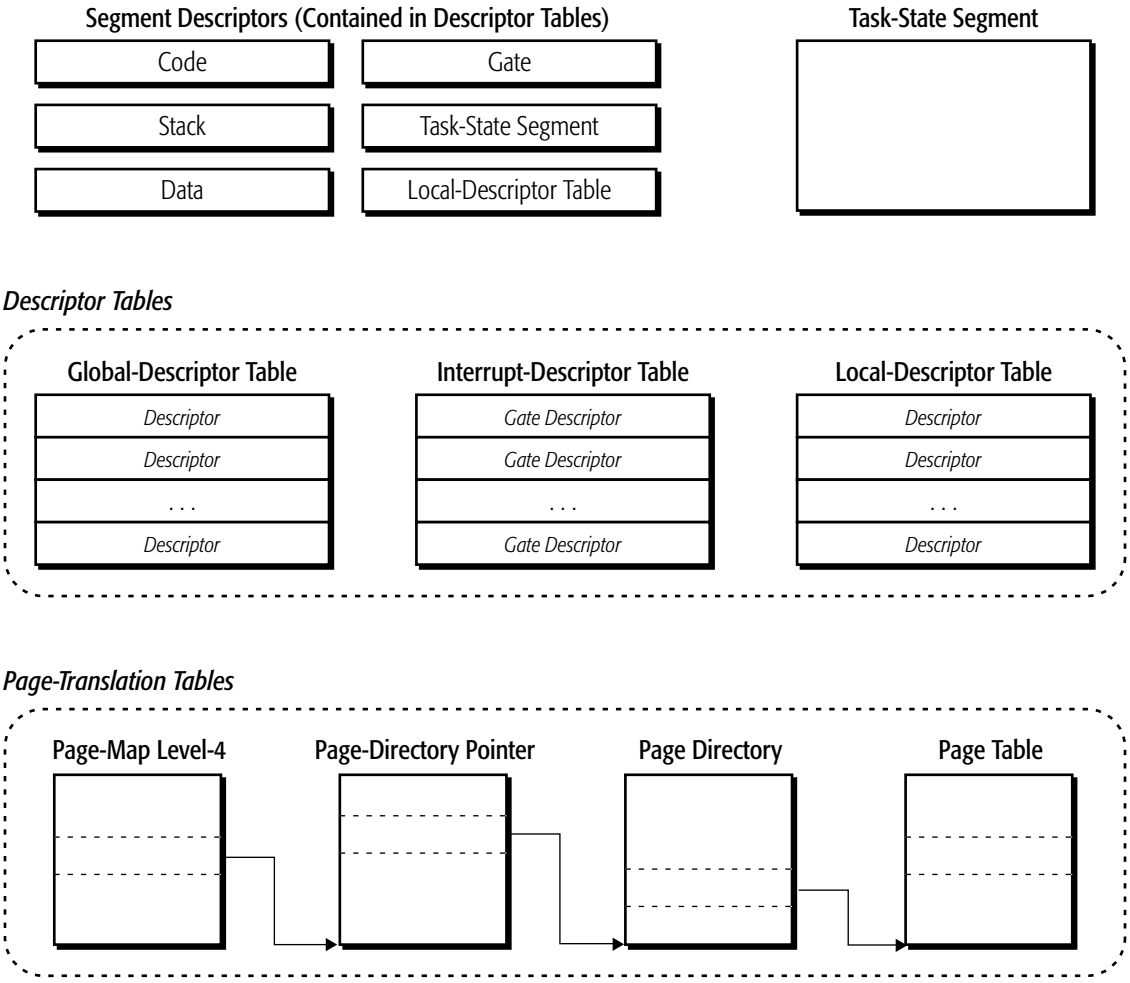


Figure 2-7. System Data Structures

2.3.3 SSE Instructions

Registers. The SSE instructions operate primarily on 128-bit and 256-bit floating-point vector operands located in the 256-bit YMM/XMM registers. Each 128-bit XMM register is defined as the lower octword of the corresponding YMM register. The number of available YMM/XMM data registers depends on the operating mode, as shown in Figure 2-8 below. In legacy and compatibility modes, eight YMM/XMM registers (YMM/XMM0–7) are available. In 64-bit mode, eight additional YMM/XMM data registers (YMM/XMM8–15) are available. These eight additional registers are addressed via the encoding extensions provided by the REX, VEX, and XOP prefixes.

The MXCSR register contains floating-point and other control and status flags used by the 128-bit media instructions. Some 128-bit media instructions also use the GPR (Figure 2-2 and Figure 2-3) and the MMX registers (Figure 2-12 on page 48) or set or clear flags in the rFLAGS register (see Figure 2-2 and Figure 2-3).

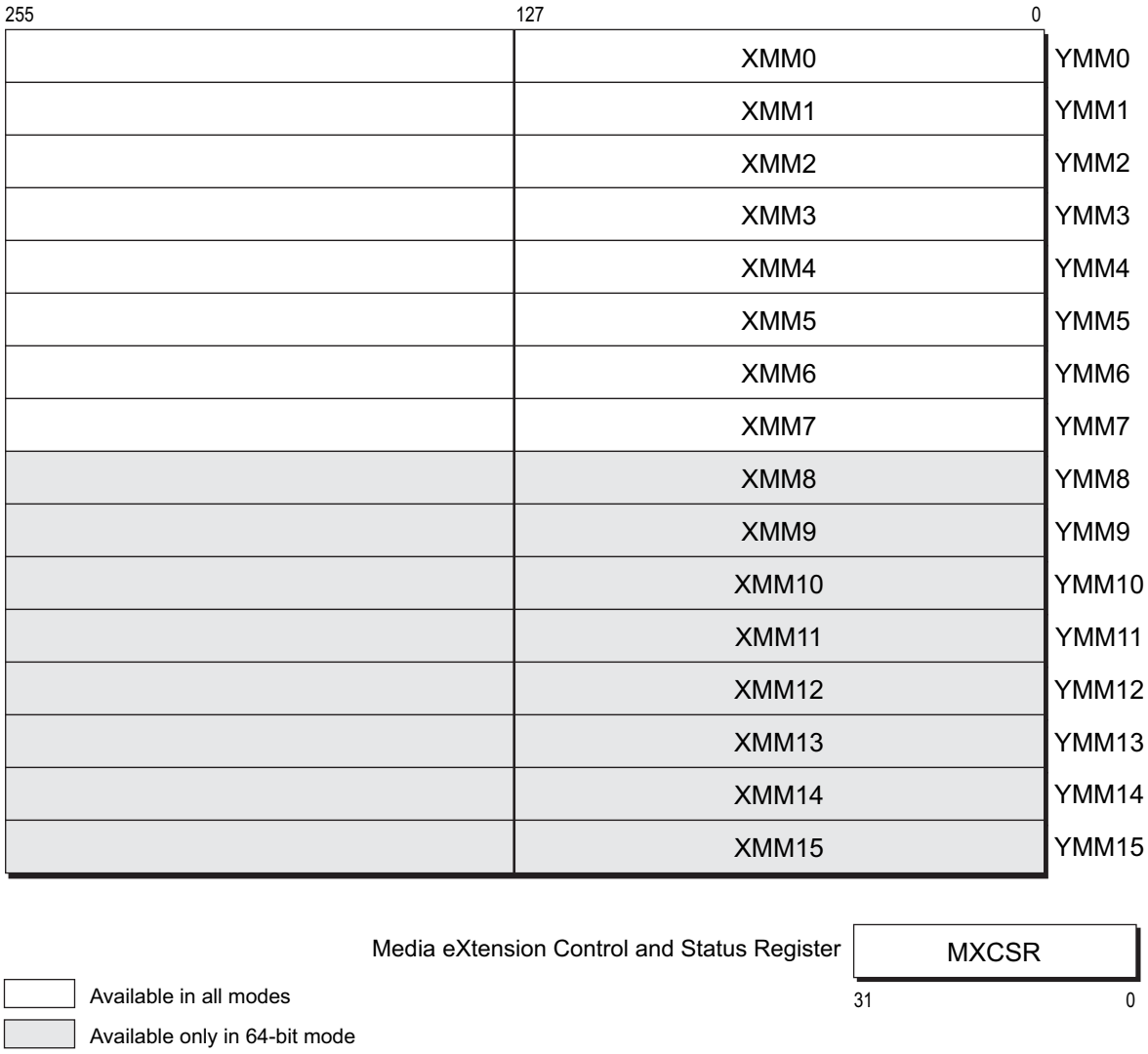


Figure 2-8. SSE Registers

Data Types. The SSE instruction set architecture provides support for 128-bit and 256-bit packed floating-point and integer data types as well as integer and floating-point scalars. Figure 2-9 below shows the 128-bit data types. Figure 2-10 on page 46 and Figure 2-11 on page 47 show the 256-bit data types. The floating-point data types include IEEE-754 single precision and double precision types.

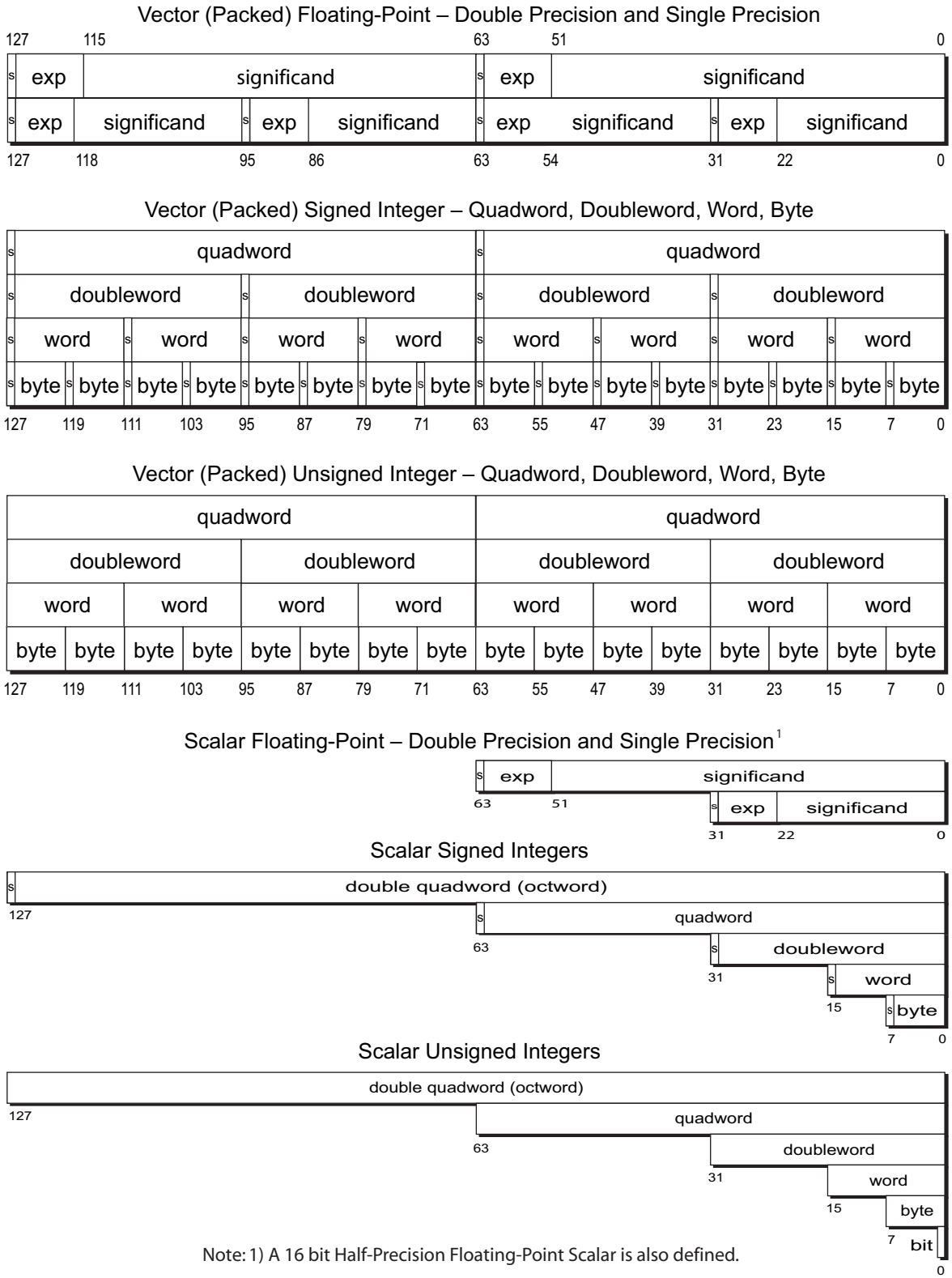


Figure 2-9. 128-Bit SSE Data Types

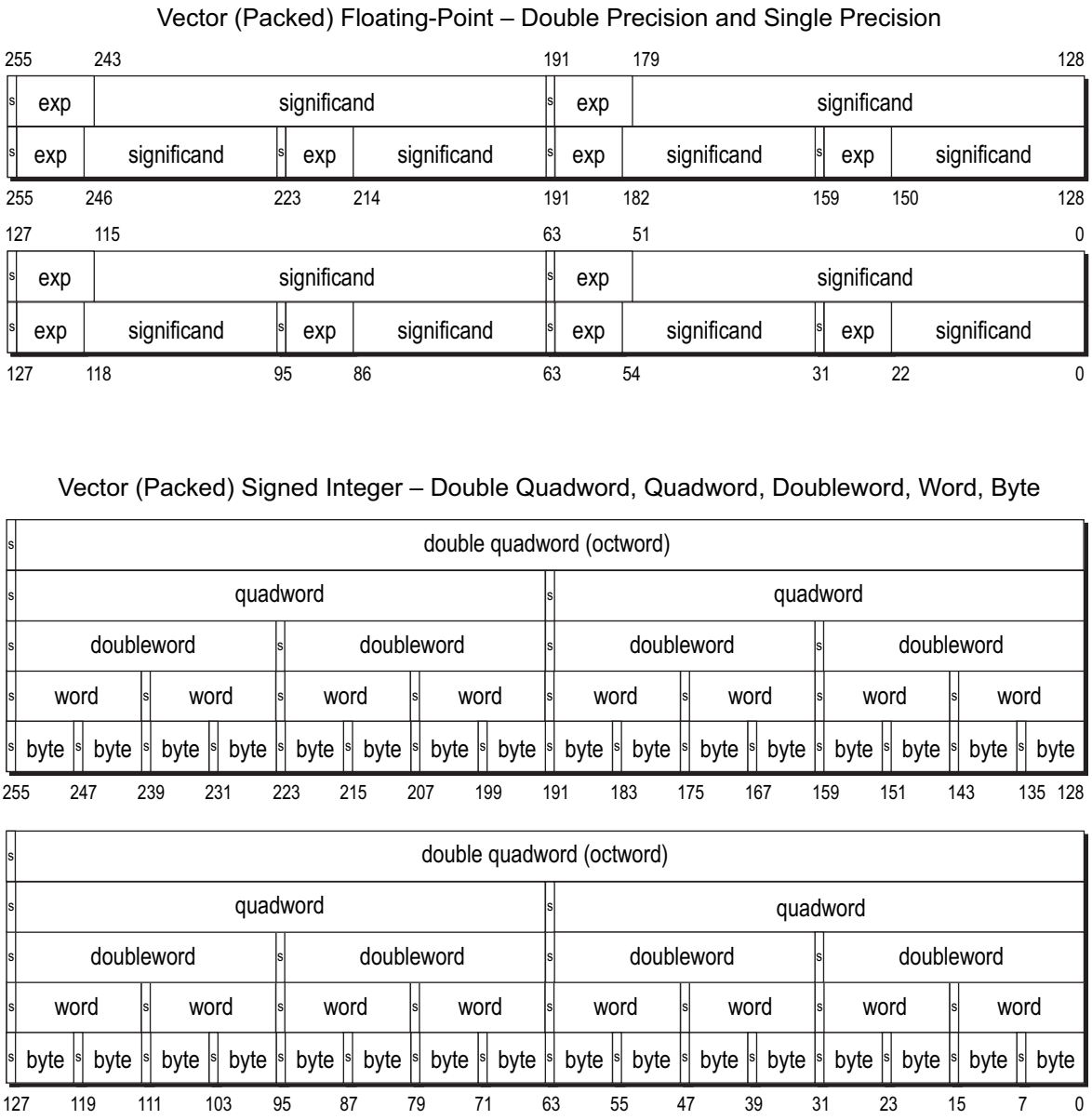
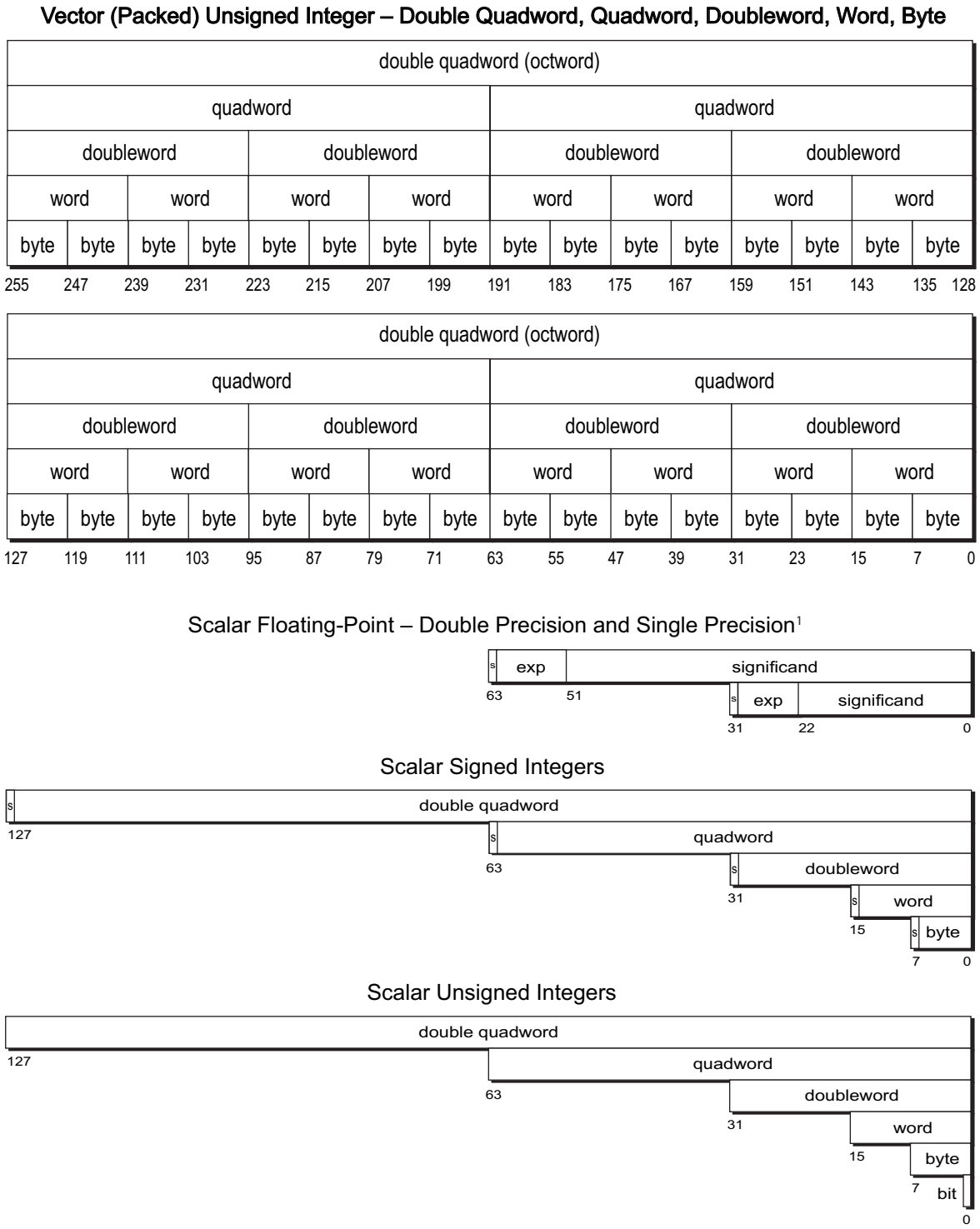


Figure 2-10. SSE 256-bit Data Types



Note: 1) A 16 bit Half-Precision Floating-Point Scalar is also defined.

Figure 2-11. SSE 256-Bit Data Types (Continued)

2.3.4 64-Bit Media Instructions

Registers. The 64-bit media instructions use the eight 64-bit MMX registers, as shown in Figure 2-12. These registers are mapped onto the x87 floating-point registers, and 64-bit media instructions write the x87 tag word in a way that prevents an x87 instruction from using MMX data.

Some 64-bit media instructions also use the GPR (Figure 2-2 and Figure 2-3) and the XMM registers (Figure 2-8).

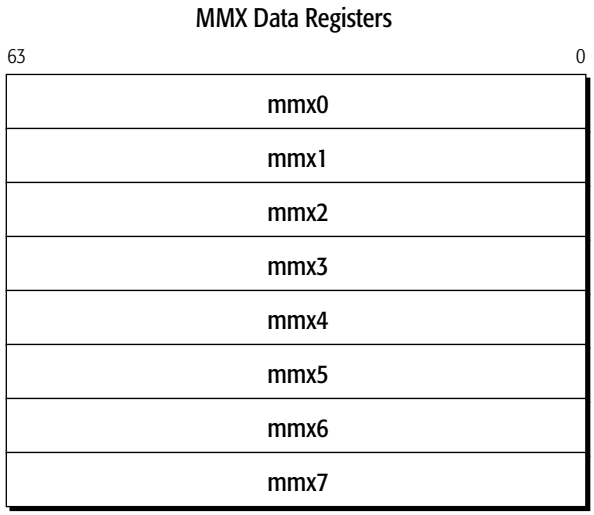


Figure 2-12. 64-Bit Media Registers

Data Types. Figure 2-13 on page 49 shows the 64-bit media data types. They include floating-point and integer vectors and integer scalars. The floating-point data type, used by 3DNow! instructions, consists of a packed vector or two IEEE-754 32-bit single-precision data types. Unlike other kinds of floating-point instructions, however, the 3DNow!™ instructions do not generate floating-point exceptions. For this reason, there is no register for reporting or controlling the status of exceptions in the 64-bit-media instruction subset.

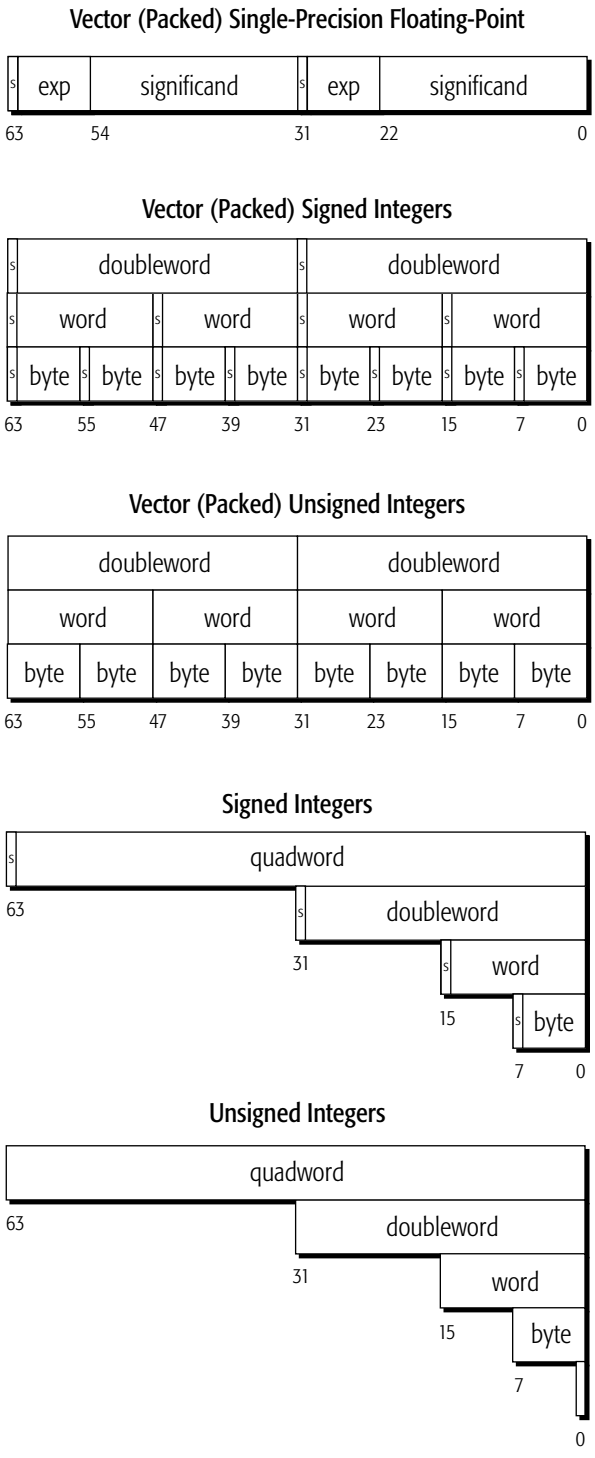


Figure 2-13. 64-Bit Media Data Types

2.3.5 x87 Floating-Point Instructions

Registers. The x87 floating-point instructions use the x87 registers shown in Figure 2-14. There are eight 80-bit data registers, three 16-bit registers that hold the x87 control word, status word, and tag word, and three registers (last instruction pointer, last opcode, last data pointer) that hold information about the last x87 operation.

The physical data registers are named FPR0–FPR7, although x87 software references these registers as a stack of registers, named ST(0)–ST(7). The x87 instructions store operands only in their own 80-bit floating-point registers or in memory. They do not access the GPR or XMM registers.

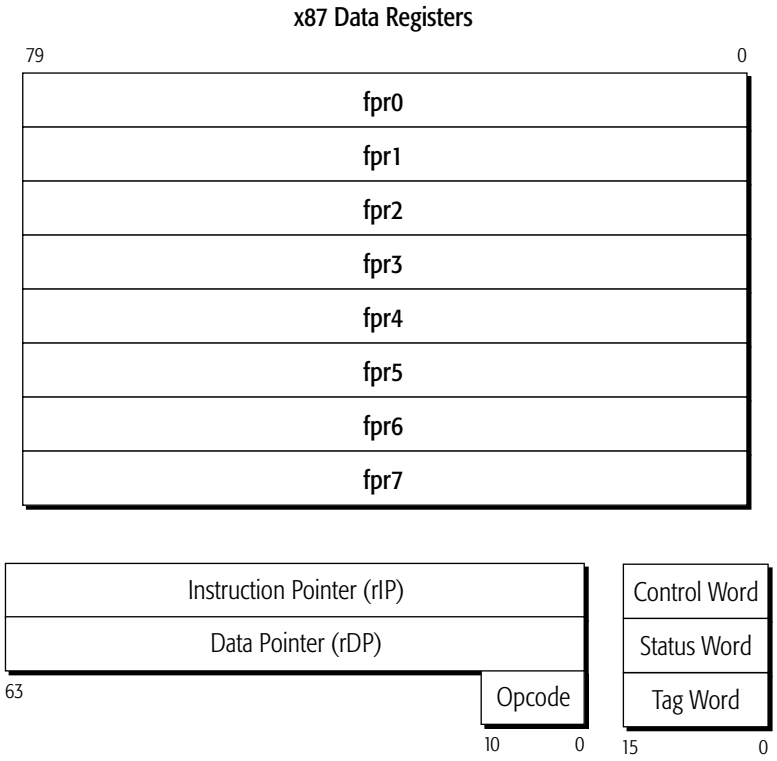


Figure 2-14. x87 Registers

Data Types. Figure 2-15 on page 51 shows all x87 data types. They include three floating-point formats (80-bit double-extended precision, 64-bit double precision, and 32-bit single precision), three signed-integer formats (quadword, doubleword, and word), and an 80-bit packed binary-coded decimal (BCD) format.

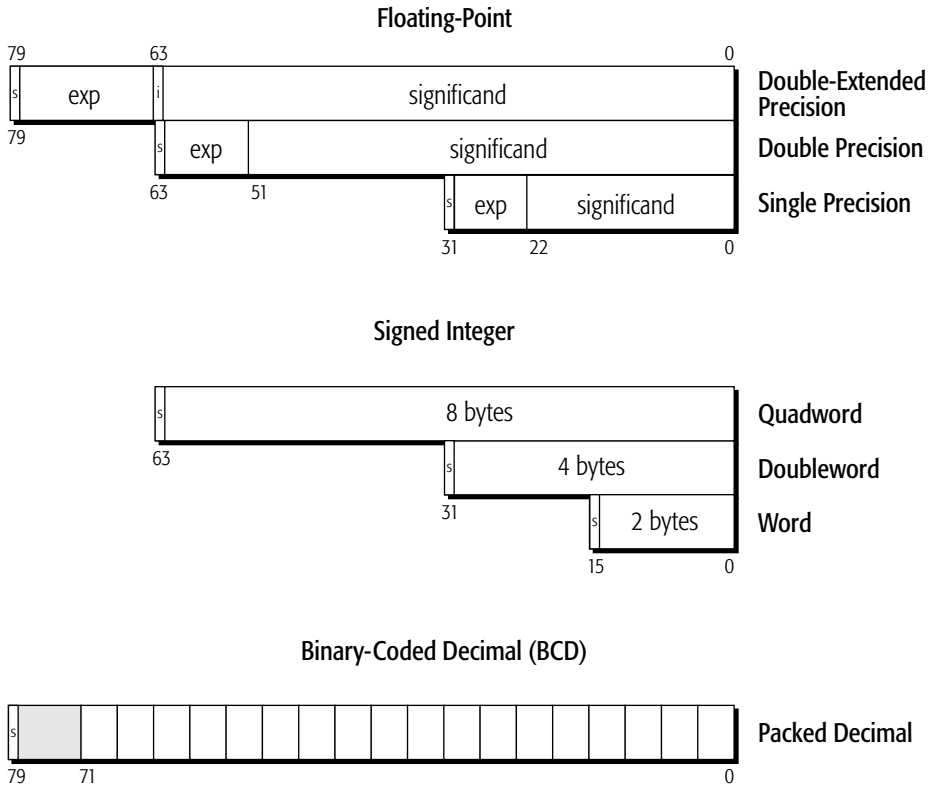


Figure 2-15. x87 Data Types

2.4 Summary of Exceptions

Table 2-1 on page 52 lists all possible exceptions. The table shows the interrupt-vector numbers, names, mnemonics, source, and possible causes. Exceptions that apply to specific instructions are documented with each instruction in the instruction-detail pages that follow.

Table 2-1. Interrupt-Vector Source and Cause

Vector	Interrupt (Exception)	Mnemonic	Source	Cause
0	Divide-By-Zero-Error	#DE	Software	DIV, IDIV, AAM instructions
1	Debug	#DB	Internal	Instruction accesses and data accesses
2	Non-Maskable-Interrupt	#NMI	External	External NMI signal
3	Breakpoint	#BP	Software	INT3 instruction
4	Overflow	#OF	Software	INTO instruction
5	Bound-Range	#BR	Software	BOUND instruction
6	Invalid-Opcode	#UD	Internal	Invalid instructions
7	Device-Not-Available	#NM	Internal	x87 instructions
8	Double-Fault	#DF	Internal	Interrupt during an interrupt
9	Coprocessor-Segment-Overrun	—	External	Unsupported (reserved)
10	Invalid-TSS	#TS	Internal	Task-state segment access and task switch
11	Segment-Not-Present	#NP	Internal	Segment access through a descriptor
12	Stack	#SS	Internal	SS register loads and stack references
13	General-Protection	#GP	Internal	Memory accesses and protection checks
14	Page-Fault	#PF	Internal	Memory accesses when paging enabled
15	Reserved	—		
16	Floating-Point Exception-Pending	#MF	Software	x87 floating-point and 64-bit media floating-point instructions
17	Alignment-Check	#AC	Internal	Memory accesses
18	Machine-Check	#MC	Internal External	Model specific
19	SIMD Floating-Point	#XF	Internal	128-bit media floating-point instructions
20	Reserved	—		
21	Control-Protection	#CP	Internal	Shadow Stack Protection checks
22—27	Reserved (Internal and External)	—		
28	Hypervisor Injection Exception	#HV	Software	Event injection
29	VMM Communication Exception	#VC	Internal	Virtualization event
30	SVM Security Exception	#SX	External	Security-sensitive events
31	Reserved (Internal and External)	—		
0—255	External Interrupts (Maskable)	#INTR	External	External interrupt signal
0—255	Software Interrupts	—	Software	INT*n* instruction

2.5 Notation

2.5.1 Mnemonic Syntax

Each instruction has a syntax that includes the mnemonic and any operands that the instruction can take. Figure 2-16 shows an example of a syntax in which the instruction takes two operands. In most instructions that take two operands, the first (left-most) operand is both a source operand (the first source operand) and the destination operand. The second (right-most) operand serves only as a source, not a destination.

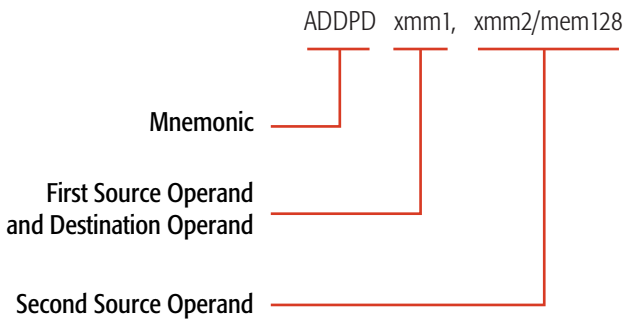


Figure 2-16. Syntax for Typical Two-Operand Instruction

The following notation is used to denote the size and type of source and destination operands:

- *cReg*—Control register.
- *dReg*—Debug register.
- *imm8*—Byte (8-bit) immediate.
- *imm16*—Word (16-bit) immediate.
- *imm16/32*—Word (16-bit) or doubleword (32-bit) immediate.
- *imm32*—Doubleword (32-bit) immediate.
- *imm32/64*—Doubleword (32-bit) or quadword (64-bit) immediate.
- *imm64*—Quadword (64-bit) immediate.
- *mem*—An operand of unspecified size in memory.
- *mem8*—Byte (8-bit) operand in memory.
- *mem16*—Word (16-bit) operand in memory.
- *mem16/32*—Word (16-bit) or doubleword (32-bit) operand in memory.
- *mem32*—Doubleword (32-bit) operand in memory.
- *mem32/48*—Doubleword (32-bit) or 48-bit operand in memory.
- *mem48*—48-bit operand in memory.

- *mem64*—Quadword (64-bit) operand in memory.
- *mem128*—Double quadword (128-bit) operand in memory.
- *mem16:16*—Two sequential word (16-bit) operands in memory.
- *mem16:32*—A doubleword (32-bit) operand followed by a word (16-bit) operand in memory.
- *mem32real*—Single-precision (32-bit) floating-point operand in memory.
- *mem16int*—Word (16-bit) integer operand in memory.
- *mem32int*—Doubleword (32-bit) integer operand in memory.
- *mem64real*—Double-precision (64-bit) floating-point operand in memory.
- *mem64int*—Quadword (64-bit) integer operand in memory.
- *mem80real*—Double-extended-precision (80-bit) floating-point operand in memory.
- *mem80dec*—80-bit packed BCD operand in memory, containing 18 4-bit BCD digits.
- *mem2env*—16-bit x87 control word or x87 status word.
- *mem14/28env*—14-byte or 28-byte x87 environment. The x87 environment consists of the x87 control word, x87 status word, x87 tag word, last non-control instruction pointer, last data pointer, and opcode of the last non-control instruction completed.
- *mem94/108env*—94-byte or 108-byte x87 environment and register stack.
- *mem512env*—512-byte environment for 128-bit media, 64-bit media, and x87 instructions.
- *mmx*—Quadword (64-bit) operand in an MMX register.
- *mmx1*—Quadword (64-bit) operand in an MMX register, specified as the left-most (first) operand in the instruction syntax.
- *mmx2*—Quadword (64-bit) operand in an MMX register, specified as the right-most (second) operand in the instruction syntax.
- *mmx/mem32*—Doubleword (32-bit) operand in an MMX register or memory.
- *mmx/mem64*—Quadword (64-bit) operand in an MMX register or memory.
- *mmx1/mem64*—Quadword (64-bit) operand in an MMX register or memory, specified as the left-most (first) operand in the instruction syntax.
- *mmx2/mem64*—Quadword (64-bit) operand in an MMX register or memory, specified as the right-most (second) operand in the instruction syntax.
- *moffset*—Direct memory offset that specifies an operand in memory.
- *moffset8*—Direct memory offset that specifies a byte (8-bit) operand in memory.
- *moffset16*—Direct memory offset that specifies a word (16-bit) operand in memory.
- *moffset32*—Direct memory offset that specifies a doubleword (32-bit) operand in memory.
- *moffset64*—Direct memory offset that specifies a quadword (64-bit) operand in memory.
- *pntr16:16*—Far pointer with 16-bit selector and 16-bit offset.
- *pntr16:32*—Far pointer with 16-bit selector and 32-bit offset.
- *reg*—Operand of unspecified size in a GPR register.

- *reg8*—Byte (8-bit) operand in a GPR register.
- *reg16*—Word (16-bit) operand in a GPR register.
- *reg16/32*—Word (16-bit) or doubleword (32-bit) operand in a GPR register.
- *reg32*—Doubleword (32-bit) operand in a GPR register.
- *reg64*—Quadword (64-bit) operand in a GPR register.
- *reg/mem8*—Byte (8-bit) operand in a GPR register or memory.
- *reg/mem16*—Word (16-bit) operand in a GPR register or memory.
- *reg/mem32*—Doubleword (32-bit) operand in a GPR register or memory.
- *reg/mem64*—Quadword (64-bit) operand in a GPR register or memory.
- *rel8off*—Signed 8-bit offset relative to the instruction pointer.
- *rel16off*—Signed 16-bit offset relative to the instruction pointer.
- *rel32off*—Signed 32-bit offset relative to the instruction pointer.
- *segReg or sReg*—Word (16-bit) operand in a segment register.
- *ST(0)*—x87 stack register 0.
- *ST(i)*—x87 stack register *i*, where *i* is between 0 and 7.
- *xmm*—Double quadword (128-bit) operand in an XMM register.
- *xmm1*—Double quadword (128-bit) operand in an XMM register, specified as the left-most (first) operand in the instruction syntax.
- *xmm2*—Double quadword (128-bit) operand in an XMM register, specified as the right-most (second) operand in the instruction syntax.
- *xmm/mem64*—Quadword (64-bit) operand in a 128-bit XMM register or memory.
- *xmm/mem128*—Double quadword (128-bit) operand in an XMM register or memory.
- *xmm1/mem128*—Double quadword (128-bit) operand in an XMM register or memory, specified as the left-most (first) operand in the instruction syntax.
- *xmm2/mem128*—Double quadword (128-bit) operand in an XMM register or memory, specified as the right-most (second) operand in the instruction syntax.
- *ymm*—Double octword (256-bit) operand in an YMM register.
- *ymm1*—Double octword (256-bit) operand in an YMM register, specified as the left-most (first) operand in the instruction syntax.
- *ymm2*—Double octword (256-bit) operand in an YMM register, specified as the right-most (second) operand in the instruction syntax.
- *ymm/mem64*—Quadword (64-bit) operand in a 256-bit YMM register or memory.
- *ymm/mem128*—Double quadword (128-bit) operand in an YMM register or memory.
- *ymm1/mem256*—Double octword (256-bit) operand in an YMM register or memory, specified as the left-most (first) operand in the instruction syntax.

- *ymm2/mem256*—Double octword (256-bit) operand in an YMM register or memory, specified as the right-most (second) operand in the instruction syntax.

2.5.2 Opcode Syntax

In addition to the notation shown above in "Mnemonic Syntax" on page 53, the following notation indicates the size and type of operands in the syntax of an instruction opcode:

- */digit*—Indicates that the ModRM byte specifies only one register or memory (r/m) operand. The digit is specified by the ModRM reg field and is used as an instruction-opcode extension. Valid digit values range from 0 to 7.
- */r*—Indicates that the ModRM byte specifies both a register operand and a reg/mem (register or memory) operand.
- *cb, cw, cd, cp*—Specifies a code-offset value and possibly a new code-segment register value. The value following the opcode is either one byte (cb), two bytes (cw), four bytes (cd), or six bytes (cp).
- *ib, iw, id, iq*—Specifies an immediate-operand value. The opcode determines whether the value is signed or unsigned. The value following the opcode, ModRM, or SIB byte is either one byte (ib), two bytes (iw), or four bytes (id). Word and doubleword values start with the low-order byte.
- *+rb, +rw, +rd, +rq*—Specifies a register value that is added to the hexadecimal byte on the left, forming a one-byte opcode. The result is an instruction that operates on the register specified by the register code. Valid register-code values are shown in Table 2-2.
- *m64*—Specifies a quadword (64-bit) operand in memory.
- *+i*—Specifies an x87 floating-point stack operand, ST(*i*). The value is used only with x87 floating-point instructions. It is added to the hexadecimal byte on the left, forming a one-byte opcode. Valid values range from 0 to 7.

Table 2-2. +rb, +rw, +rd, and +rq Register Value

REX.B Bit[1]	Value	Specified Register			
		+rb	+rw	+rd	+rq
0 or no REX Prefix	0	AL	AX	EAX	RAX
	1	CL	CX	ECX	RCX
	2	DL	DX	EDX	RDX
	3	BL	BX	EBX	RBX
	4	AH, SPL[1]	SP	ESP	RSP
	5	CH, BPL[1]	BP	EBP	RBP
	6	DH, SIL[1]	SI	ESI	RSI
	7	BH, DIL[1]	DI	EDI	RDI

1. See "REX Prefix" on page 14.

Table 2-2. +rb, +rw, +rd, and +rq Register Value (continued)

REX.B Bit[1]	Value	Specified Register			
		+rb	+rw	+rd	+rq
1	0	R8B	R8W	R8D	R8
	1	R9B	R9W	R9D	R9
	2	R10B	R10W	R10D	R10
	3	R11B	R11W	R11D	R11
	4	R12B	R12W	R12D	R12
	5	R13B	R13W	R13D	R13
	6	R14B	R14W	R14D	R14
	7	R15B	R15W	R15D	R15

1. *See “REX Prefix” on page 14.*

2.5.3 Pseudocode Definition

Pseudocode examples are given for the actions of several complex instructions (for example, see “CALL (Near)” on page 130). The following definitions apply to all such pseudocode examples:

```
////////////////////////////////////////////////////////////////////////////////
// Pseudo Code Definition
////////////////////////////////////////////////////////////////////////////////
//
// Comments start with double slashes.
//
// '=' can mean "is", or assignment based on context
// '==' is the equals comparison operator
//
////////////////////////////////////////////////////////////////////////////////
// Constants
////////////////////////////////////////////////////////////////////////////////

0               // numbers are in base-10 (decimal), unless followed by a suffix
0000_0001b      // a number in binary notation, underbars added for readability
FFE0_0000h      // a number expressed in hexadecimal notation

// in the following, '&&' is the logical AND operator. See "Logical Operators"
// below.
// reg[fld] identifies a field (one or more bits) within architected register
// or within a sub-element of a larger data structure. A dot separates the
// higher-level data structure name from the sub-element name.
//
CS.desc = Code Segment descriptor   // CS.desc has sub-elements: base, limit, attr
SS.desc = Stack Segment descriptor  // SS.desc has the same sub-elements
CS.desc.base = base subfield of CS.desc
CS = Code Segment Register
SS = Stack Segment Register
CPL = Current Privilege Level (0 <= CPL <= 3)
REAL_MODE = (CR0[PE] == 0)
```

```
PROTECTED_MODE = ((CR0[PE] == 1) && (RFLAGS[VM] == 0))
VIRTUAL_MODE = ((CR0[PE] == 1) && (RFLAGS[VM] == 1))
LEGACY_MODE = (EFER[LMA] == 0)
LONG_MODE = (EFER[LMA] == 1)
64BIT_MODE = ((EFER[LMA]==1) && (CS_desc.attr[L] == 1) && (CS_desc.attr[D] == 0))
COMPATIBILITY_MODE = (EFER[LMA] == 1) && (CS_desc.attr[L] == 0)
PAGING_ENABLED = (CR0[PG] == 1)
ALIGNMENT_CHECK_ENABLED = ((CR0[AM] == 1) && (RFLAGS[AC] == 1) && (CPL == 3))

OPERAND_SIZE = 16, 32, or 64   // size, in bits, of an operand
// OPERAND_SIZE depends on processor mode, the current code segment descriptor
// default operand size [D], presence of the operand size override prefix (66h)
// and, in 64-bit mode, the REX prefix.
// NOTE: Specific instructions take 8-bit operands, but for these instructions,
// operand size is fixed and the variable OPERAND_SIZE is not needed.

ADDRESS_SIZE = 16, 32, or 64   // size, in bits, of the effective address for
// memory reads. ADDRESS_SIZE depends processor mode, the current code segment
// descriptor default operand size [D], and the presence of the address size
// override prefix (67h)

STACK_SIZE = 16, 32, or 64     // size, in bits of stack operation operand
// STACK_SIZE depends on current code segment descriptor attribute D bit and
// the Stack Segment descriptor attribute B bit.


////////////////////////////////////////////////////////////////////////////////
// Architected Registers
////////////////////////////////////////////////////////////////////////////////
// Identified using abbreviated names assigned by the Architecture; can represent
//  the register or its contents depending on context.
RAX = the 64-bit contents of the general-purpose register
EAX = 32-bit contents of GPR EAX
AX = 16-bit contents of GPR AX
AL = lower 8 bits of GPR AX
AH = upper 8 bits of GPR AX

index_of(reg) = value used to encode the register.
index_of(AX) = 0000b
index_of(RAX) = 0000b

// in legacy and compatibility modes the msb of the index is fixed as 0


////////////////////////////////////////////////////////////////////////////////
// Defined Variables
////////////////////////////////////////////////////////////////////////////////

old_RIP = RIP at the start of current instruction
old_RSP = RSP at the start of current instruction
old_RFLAGS = RFLAGS at the start of the instruction
```

```
old_CS = CS selector at the start of current instruction
old_DS = DS selector at the start of current instruction
old_ES = ES selector at the start of current instruction
old_FS = FS selector at the start of current instruction
old_GS = GS selector at the start of current instruction
old_SS = SS selector at the start of current instruction

RIP = the current RIP register
RSP = the current RSP register
RBP = the current RBP register
RFLAGS = the current RFLAGS register
next_RIP = RIP at start of next instruction

CS.desc = the current CS descriptor, including the subfields:
     base limit attr
SS.desc = the current SS descriptor, including the subfields:
     base limit attr

SRC = the instruction’s source operand
SRC1 = the instruction's first source operand
SRC2 = the instruction's second source operand
SRC3 = the instruction's third source operand
IMM8 = 8-bit immediate encoded in the instruction
IMM16 = 16-bit immediate encoded in the instruction
IMM32 = 32-bit immediate encoded in the instruction
IMM64 = 64-bit immediate encoded in the instruction
DEST = instruction’s destination register

temp_* // 64-bit temporary register
temp_*_desc   // temporary descriptor, with sub-elements:
              // if it points to a block of memory: base limit attr
              // if it’s a gate descriptor: offet segment attr

NULL = 0000h // null selector is all zeros

////////////////////////////////////////////////////////////////////////////////
// Exceptions
////////////////////////////////////////////////////////////////////////////////
EXCEPTION [#GP(0)] // Signals an exception; error code in parenthesis
EXCEPTION [#UD]    // if no error code

// possible exception types:
#DE // Divide-By-Zero-Error Exception (Vector 0)
#DB // Debug Exception (Vector 1)
#BP // INT3 Breakpoint Exception (Vector 3)
#OF // INTO Overflow Exception (Vector 4)
#BR // Bound-Range Exception (Vector 5)
#UD // Invalid-Opcode Exception (Vector 6)
#NM // Device-Not-Available Exception (Vector 7)
#DF // Double-Fault Exception (Vector 8)
#TS // Invalid-TSS Exception (Vector 10)
```

```
#NP // Segment-Not-Present Exception (Vector 11)
#SS // Stack Exception (Vector 12)
#GP // General-Protection Exception (Vector 13)
#PF // Page-Fault Exception (Vector 14)
#MF // x87 Floating-Point Exception-Pending (Vector 16)
#AC // Alignment-Check Exception (Vector 17)
#MC // Machine-Check Exception (Vector 18)
#XF // SIMD Floating-Point Exception (Vector 19)

////////////////////////////////////////////////////////////////////////////////
// Implicit Assignments
////////////////////////////////////////////////////////////////////////////////

// V,Z,A,S are integer variables, assigned a value when an instruction begins
// executing (they can be assigned a different value in the middle of an
// instruction, if needed)
IF (OPERAND_SIZE == 16) V = 2
IF (OPERAND_SIZE == 32) V = 4
IF (OPERAND_SIZE == 64) V = 8
IF (OPERAND_SIZE == 16) Z = 2
IF (OPERAND_SIZE == 32) Z = 4
IF (OPERAND_SIZE == 64) Z = 4
IF (ADDRESS_SIZE == 16) A = 2
IF (ADDRESS_SIZE == 32) A = 4
IF (ADDRESS_SIZE == 64) A = 8
IF (STACK_SIZE == 16)   S = 2
IF (STACK_SIZE == 32)   S = 4
IF (STACK_SIZE == 64)   S = 8

////////////////////////////////////////////////////////////////////////////////
// Bit Range Inside a Register
////////////////////////////////////////////////////////////////////////////////

temp_data[x:y] // Bits x through y (inclusive) of temp_data

////////////////////////////////////////////////////////////////////////////////
// Variables and data types
////////////////////////////////////////////////////////////////////////////////
NxtValue = 5       //default data type is unsigned int.

int            //abstract data type representing an integer
bool           //abstract data type; either TRUE or FALSE
vector         //An array of data elements. Individual elements are accessed via
               //an unsigned integer zero-based index. Elements have a data type.
bit            //a single bit
byte           //8-bit value
word           //16-bit value
doubleword     //32-bit value
quadword       //64-bit value
octword        //128-bit value
double octword //256-bit value
```

```
unsigned int aval  //treat aval as an unsigned integer value
signed int valx    //treat valx as a signed integer value
bit vector b_vect  //b_vect is an array of data elements. Each element is a bit.
b_vect[5]          //The sixth element (bit) in the array. Indices are 0-based.

////////////////////////////////////////////////////////////////////////////
// Elements Within a packed data type
////////////////////////////////////////////////////////////////////////////

// element i of size w occupies bits [wi-1:wi]

////////////////////////////////////////////////////////////////////////////
// Moving Data From One Register To Another
////////////////////////////////////////////////////////////////////////////
temp_dest.b = temp_src;  // 1-byte move (copies lower 8 bits of temp_src to
                         // temp_dest, preserving the upper 56 bits of temp_dest)
temp_dest.w = temp_src;  // 2-byte move (copies lower 16 bits of temp_src to
                         // temp_dest, preserving the upper 48 bits of temp_dest)
temp_dest.d = temp_src;  // 4-byte move (copies lower 32 bits of temp_src to
                         // temp_dest; zeros out the upper 32 bits of temp_dest)
temp_dest.q = temp_src;  // 8-byte move (copies all 64 bits of temp_src to
                         // temp_dest)
temp_dest.v = temp_src;  // 2-byte move if V==2
                         // 4-byte move if V==4
                         // 8-byte move if V==8
temp_dest.z = temp_src;  // 2-byte move if Z==2
                         // 4-byte move if Z==4
temp_dest.a = temp_src;  // 2-byte move if A==2
                         // 4-byte move if A==4
                         // 8-byte move if A==8
temp_dest.s = temp_src;  // 2-byte move if S==2
                         // 4-byte move if S==4
                         // 8-byte move if S==8

////////////////////////////////////////////////////////////////////////////
// Arithmetic Operators
////////////////////////////////////////////////////////////////////////////
a + b    // integer addition
a - b    // integer subtraction
a * b    // integer multiplication
a / b    // integer division.  Result is the quotient
a % b    // modulo.  Result is the remainder after a is divided by b
// multiplication has precedence over addition where precedence is not explicitly
// indicated by grouping terms with parentheses

////////////////////////////////////////////////////////////////////////////
// Bitwise Operators
////////////////////////////////////////////////////////////////////////////
// temp, a, and b are values or register contents of the same size
temp = a AND b;   // Corresponding bits of a and b are logically ANDed together
```

```
temp = a OR b;    // Corresponding bits of a and b are logically ORed together
temp = a XOR b;   // Each bit of temp is the exclusive OR of the corresponding
                  // bits of a and b
temp = NOT a;     // Each bit of temp is the complement of the corresponding
                  // bit of a

// Concatenation
value = {field1,field2,100b};  //pack values of field1, field2 and 100b
size_of(value) = (size_of(field1) + size_of(field2) + 3)

////////////////////////////////////////////////////////////////////////////
// Logical Shift Operators
////////////////////////////////////////////////////////////////////////////
temp = a << b;    // Result is a shifted left by _b_ bit positions. Zeros are
                  // shifted into vacant positions. Bits shifted out are lost.
temp = a >> b;    // Result is a shifted right by _b_ bit positions. Zeros are
                  // shifted into vacant positions. Bits shifted out are lost.

////////////////////////////////////////////////////////////////////////////
// Logical Operators
////////////////////////////////////////////////////////////////////////////
// a boolean variable can assume one of two values (TRUE or FALSE)
// In these examples, FOO, BAR, CONE, and HEAD have been defined to be boolean
// variables
FOO && BAR  // Logical AND
FOO || BAR  // Logical OR
!FOO        // Logical complement (NOT)

////////////////////////////////////////////////////////////////////////////
// Comparison Operators
////////////////////////////////////////////////////////////////////////////
// a and b are integer values.  The result is a boolean value.
a == b      // if a and b are equal, the result is TRUE; otherwise it is FALSE.
a != b     // if a and b are not equal, the result is TRUE; otherwise it is FALSE.
a > b      // if a is greater than b, the result is TRUE; otherwise it is FALSE.
a < b      // if a is less than b, the result is TRUE; otherwise it is FALSE.
a >= b     // if a is greater than or equal to b, the result is TRUE; otherwise
           // it is FALSE.
a <= b     // if a is less than or equal to b, the result is TRUE; otherwise
           // it is FALSE.
////////////////////////////////////////////////////////////////////////////
// Logical Expressions
////////////////////////////////////////////////////////////////////////////
// Logical binary (two operand) and unary (one operand) operators can be combined
// with comparison operators to form more complex expressions.  Parentheses are
// used to enclose comparison terms and to show precedence. If precedence is not
// explicitly shown, logical AND has precedence over logical OR. Unary operators
// have precedence over binary operators.

FOO && (a < b) || !BAR  // evaluate the comparison a < b first, then
                        // AND this with FOO. Finally OR this intermediate result
```

```
                        // with the complement of BAR.

// Logical expressions can be English phrases that can be evaluated to be TRUE
// or FALSE. Statements assume knowledge of the system architecture (Volumes 1 and
// 2).
////////////////////////////////////////////////////////////////////////////////

IF (it is raining)
   close the window

////////////////////////////////////////////////////////////////////////////////
// Assignment Operators
////////////////////////////////////////////////////////////////////////////////
a = a + b   // The value a is assigned the sum of the values a and b
            //
temp = R1   // The contents of the register temp is replaced by a copy of the
            // contents of register R1.
R0 += 2     // R0 is assigned the sum of the contents of R0 and the integer 2.
            //
R5 |= R6    // R5 is assigned the result of the bit-wise OR of the contents of R5
            // and R6. Contents of R6 is unchanged.
R4 &= R7    // R4 is assigned the result of the bit-wise AND of the contents of
            // R4 and R7. Contents of R7 is unchanged.
////////////////////////////////////////////////////////////////////////////////
// IF-THEN-ELSE
////////////////////////////////////////////////////////////////////////////////
IF (FOO) <expression>        // evaluation of <expression> is dependent on FOO
                             // being TRUE. If FOO is FALSE, <expression> is not
                             // evaluated.

IF (FOO)
   <dependent expression1>   // scope of IF is indicated by indentation
   ...
   <dependent expressionx>

IF (FOO)                     // If FOO is TRUE, <dependent expression> is
                             // evaluated and the remaining ELSEIF and ELSE
   <dependent expression>    // clauses are skipped.
                             //
ELSIF (BAR)                  // IF FOO is FALSE and BAR is TRUE, <alt expression>
   <alt expression>          // is evaluated and the subsequent ELSEIF or ELSE
                             // clauses are skipped.
ELSE
   <default expressions>     // evaluated if all the preceeding IF and ELSEIF
                             // conditions are FALSE.

IF ((FOO && BAR) || (CONE && HEAD))  // The condition can be an expression.
   <dependent expressions>

////////////////////////////////////////////////////////////////////////////////
// Loops
```

```
////////////////////////////////////////////////////////////////////////////
FOR i = <init_val> to <final_val>, BY <step>
    <expression>                    // scope of loop is indicated by indentation
                                    // if <step> = 1, may omit "BY" clause

// nested loop example
temp = 0                      //initialize temp
FOR i = 0 to 7                // i takes on the values 0 through 7 in succession
    temp += 1                 // In the outer loop. Evaluated a total of 8 times.
    For j = 0 to 7, BY 2      // j takes on the values 0, 2, 4, and 6; but not 7.
       <inner-most exp>       // This will be evaluated a total of 8 * 4 times.
<next expression outside both loops>

// C Language form of loop syntax is also allowed

FOR (i = 0; i < MAX; i++)
{
      <expressions>           //evaluated MAX times
}

////////////////////////////////////////////////////////////////////////////
// Functions
////////////////////////////////////////////////////////////////////////////
// Syntax for function definition
<return data type> <function_name>(argument,..)
   <expressions>
RETURN <result>

////////////////////////////////////////////////////////////////////////////
// Built-in Functions
////////////////////////////////////////////////////////////////////////////
SignExtend(arg)  // returns value of _arg_ sign extended to the width of the data
                 // type of the function. Data type of function is inferred from
                 // the context of the function's invocation.


ZeroExtend(arg)  // returns value of _arg_ zero extended to the width of the data
                 // type of the function. Data type of function is inferred from
                 // the context of the function's invocation.

indexof(reg)      //returns binary value used to encode reg specification

////////////////////////////////////////////////////////////////////////////
// READ_MEM
// General memory read. This zero-extends the data to 64 bits and returns it.
////////////////////////////////////////////////////////////////////////////

usage:
    temp = READ_MEM.x [seg:offset]   // where x is one of {v, z, b, w, d, q}
                                     // and denotes the size of the memory read
```

```
definition:

IF ((seg AND 0xFFFC) == NULL)
        // GP fault for using a null segment to reference memory
        EXCEPTION [#GP(0)]

    IF ((seg==CS) || (seg==DS) || (seg==ES) || (seg==FS) || (seg==GS))
        // CS,DS,ES,FS,GS check for segment limit or canonical

        IF ((!64BIT_MODE) && (offset is outside seg's limit))
            // #GP fault for segment limit violation in non-64-bit mode
            EXCEPTION [#GP(0)]

        IF ((64BIT_MODE) && (offset is non-canonical))
            // #GP fault for non-canonical address in 64-bit mode
            EXCEPTION [#GP(0)]

    ELSIF (seg==SS) // SS checks for segment limit or canonical

        IF ((!64BIT_MODE) && (offset is outside seg's limit))
            // stack fault for segment limit violation in non-64-bit mode
            EXCEPTION [#SS(0)]

        IF ((64BIT_MODE) && (offset is non-canonical))
            // stack fault for non-canonical address in 64-bit mode
            EXCEPTION [#SS(0)]

    ELSE // ((seg==GDT) || (seg==LDT) || (seg==IDT) || (seg==TSS))
        // GDT,LDT,IDT,TSS check for segment limit and canonical

        IF (offset > seg.limit)
            // #GP fault for segment limit violation in all modes
            EXCEPTION [#GP(0)]

        IF ((LONG_MODE) && (offset is non-canonical))
           EXCEPTION [#GP(0)]  // #GP fault for non-canonical address in long mode

    IF ((ALIGNMENT_CHECK_ENABLED) && (offset misaligned, considering its
                                      size and alignment))
        EXCEPTION [#AC(0)]

    IF ((64_bit_mode) && ((seg==CS) || (seg==DS) || (seg==ES) || (seg==SS))
        temp_linear = offset
    ELSE
        temp_linear = seg.base + offset

    IF ((PAGING_ENABLED) && (virtual-to-physical translation for temp_linear
                             results in a page-protection violation))
        EXCEPTION [#PF(error_code)]  // page fault for page-protection violation
                                     // (U/S violation, Reserved bit violation)
```

```
    IF ((PAGING_ENABLED) && (temp_linear is on a not-present page))
        EXCEPTION [#PF(error_code)]  // page fault for not-present page

    temp_data = memory [temp_linear].x   // zero-extends the data to 64
                                         // bits, and saves it in temp_data

    RETURN (temp_data)                   // return the zero-extended data


////////////////////////////////////////////////////////////////////////////
// WRITE_MEM  // General memory write
////////////////////////////////////////////////////////////////////////////

usage:
    WRITE_MEM.x [seg:offset] = temp.x   // where <X> is one of these:
                                        // {V, Z, B, W, D, Q} and denotes the
                                        // size of the memory write

definition:

    IF ((seg & 0xFFFC)== NULL)         // GP fault for using a null segment
                                    // to reference memory
        EXCEPTION [#GP(0)]

   IF ((seg==CS) || (seg==DS) || (seg==ES) || (seg==FS) || (seg==GS))
      // CS,DS,ES,FS,GS check for segment limit or canonical
      IF ((!64BIT_MODE) && (offset is outside seg's limit))
         // #GP fault for segment limit violation in non-64-bit mode
         EXCEPTION [#GP(0)]
      IF ((64BIT_MODE) && (offset is non-canonical))
         // #GP fault for non-canonical address in 64-bit mode
         EXCEPTION [#GP(0)]
   ELSEIF (seg==SS) // SS checks for segment limit or canonical
      IF ((!64BIT_MODE) && (offset is outside seg's limit))
         // stack fault for segment limit violation in non-64-bit mode
         EXCEPTION [#SS(0)]
      IF ((64BIT_MODE) && (offset is non-canonical))
         // stack fault for non-canonical address in 64-bit mode
         EXCEPTION [#SS(0)]
   ELSE  // ((seg==GDT) || (seg==LDT) || (seg==IDT) || (seg==TSS))
         // GDT,LDT,IDT,TSS check for segment limit and canonical
      IF (offset > seg.limit)
         // #GP fault for segment limit violation in all modes
         EXCEPTION [#GP(0)]
      IF ((LONG_MODE) && (offset is non-canonical))
         // #GP fault for non-canonical address in long mode
         EXCEPTION [#GP(0)]

    IF ((ALIGNMENT_CHECK_ENABLED) && (offset is misaligned, considering
                                      its size and alignment))
        EXCEPTION [#AC(0)]
```

```
    IF ((64_bit_mode) && ((seg==CS) || (seg==DS) || (seg==ES) || (seg==SS))
        temp_linear = offset
    ELSE
        temp_linear = seg.base + offset

    IF ((PAGING_ENABLED) && (the virtual-to-physical translation for
       temp_linear results in a page-protection violation))
    {
        EXCEPTION [#PF(error_code)]
                       // page fault for page-protection violation
                       // (U/S violation, Reserved bit violation)
    }

    IF ((PAGING_ENABLED) && (temp_linear is on a not-present page))
        EXCEPTION [#PF(error_code)]      // page fault for not-present page

    memory [temp_linear].x = temp.x      // write the bytes to memory


////////////////////////////////////////////////////////////////////////////////
// PUSH // Write data to the stack
////////////////////////////////////////////////////////////////////////////////

usage:
    PUSH.x temp         // where x is one of these: {v, z, b, w, d, q} and
                        // denotes the size of the push

definition:

    WRITE_MEM.x [SS:RSP.s - X] = temp.x      // write to the stack
    RSP.s = RSP - X                          // point RSP to the data just written


////////////////////////////////////////////////////////////////////////////////
// POP  // Read data from the stack, zero-extend it to 64 bits
////////////////////////////////////////////////////////////////////////////////

usage:
    POP.x temp            // where x is one of these: {v, z, b, w, d, q} and
                          // denotes the size of the pop

definition:

    temp = READ_MEM.x [SS:RSP.s]          // read from the stack
    RSP.s = RSP + X                       // point RSP above the data just read


////////////////////////////////////////////////////////////////////////////////
// READ_DESCRIPTOR // Read 8-byte descriptor from GDT/LDT, return the descriptor
////////////////////////////////////////////////////////////////////////////////
```

```
usage:
    temp_descriptor = READ_DESCRIPTOR (selector, chktype)
    // chktype field is one of the following:
    // cs_chk     used for far call and far jump
    // clg_chk    used when reading CS for far call or far jump through call gate
    // ss_chk     used when reading SS
    // iret_chk   used when reading CS for IRET or RETF
    // intcs_chk  used when readin the CS for interrupts and exceptions

definition:

    temp_offset = selector AND 0xfff8  // upper 13 bits give an offset
                                       // in the descriptor table

    IF (selector.TI == 0)              // read 8 bytes from the gdt, split it into
                                       // (base,limit,attr) if the type bits
        temp_desc = READ_MEM.q [gdt:temp_offset]
                                       // indicate a block of memory, or split
                                       // it into (segment,offset,attr)
                                       // if the type bits indicate
                                      // a gate, and save the result in temp_desc
    ELSE
        temp_desc = READ_MEM.q [ldt:temp_offset]
                                     // read 8 bytes from the LDT, split it into
                                      // (base,limit,attr) if the type bits
                                     // indicate a block of memory, or split
                                    // it into (segment,offset,attr) if the type
                                    // bits indicate a gate, and save the result
                                      // in temp_desc

    IF (selector.rpl or temp_desc.attr.dpl is illegal for the current mode/cpl)
        EXCEPTION [#GP(selector)]

    IF (temp_desc.attr.type is illegal for the current mode/chktype)
        EXCEPTION [#GP(selector)]

    IF (temp_desc.attr.p==0)
        EXCEPTION [#NP(selector)]

    RETURN (temp_desc)


////////////////////////////////////////////////////////////////////////////
// READ_IDT // Read an 8-byte descriptor from the IDT, return the descriptor
////////////////////////////////////////////////////////////////////////////

usage:
    temp_idt_desc = READ_IDT (vector)
                                     // "vector" is the interrupt vector number
```

```
definition:

    IF (LONG_MODE)        // long-mode idt descriptors are 16 bytes long
        temp_offset = vector*16
    ELSE // (LEGACY_MODE) legacy-protected-mode idt descriptors are 8 bytes long
        temp_offset = vector*8

    // read 8 bytes from the idt, split it into
    // (segment,offset,attr), and save it in temp_desc
    temp_desc = READ_MEM.q [idt:temp_offset]

    IF (temp_desc.attr.dpl is illegal for the current mode/cpl)
        // exception, with error code that indicates this IDT gate
        EXCEPTION [#GP(vector*8+2)]

    IF (temp_desc.attr.type is illegal for the current mode)
        // exception, with error code that indicates this IDT gate
        EXCEPTION [#GP(vector*8+2)]

    IF (temp_desc.attr.p==0)
        // segment-not-present exception, with an error code that
        // indicates this IDT gate
        EXCEPTION [#NP(vector*8+2)]

    RETURN (temp_desc)


////////////////////////////////////////////////////////////////////////////////
// READ_INNER_LEVEL_SP
// Read a new stack pointer (RSP or SS:ESP) from the TSS
////////////////////////////////////////////////////////////////////////////////

usage:
    temp_SS_desc:temp_RSP = READ_INNER_LEVEL_SP (new_cpl, ist_index)

definition:

    IF (LONG_MODE)
    {
        IF (ist_index>0)
            temp_RSP = READ_MEM.q [tss:ist_index*8+28] // read ISTn stack
                                                       // pointer from the TSS
        ELSE // (ist_index==0)
            temp_RSP = READ_MEM.q [tss:new_cpl*8+4]    // read RSPn stack
                                                       // pointer from the TSS

        // in long mode, changing to lower cpl sets SS.sel to NULL+new_cpl
        temp_SS_desc.sel = NULL + new_cpl

    ELSE // (LEGACY_MODE)
    {
```

```
        temp_RSP = READ_MEM.d [tss:new_cpl*8+4]         // read ESPn from the TSS
        temp_sel = READ_MEM.d [tss:new_cpl*8+8]         // read SSn from the TSS
        temp_SS_desc = READ_DESCRIPTOR (temp_sel, ss_chk)
    }

    return (temp_RSP:temp_SS_desc)


////////////////////////////////////////////////////////////////////////////
// READ_BIT_ARRAY  // Read 1 bit from a bit array in memory
////////////////////////////////////////////////////////////////////////////

usage:
    temp_value = READ_BIT_ARRAY ([mem], bit_number)

definition:

    temp_BYTE = READ_MEM.b [mem + (bit_number SHR 3)]
                                // read the byte containing the bit

    temp_BIT = temp_BYTE SHR (bit_number & 7)
                                // shift the requested bit position into bit 0

    return (temp_BIT & 0x01)    // return '0' or '1'

////////////////////////////////////////////////////////////////////////////
//  Shadow Stack Functions
////////////////////////////////////////////////////////////////////////////

define SSTK_ENABLED      = (CR4.CET) && (CR0.PE) && (!EFLAGS.VM)
define SSTK_USER_ENABLED = SSTK_ENABLED && (CPL==3) && (U_CET.SH_STK_EN)
define SSTK_SUPV_ENABLED = SSTK_ENABLED && (CPL <3) && (S_CET.SH_STK_EN)

bool ShadowStacksEnabled (privLevel)
IF ( SSTK_ENABLED &&
   (( privLevel == 3) && U_CET.SH_STK_EN) ||
   (( privLevel <  3) && S_CET.SH_STK_EN))
   RETURN (TRUE)
ELSE
   RETURN (FALSE)

////////////////////////////////////////////////////////////////////////////
//  SSTK_READ_MEM   // read shadow stack memory
//       Usage: temp = SSTK_READ_MEM.x [linear_addr]
//                     where x is either d or q (4 or 8 bytes)
////////////////////////////////////////////////////////////////////////////

IF (PAGING_ENABLED) && (
       ( the linear address maps to a not-present page )
    || ( the linear address maps to a non-shadow stack page )
    || ( the access is user-mode &&
```

```
         the linear address maps to a supervisor shadow stack page )
    || ( the access is supervisor-mode &&
         the linear address maps to a user shadow stack page ))
   EXCEPTION [PF(error_code)] // page fault, with the SS (shadow stack) bit
                              // set in error_code and the present and
                              // protection violation bits as appropriate
temp_data.x = memory [linear_addr].x
RETURN  (temp_data)

////////////////////////////////////////////////////////////////////////////
//  SSTK_WRITE_MEM   // write shadow stack memory
//    Usage: SSTK_WRITE_MEM.x [linear_addr] = temp.x
//                      where x is either d or q (4 or 8 bytes)
////////////////////////////////////////////////////////////////////////////

IF (PAGING_ENABLED) && (
       ( the linear address maps to a not-present page )
    || ( the linear address maps to a non-shadow stack page )
    || ( the access is user-mode &&
         the linear address maps to a supervisor shadow stack page )
    || ( the access is supervisor-mode &&
         the linear address maps to a user shadow stack page ))
   EXCEPTION [PF(error_code)] // page fault, w/ the SS (shadow stack) bit
                              // set in error_code and the present and
                              // protection violation bits as appropriate
memory [linear_addr].x = temp.x

////////////////////////////////////////////////////////////////////////////
// SET_SSTK_TOKEN_BUSY (new_SSP)
//       Checks shadow stack token and if valid set the token's busy bit
// Usage: SET_SSTK_TOKEN_BUSY (new_SSP)
////////////////////////////////////////////////////////////////////////////

   IF (new_SSP[2:0] != 0)            // new SSP must be 8-byte aligned
         EXCEPTION [#GP(0)]
   // check shadow stack token and set busy
   bool FAULT = FALSE
   < start atomic section >
   temp_Token = SSTK_READ_MEM.q [new_SSP] // fetch token with locked read
   IF ((!64-bit mode) && (temp_token[63:32] != 0))
      FAULT = TRUE                 // address in token must be <4GB
                                   // in legacy/compatibility mode
   IF ((temp_Token AND 0x01) != 0)
      FAULT = TRUE                 // token busy bit must be 0
   IF ((temp_Token AND ~0x01) != new_SSP)
      FAULT = TRUE                 // address in token must match new SSP
   IF (!FAULT)
      temp_Token = temp_Token OR 0x01 // if no faults, set token busy bit
   SSTK_WRITE_MEM.q [new_SSP] = temp_Token // write token and unlock
   < end atomic section >
   IF (FAULT)
```

```
EXCEPTION [#GP(0)]
```

3 General-Purpose Instruction Reference

This chapter describes the function, mnemonic syntax, opcodes, affected flags, and possible exceptions generated by the general-purpose instructions. General-purpose instructions are used in basic software execution. Most of these instructions load, store, or operate on data located in the general-purpose registers (GPRs), in memory, or in both. The remaining instructions are used to alter the sequential flow of the program by branching to other locations within the program, or to entirely different programs. With the exception of the MOVD, MOVMSKPD and MOVMSKPS instructions, which operate on MMX/XMM registers, the instructions within the category of general-purpose instructions do not operate on any other register set.

Most general-purpose instructions are supported in all hardware implementations of the AMD64 architecture. However, some instructions in this group are optional and support must be determined by testing processor feature flags using the CPUID instruction. These instructions are listed in Table 3-1, along with the CPUID function, register and bit used to test for the presence of the instruction.

Table 3-1. Instruction Support Indicated by CPUID Feature Bits

Instruction	CPUID Function(s)	Register[Bit]	Feature Flag
ADCX, ADOX	0000_0007h (ECX=0)	EBX[19]	ADX
Bit Manipulation Instructions - group 1	0000_0007h (ECX=0)	EBX[3]	BMI1
Bit Manipulation Instructions - group 2	0000_0007h (ECX=0)	EBX[8]	BMI2
CLFLOPT	0000_0007_0	EBX[23]	CLFLOPT
CLWB	0000_0007h (ECX=0)	EBX[24]	CLWB
CLZERO	8000_0008h	EBX[0]	CLZERO
CMPXCHG8B	0000_0001h, 8000_0001h	EDX[8]	CMPXCHG8B
CMPXCHG16B	0000_0001h	ECX[13]	CMPXCHG16B
CMOV*cc* (Conditional Moves)	0000_0001h, 8000_0001h	EDX[15]	CMOV
CLFLUSH	0000_0001h	EDX[19]	CLFSH
CRC32	0000_0001h	ECX[20]	SSE42
LAHF, SAHF	8000_0001h	ECX[0]	LahfSahf
LZCNT	8000_0001h	ECX[5]	ABM
Long Mode and Long Mode instructions	8000_0001h	EDX[29]	LM
MCOMMIT	8000_0008h	EBX[8]	MCOMMIT
MFENCE, LFENCE	0000_0001h	EDX[26]	SSE2
MONITORX, MWAITX	8000_0001h	ECX[29]	MONITORX
MOVBE	0000_0001h	ECX[22]	MOVBE

Table 3-1. Instruction Support Indicated by CPUID Feature Bits (continued)

Instruction	CPUID Function(s)	Register[Bit]	Feature Flag
MOVD[1]	0000_0001h, 8000_0001h	EDX[23]	MMX
	0000_0001h	EDX[26]	SSE2
MOVNTI	0000_0001h	EDX[26]	SSE2
POPCNT	0000_0001h	ECX[23]	POPCNT
PREFETCH / PREFETCHW[2]	8000_0001h	ECX[8]	3DNowPrefetch
		EDX[29]	LM
		EDX[31]	3DNow
RDFSBASE, RDGSBASE WRFSBASE, WRGSBASE	0000_0007h (ECX=0)	EBX[0]	FSGSBASE
RDPRU	8000_0008h	EBX[4]	RDPRU
RDRAND	0000_0001h	ECX[30]	RDRAND
RDSEED	0000_0007h (ECX=0)	EBX[18]	RDSEED
RDPID	0000_0007h (ECX=0)	ECX[22]	RDPID
SFENCE	0000_0001h	EDX[25]	SSE
Trailing Bit Manipulation Instructions	8000_0001h	ECX[21]	TBM

Notes:

1. The MOVD variant that moves values to or from MMX registers is part of the MMX subset; the MOVD variant that moves data to or from XMM registers is part of the SSE2 subset.
2. Instruction is supported if any one of the listed feature flags is set.

For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 165. For a comprehensive list of all instruction support feature flags, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

The general-purpose instructions can be used in legacy mode or 64-bit long mode. Compilation of general-purpose programs for execution in 64-bit long mode offers three primary advantages: access to the eight extended, 64-bit general-purpose registers (for a register set consisting of GPR0–GPR15), access to the 64-bit virtual address space, and access to the RIP-relative addressing mode.

For further information about the general-purpose instructions and register resources, see:

- “General-Purpose Programming” in Volume 1.
- “Summary of Registers and Data Types” on page 38.
- “Notation” on page 53.
- “Instruction Prefixes” on page 5.
- Appendix B, “General-Purpose Instructions in 64-Bit Mode.” In particular, see “General Rules for 64-Bit Mode” on page 555.

AAA ASCII Adjust After Addition

Adjusts the value in the AL register to an unpacked BCD value. Use the AAA instruction after using the ADD instruction to add two unpacked BCD numbers.

The instruction is coded without explicit operands:

```
AAA
```

If the value in the lower nibble of AL is greater than 9 or the AF flag is set to 1, the instruction increments the AH register, adds 6 to the AL register, and sets the CF and AF flags to 1. Otherwise, it does not change the AH register and clears the CF and AF flags to 0. In either case, AAA clears bits 7:4 of the AL register, leaving the correct decimal digit in bits 3:0.

This instruction also makes it possible to add ASCII numbers without having to mask off the upper nibble ‘3’.

MXCSR Flags Affected

Using this instruction in 64-bit mode generates an invalid-opcode exception.

Mnemonic	Opcode	Description
AAA	37	Create an unpacked BCD number. (Invalid in 64-bit mode.)

Related Instructions

AAD, AAM, AAS

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	U	M	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	This instruction was executed in 64-bit mode.

AAD ASCII Adjust Before Division

Converts two unpacked BCD digits in the AL (least significant) and AH (most significant) registers to a single binary value in the AL register.

The instruction is coded without explicit operands:

```
AAD
```

The instruction performs the following operation on the contents of AL and AH using the formula:

```
AL = ((10d * AH) + (AL))
```

After the conversion, AH is cleared to 00h.

In most modern assemblers, the AAD instruction adjusts from base-10 values. However, by coding the instruction directly in binary, it can adjust from any base specified by the immediate byte value (*ib*) suffixed onto the D5h opcode. For example, code D508h for octal, D50Ah for decimal, and D50Ch for duodecimal (base 12).

Using this instruction in 64-bit mode generates an invalid-opcode exception.

Mnemonic	Opcode	Description
AAD	D5 0A	Adjust two BCD digits in AL and AH. (Invalid in 64-bit mode.)
(None)	D5 *ib*	Adjust two BCD digits to the immediate byte base. (Invalid in 64-bit mode.)

Related Instructions

AAA, AAM, AAS

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				M	M	U	M	U
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	This instruction was executed in 64-bit mode.

AAM ASCII Adjust After Multiply

Converts the value in the AL register from binary to two unpacked BCD digits in the AH (most significant) and AL (least significant) registers.

The instruction is coded without explicit operands:

```
AAM
```

The instruction performs the following operation on the contents of AL and AH using the formula:

```
AH = (AL/10d)
AL = (AL mod 10d)
```

In most modern assemblers, the AAM instruction adjusts to base-10 values. However, by coding the instruction directly in binary, it can adjust to any base specified by the immediate byte value (*ib*) suffixed onto the D4h opcode. For example, code D408h for octal, D40Ah for decimal, and D40Ch for duodecimal (base 12).

Using this instruction in 64-bit mode generates an invalid-opcode exception.

Mnemonic	Opcode	Description
AAM	D4 0A	Create a pair of unpacked BCD values in AH and AL. (Invalid in 64-bit mode.)
(None)	D4 *ib*	Create a pair of unpacked values to the immediate byte base. (Invalid in 64-bit mode.)

Related Instructions

AAA, AAD, AAS

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				M	M	U	M	U
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M. Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Divide by zero, #DE	X	X	X	8-bit immediate value was 0.
Invalid opcode, #UD			X	This instruction was executed in 64-bit mode.

AAS ASCII Adjust After Subtraction

Adjusts the value in the AL register to an unpacked BCD value. Use the AAS instruction after using the SUB instruction to subtract two unpacked BCD numbers.

The instruction is coded without explicit operands:

```
AAS
```

If the value in AL is greater than 9 or the AF flag is set to 1, the instruction decrements the value in AH, subtracts 6 from the AL register, and sets the CF and AF flags to 1. Otherwise, it clears the CF and AF flags and the AH register is unchanged. In either case, the instruction clears bits 7:4 of the AL register, leaving the correct decimal digit in bits 3:0.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

Mnemonic	Opcode	Description
AAS	3F	Create an unpacked BCD number from the contents of the AL register. (Invalid in 64-bit mode.)

Related Instructions

AAA, AAD, AAM

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	U	M	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	This instruction was executed in 64-bit mode.

ADC Add with Carry

Adds the carry flag (CF), the value in a register or memory location (first operand), and an immediate value or the value in a register or memory location (second operand), and stores the result in the first operand location.

The instruction has two operands:

ADC *dest*, *src*

The instruction cannot add two memory operands. The CF flag indicates a pending carry from a previous addition operation. The instruction sign-extends an immediate value to the length of the destination register or memory location.

This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a carry in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result.

Use the ADC instruction after an ADD instruction as part of a multibyte or multiword addition.

The forms of the ADC instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
ADC AL, *imm8*	14 *ib*	Add *imm8* to AL + CF.
ADC AX, *imm16*	15 *iw*	Add *imm16* to AX + CF.
ADC EAX, *imm32*	15 *id*	Add *imm32* to EAX + CF.
ADC RAX, *imm32*	15 *id*	Add sign-extended *imm32* to RAX + CF.
ADC *reg/mem8*, *imm8*	80 /2 *ib*	Add *imm8* to *reg/mem8* + CF.
ADC *reg/mem16*, *imm16*	81 /2 *iw*	Add *imm16* to *reg/mem16* + CF.
ADC *reg/mem32*, *imm32*	81 /2 *id*	Add *imm32* to *reg/mem32* + CF.
ADC *reg/mem64*, *imm32*	81 /2 *id*	Add sign-extended *imm32* to *reg/mem64* + CF.
ADC *reg/mem16*, *imm8*	83 /2 *ib*	Add sign-extended *imm8* to *reg/mem16* + CF.
ADC *reg/mem32*, *imm8*	83 /2 *ib*	Add sign-extended *imm8* to *reg/mem32* + CF.
ADC *reg/mem64*, *imm8*	83 /2 *ib*	Add sign-extended *imm8* to *reg/mem64* + CF.
ADC *reg/mem8*, *reg8*	10 */r*	Add *reg8* to *reg/mem8* + CF
ADC *reg/mem16*, *reg16*	11 */r*	Add *reg16* to *reg/mem16* + CF.
ADC *reg/mem32*, *reg32*	11 */r*	Add *reg32* to *reg/mem32* + CF.
ADC *reg/mem64*, *reg64*	11 */r*	Add *reg64* to *reg/mem64* + CF.
ADC *reg8*, *reg/mem8*	12 */r*	Add *reg/mem8* to *reg8* + CF.
ADC *reg16*, *reg/mem16*	13 */r*	Add *reg/mem16* to *reg16* + CF.

Mnemonic	Opcode	Description
ADC *reg32*, *reg/mem32*	13 */r*	Add *reg/mem32* to *reg32* + CF.
ADC *reg64*, *reg/mem64*	13 */r*	Add *reg/mem64* to *reg64* + CF.

Related Instructions

ADD, SBB, SUB

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

ADCX Unsigned ADD with Carry Flag

Adds the value in a register (first operand) with a register or memory (second operand) and the carry flag, and stores the result in the first operand location. This instruction sets the CF based on the unsigned addition. This instruction is useful in multi-precision addition algorithms.

This is an ADX instructions. Support for this instruction is indicated by CPUID Fn0000_0007_EBX[ADX]=1.

Mnemonic	Opcode	Description
ADCX *reg32*, *reg/mem32*	66 0F 38 F6 /r	Unsigned add with carryflag
ADCX *reg64*, *reg/mem64*	66 0F 38 F6 /r	Unsigned add with carry flag.

Related Instructions

ADOX

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
																M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0
Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank.Undefined flags are U.*																

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID Fn0000_0007_EBX[ADX] = 0.
	X		X	Lock prefix (F0h) preceding opcode.

ADD Signed or Unsigned Add

Adds the value in a register or memory location (first operand) and an immediate value or the value in a register or memory location (second operand), and stores the result in the first operand location.

The instruction has two operands:

ADD *dest*, *src*

The instruction cannot add two memory operands. The instruction sign-extends an immediate value to the length of the destination register or memory operand.

This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a carry in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result.

The forms of the ADD instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
ADD AL, *imm8*	04 *ib*	Add *imm8* to AL.
ADD AX, *imm16*	05 *iw*	Add *imm16* to AX.
ADD EAX, *imm32*	05 *id*	Add *imm32* to EAX.
ADD RAX, *imm32*	05 *id*	Add sign-extended *imm32* to RAX.
ADD *reg/mem8*, *imm8*	80 /0 *ib*	Add *imm8* to *reg/mem8*.
ADD *reg/mem16*, *imm16*	81 /0 *iw*	Add *imm16* to *reg/mem16*
ADD *reg/mem32*, *imm32*	81 /0 *id*	Add *imm32* to *reg/mem32*.
ADD *reg/mem64*, *imm32*	81 /0 *id*	Add sign-extended *imm32* to *reg/mem64*.
ADD *reg/mem16*, *imm8*	83 /0 *ib*	Add sign-extended *imm8* to *reg/mem16*
ADD *reg/mem32*, *imm8*	83 /0 *ib*	Add sign-extended *imm8* to *reg/mem32.*
ADD *reg/mem64*, *imm8*	83 /0 *ib*	Add sign-extended *imm8* to *reg/mem64.*
ADD *reg/mem8, reg8*	00 */r*	Add *reg8* to *reg/mem8*.
ADD *reg/mem16*, *reg16*	01 */r*	Add *reg16* to *reg/mem16.*
ADD *reg/mem32*, *reg32*	01 */r*	Add *reg32* to *reg/mem32.*
ADD *reg/mem64*, *reg64*	01 */r*	Add *reg64* to *reg/mem64.*
ADD *reg8*, *reg/mem8*	02 */r*	Add *reg/mem8* to *reg8*.
ADD *reg16*, *reg/mem16*	03 */r*	Add *reg/mem16* to *reg16*.
ADD *reg32*, *reg/mem32*	03 */r*	Add *reg/mem32* to *reg32*.
ADD *reg64*, *reg/mem64*	03 */r*	Add *reg/mem64* to *reg64*.

Related Instructions

ADC, SBB, SUB

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

ADOX Unsigned ADD with Overflow Flag

Adds the value in a register (first operand) with a register or memory (second operand) and the overflow flag, and stores the result in the first operand location. This instruction sets the OF based on the unsigned addition and whether there is a carry out. This instruction is useful in multi-precision addition algorithms.

This is an ADX instructions. Support for this instruction is indicated by CPUID Fn0000_0007_EBX[ADX]=1.

Mnemonic	Opcode	Description
ADOX *reg32*, *reg/mem32*	F3 0F 38 F6 /r	Unsigned add with overflow flag
ADOX *reg64*, *reg/mem64*	F3 0F 38 F6 /r	Unsigned add with overflow flag.

Related Instructions

ADCX

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M								
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank.Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID Fn0000_0007_EBX[ADX] = 0.
	X		X	Lock prefix (F0h) preceding opcode.

AND Logical AND

Performs a bit-wise logical and operation on the value in a register or memory location (first operand) and an immediate value or the value in a register or memory location (second operand), and stores the result in the first operand location. Both operands cannot be memory locations.

The instruction has two operands:

AND *dest*, *src*

The instruction sets each bit of the result to 1 if the corresponding bit of both operands is set; otherwise, it clears the bit to 0. The following table shows the truth table for the logical and operation:

X	Y	X and Y
0	0	0
0	1	0
1	0	0
1	1	1

The forms of the AND instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
AND AL, *imm8*	24 *ib*	and the contents of AL with an immediate 8-bit value and store the result in AL.
AND AX, *imm16*	25 *iw*	and the contents of AX with an immediate 16-bit value and store the result in AX.
AND EAX, *imm32*	25 *id*	and the contents of EAX with an immediate 32-bit value and store the result in EAX.
AND RAX, *imm32*	25 *id*	and the contents of RAX with a sign-extended immediate 32-bit value and store the result in RAX.
AND *reg/mem8*, *imm8*	80 /4 *ib*	and the contents of *reg/mem8* with *imm8*.
AND *reg/mem16*, *imm16*	81 /4 *iw*	and the contents of *reg/mem16* with *imm16*.
AND *reg/mem32*, *imm32*	81 /4 *id*	and the contents of *reg/mem32* with *imm32*.
AND *reg/mem64*, *imm32*	81 /4 *id*	and the contents of *reg/mem64* with sign-extended *imm32*.
AND *reg/mem16*, *imm8*	83 /4 *ib*	and the contents of *reg/mem16* with a sign-extended 8-bit value.
AND *reg/mem32*, *imm8*	83 /4 *ib*	and the contents of *reg/mem32* with a sign-extended 8-bit value.
AND *reg/mem64*, *imm8*	83 /4 *ib*	and the contents of *reg/mem64* with a sign-extended 8-bit value.

Mnemonic	Opcode	Description
AND *reg/mem8*, *reg8*	20 /r	and the contents of an 8-bit register or memory location with the contents of an 8-bit register.
AND *reg/mem16*, *reg16*	21 /r	and the contents of a 16-bit register or memory location with the contents of a 16-bit register.
AND *reg/mem32*, *reg32*	21 /r	and the contents of a 32-bit register or memory location with the contents of a 32-bit register.
AND *reg/mem64*, *reg64*	21 /r	and the contents of a 64-bit register or memory location with the contents of a 64-bit register.
AND *reg8*, *reg/mem8*	22 /r	and the contents of an 8-bit register with the contents of an 8-bit memory location or register.
AND *reg16*, *reg/mem16*	23 /r	and the contents of a 16-bit register with the contents of a 16-bit memory location or register.
AND *reg32*, *reg/mem32*	23 /r	and the contents of a 32-bit register with the contents of a 32-bit memory location or register.
AND *reg64*, *reg/mem64*	23 /r	and the contents of a 64-bit register with the contents of a 64-bit memory location or register.

Related Instructions

TEST, OR, NOT, NEG, XOR

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	M	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonix‘cal.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

ANDN Logical And-Not

Performs a bit-wise logical and of the second source operand and the one's complement of the first source operand and stores the result into the destination operand.

This instruction has three operands:

ANDN *dest*, *src1*, *src2*

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64-bit; if VEX.W is 0, the operand size is 32-bit. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination operand (*dest*) is always a general purpose register.

The first source operand (*src1*) is a general purpose register and the second source operand (*src2*) is either a general purpose register or a memory operand.

This instruction implements the following operation:

```
not tmp, src1
and dest, tmp, src2
```

The flags are set according to the result of the and pseudo-operation.

The ANDN instruction is a BMI1 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI1] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
ANDN *reg32*, *reg32*, *reg/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\text{src1}}$.0.00	F2 /r
ANDN *reg64, reg64, reg/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\text{src1}}$.0.00	F2 /r

Related Instructions

BEXTR, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Real	Virtual 80806	Protected	Cause of Exception
Invalid opcode, #UD	X	X		BMI instructions are only recognized in protected mode.
			X	BMI instructions are not supported as indicated by CPUID Fn0000_0007_EBX_x0[BMI] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BEXTR (register form) Bit Field Extract

Extracts a contiguous field of bits from the first source operand, as specified by the control field setting in the second source operand and puts the extracted field into the least significant bit positions of the destination. The remaining bits in the destination register are cleared to 0.

This instruction has three operands:

BEXTR *dest*, *src*, *cntl*

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64-bit; if VEX.W is 0, the operand size is 32-bit. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is either a general purpose register or a memory operand.

The control (*cntl*) operand is a general purpose register that provides two fields describing the range of bits to extract:

- *lsb_index* (in bits 7:0)—specifies the index of the least significant bit of the field
- *length* (in bits 15:8)—specifies the number of bits in the field.

The position of the extracted field can be expressed as:

[*lsb_ index* + *length* – 1] : [*lsb_index*]

For example, if the *lsb_index* is 7 and *length* is 5, then bits 11:7 of the source will be copied to bits 4:0 of the destination, with the rest of the destination being zero-filled. Zeros are provided for any bit positions in the specified range that lie beyond the most significant bit of the source operand. A length value of zero results in all zeros being written to the destination.

This form of the BEXTR instruction is a BMI1 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI1] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
BEXTR *reg32*, *reg/mem32*, *reg32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\text{cntl}}$.0.00	F7 /r
BEXTR *reg64, reg/mem64, reg64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\text{cntl}}$.0.00	F7 /r

Related Instructions

ANDN, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				U	M	U	U	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI instructions are only recognized in protected mode.
			X	BMI instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BEXTR (immediate form) Bit Field Extract

Extracts a contiguous field of bits from the first source operand, as specified by the control field setting in the second source operand and puts the extracted field into the least significant bit positions of the destination. The remaining bits in the destination register are cleared to 0.

This instruction has three operands:

BEXTR *dest*, *src*, *cntl*

In 64-bit mode, the operand size is determined by the value of XOP.W. If XOP.W is 1, the operand size is 64-bit; if XOP.W is 0, the operand size is 32-bit. In 32-bit mode, XOP.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is either a general purpose register or a memory operand.

The control (*cntl*) operand is a 32-bit immediate value that provides two fields describing the range of bits to extract:

- *lsb_index* (in immediate operand bits 7:0)—specifies the index of the least significant bit of the field
- *length* (in immediate operand bits 15:8)—specifies the number of bits in the field.

The position of the extracted field can be expressed as:

[*lsb_ index* + *length* – 1] : [*lsb_index*]

For example, if the *lsb_index* is 7 and *length* is 5, then bits 11:7 of the source will be copied to bits 4:0 of the destination, with the rest of the destination being zero-filled. Zeros are provided for any bit positions in the specified range that lie beyond the most significant bit of the source operand. A length value of zero results in all zeros being written to the destination.

This form of the BEXTR instruction is a TBM instruction. Support for this instruction is indicated by CPUID Fn8000_0001_ECX[TBM] =1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
BEXTR *reg32*, *reg/mem32*, *imm32*	8F	$\overline{\text{RXB}}$.0A	0.1111.0.00	10 /r /id
BEXTR *reg64, reg/mem64, imm32*	8F	$\overline{\text{RXB}}$.0A	1.1111.0.00	10 /r /id

Related Instructions

ANDN, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				U	M	U	U	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		TBM instructions are only recognized in protected mode.
			X	TBM instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[TBM] = 0.
			X	XOP.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BLCFILL Fill From Lowest Clear Bit

Finds the least significant zero bit in the source operand, clears all bits below that bit to 0 and writes the result to the destination. If there is no zero bit in the source operand, the destination is written with all zeros.

This instruction has two operands:

BLCFILL *dest*, *src*

In 64-bit mode, the operand size is determined by the value of XOP.W. If XOP.W is 1, the operand size is 64-bit; if XOP.W is 0, the operand size is 32-bit. In 32-bit mode, XOP.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is a general purpose register or a memory operand.

The BLCFILL instruction effectively performs a bit-wise logical and of the source operand and the result of incrementing the source operand by 1 and stores the result to the destination register:

```
add tmp, src, 1
and dest,tmp, src
```

The value of the carry flag of rFLAGS is generated according to the result of the add pseudo-instruction and the remaining arithmetic flags are generated by the and pseudo-instruction.

The BLCFILL instruction is a TBM instruction. Support for this instruction is indicated by CPUID Fn8000_0001_ECX[TBM] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
BLCFILL *reg32*, *reg/mem32*	8F	$\overline{\text{RXB}}$.09	0.$\overline{\text{dest}}$.0.00	01 /1
BLCFILL *reg64, reg/mem64*	8F	$\overline{\text{RXB}}$.09	1.$\overline{\text{dest}}$.0.00	01 /1

Related Instructions

ANDN, BEXTR, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		TBM instructions are only recognized in protected mode.
			X	TBM instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[TBM] = 0.
			X	XOP.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BLCI Isolate Lowest Clear Bit

Finds the least significant zero bit in the source operand, sets all other bits to 1 and writes the result to the destination. If there is no zero bit in the source operand, the destination is written with all ones.

This instruction has two operands:

BLCI *dest*, *src*

In 64-bit mode, the operand size is determined by the value of XOP.W. If XOP.W is 1, the operand size is 64-bit; if XOP.W is 0, the operand size is 32-bit. In 32-bit mode, XOP.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is a general purpose register or a memory operand.

The BLCI instruction effectively performs a bit-wise logical `or` of the source operand and the inverse of the result of incrementing the source operand by 1, and stores the result to the destination register:

```
add tmp,  src, 1
not tmp,  tmp
or  dest, tmp, src
```

The value of the carry flag of rFLAGS is generated according to the result of the `add` pseudo-instruction and the remaining arithmetic flags are generated by the `or` pseudo-instruction.

The BLCI instruction is a TBM instruction. Support for this instruction is indicated by CPUID Fn8000_0001_ECX[TBM] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
BLCI *reg32*, *reg/mem32*	8F	$\overline{\text{RXB}}$.09	0.$\overline{\text{dest}}$.0.00	02 /6
BLCI *reg64, reg/mem64*	8F	$\overline{\text{RXB}}$.09	1.$\overline{\text{dest}}$.0.00	02 /6

Related Instructions

ANDN, BEXTR, BLCFILL, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		TBM instructions are only recognized in protected mode.
			X	TBM instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[TBM] = 0.
			X	XOP.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BLCIC Isolate Lowest Clear Bit and Complement

Finds the least significant zero bit in the source operand, sets that bit to 1, clears all other bits to 0 and writes the result to the destination. If there is no zero bit in the source operand, the destination is written with all zeros.

This instruction has two operands:

BLCIC dest, src

In 64-bit mode, the operand size is determined by the value of XOP.W. If XOP.W is 1, the operand size is 64-bit; if XOP.W is 0, the operand size is 32-bit. In 32-bit mode, XOP.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is a general purpose register or a memory operand.

The BLCIC instruction effectively performs a bit-wise logical and of the negation of the source operand and the result of incrementing the source operand by 1, and stores the result to the destination register:

```
add tmp1, src, 1
not tmp2, src
and dest, tmp1,tmp2
```

The value of the carry flag of rFLAGS is generated according to the result of the add pseudo-instruction and the remaining arithmetic flags are generated by the and pseudo-instruction.

The BLCIC instruction is a TBM instruction. Support for this instruction is indicated by CPUID Fn8000_0001_ECX[TBM] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
BLCIC *reg32*, *reg/mem32*	8F	$\overline{\text{RXB}}$.09	0.$\overline{\text{dest}}$.0.00	01 /5
BLCIC *reg64, reg/mem64*	8F	$\overline{\text{RXB}}$.09	1.$\overline{\text{dest}}$.0.00	01 /5

Related Instructions

ANDN, BEXTR, BLCFILL, BLCI, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		TBM instructions are only recognized in protected mode.
			X	TBM instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[TBM] = 0.
			X	XOP.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BLCMSK Mask From Lowest Clear Bit

Finds the least significant zero bit in the source operand, sets that bit to 1, clears all bits above that bit to 0 and writes the result to the destination. If there is no zero bit in the source operand, the destination is written with all ones.

This instruction has two operands:

BLCMSK dest, src

In 64-bit mode, the operand size is determined by the value of XOP.W. If XOP.W is 1, the operand size is 64-bit; if XOP.W is 0, the operand size is 32-bit. In 32-bit mode, XOP.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is a general purpose register or a memory operand.

The BLCMSK instruction effectively performs a bit-wise logical `xor` of the source operand and the result of incrementing the source operand by 1 and stores the result to the destination register:

```
add tmp1, src, 1
xor dest, tmp1,src
```

The value of the carry flag of rFLAGS is generated according to the result of the `add` pseudo-instruction and the remaining arithmetic flags are generated by the `xor` pseudo-instruction.

If the input is all ones, the output is a value with all bits set to 1.

The BLCMSK instruction is a TBM instruction. Support for this instruction is indicated by CPUID Fn8000_0001_ECX[TBM] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
BLCMSK *reg32*, *reg/mem32*	8F	$\overline{\text{RXB}}$.09	0.$\overline{\text{dest}}$.0.00	02 /1
BLCMSK *reg64, reg/mem64*	8F	$\overline{\text{RXB}}$.09	1.$\overline{\text{dest}}$.0.00	02 /1

Related Instructions

ANDN, BEXTR, BLCFILL, BLCI, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		TBM instructions are only recognized in protected mode.
			X	TBM instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[TBM] = 0.
			X	XOP.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BLCS Set Lowest Clear Bit

Finds the least significant zero bit in the source operand, sets that bit to 1 and writes the result to the destination. If there is no zero bit in the source operand, the source is copied to the destination (and CF in rFLAGS is set to 1).

This instruction has two operands:

BLCS dest, src

In 64-bit mode, the operand size is determined by the value of XOP.W. If XOP.W is 1, the operand size is 64-bit; if XOP.W is 0, the operand size is 32-bit. In 32-bit mode, XOP.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is a general purpose register or a memory operand.

The BLCS instruction effectively performs a bit-wise logical `or` of the source operand and the result of incrementing the source operand by 1, and stores the result to the destination register:

```
add tmp,  src, 1
or  dest, tmp, src
```

The value of the carry flag of rFLAGS is generated by the `add` pseudo-instruction and the remaining arithmetic flags are generated by the `or` pseudo-instruction.

The BLCS instruction is a TBM instruction. Support for this instruction is indicated by CPUID Fn8000_0001_ECX[TBM] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
BLCS *reg32*, *reg/mem32*	8F	$\overline{\text{RXB}}$.09	0.$\overline{\text{dest}}$.0.00	01 /3
BLCS *reg64*, *reg/mem64*	8F	$\overline{\text{RXB}}$.09	1.$\overline{\text{dest}}$.0.00	01 /3

Related Instructions

ANDN, BEXTR, BLCFILL, BLCI, BLCIC, BLCMSK, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		TBM instructions are only recognized in protected mode.
			X	TBM instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[TBM] = 0.
			X	XOP.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BLSFILL Fill From Lowest Set Bit

Finds the least significant one bit in the source operand, sets all bits below that bit to 1 and writes the result to the destination. If there is no one bit in the source operand, the destination is written with all ones.

This instruction has two operands:

BLSFILL dest, src

In 64-bit mode, the operand size is determined by the value of XOP.W. If XOP.W is 1, the operand size is 64-bit; if XOP.W is 0, the operand size is 32-bit. In 32-bit mode, XOP.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is a general purpose register or a memory operand.

The BLSFILL instruction effectively performs a bit-wise logical `or` of the source operand and the result of subtracting 1 from the source operand, and stores the result to the destination register:

```
sub tmp,  src, 1
or  dest, tmp, src
```

The value of the carry flag of rFLAGs is generated by the `sub` pseudo-instruction and the remaining arithmetic flags are generated by the `or` pseudo-instruction.

The BLSFILL instruction is a TBM instruction. Support for this instruction is indicated by CPUID Fn8000_0001_ECX[TBM] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
BLSFILL *reg32*, *reg/mem32*	8F	$\overline{\text{RXB}}$.09	0.$\overline{\text{dest}}$.0.00	01 /2
BLSFILL *reg64, reg/mem64*	8F	$\overline{\text{RXB}}$.09	1.$\overline{\text{dest}}$.0.00	01 /2

Related Instructions

ANDN, BEXTR, BLCFILL, BLCI, BLCIC, BLCMSK, BLCS, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		TBM instructions are only recognized in protected mode.
			X	TBM instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[TBM] = 0.
			X	XOP.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BLSI Isolate Lowest Set Bit

Clears all bits in the source operand except for the least significant bit that is set to 1 and writes the result to the destination. If the source is all zeros, the destination is written with all zeros.

This instruction has two operands:

```
BLSI dest, src
```

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64-bit; if VEX.W is 0, the operand size is 32-bit. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is either a general purpose register or a bit memory operand.

This instruction implements the following operation:

```
neg tmp, src1
and dst, tmp, src1
```

The value of the carry flag is generated by the `neg` pseudo-instruction and the remaining status flags are generated by the `and` pseudo-instruction.

The BLSI instruction is a BMI1 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI1] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
BLSI *reg32*, *reg/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{dest}$.0.00	F3 /3
BLSI *reg64, reg/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{dest}$.0.00	F3 /3

Related Instructions

ANDN, BEXTR, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI instructions are only recognized in protected mode.
			X	BMI instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BLSIC Isolate Lowest Set Bit and Complement

Finds the least significant bit that is set to 1 in the source operand, clears that bit to 0, sets all other bits to 1 and writes the result to the destination. If there is no one bit in the source operand, the destination is written with all ones.

This instruction has two operands:

BLSIC dest, src

In 64-bit mode, the operand size is determined by the value of XOP.W. If XOP.W is 1, the operand size is 64-bit; if XOP.W is 0, the operand size is 32-bit. In 32-bit mode, XOP.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is a general purpose register or a memory operand.

The BLSIC instruction effectively performs a bit-wise logical `or` of the inverse of the source operand and the result of subtracting 1 from the source operand, and stores the result to the destination register:

```
sub tmp1, src, 1
not tmp2, src
or  dest, tmp1, tmp2
```

The value of the carry flag of rFLAGS is generated by the `sub` pseudo-instruction and the remaining arithmetic flags are generated by the `or` pseudo-instruction.

The BLSR instruction is a TBM instruction. Support for this instruction is indicated by CPUID Fn8000_0001_ECX[TBM] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
BLSIC *reg32*, *reg/mem32*	8F	$\overline{\text{RXB}}$.09	0.$\overline{\text{dest}}$.0.00	01 /6
BLSIC *reg64, reg/mem64*	8F	$\overline{\text{RXB}}$.09	1.$\overline{\text{dest}}$.0.00	01 /6

Related Instructions

ANDN, BEXTR, BLCFILL, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		TBM instructions are only recognized in protected mode.
			X	TBM instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[TBM] = 0.
			X	XOP.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BLSMSK Mask From Lowest Set Bit

Forms a mask with bits set to 1 from bit 0 up to and including the least significant bit position that is set to 1 in the source operand and writes the mask to the destination. If the value of the source operand is zero, the destination is written with all ones.

This instruction has two operands:

BLSMSK *dest*, *src*

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64-bit; if VEX.W is 0, the operand size is 32-bit. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is always a general purpose register.

The source operand (*src*) is either a general purpose register or a memory operand and the destination operand (*dest*) is a general purpose register.

This instruction implements the operation:

```
sub tmp, src1, 1
xor dst, tmp, src1
```

The value of the carry flag is generated by the `sub` pseudo-instruction and the remaining status flags are generated by the `xor` pseudo-instruction.

If the input is zero, the output is a value with all bits set to 1. If this is considered a corner case input, software may test the carry flag to detect the zero input value.

The BLSMSK instruction is a BMI1 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI1] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
BLSMSK *reg32*, *reg/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{dest}}$.0.00	F3 /2
BLSMSK *reg64, reg/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{dest}}$.0.00	F3 /2

Related Instructions

ANDN, BEXTR, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI instructions are only recognized in protected mode.
			X	BMI instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BLSR Reset Lowest Set Bit

Clears the least-significant bit that is set to 1 in the input operand and writes the modified operand to the destination.

This instruction has two operands:

BLSR *dest*, *src*

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64-bit; if VEX.W is 0, the operand size is 32-bit. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is always a general purpose register.

The source operand (*src*) is either a general purpose register or a memory operand.

This instruction implements the operation:

```
sub tmp, src1, 1
and dst, tmp, src1
```

The value of the carry flag is generated by the `sub` pseudo-instruction and the remaining status flags are generated by the `and` pseudo-instruction.

The BLSR instruction is a BMI1 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI1] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
BLSR *reg32*, *reg/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{dest}}$.0.00	F3 /1
BLSR *reg64, reg/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{dest}}$.0.00	F3 /1

Related Instructions

ANDN, BEXTR, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI instructions are only recognized in protected mode.
			X	BMI instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

BOUND Check Array Bound

Checks whether an array index (first operand) is within the bounds of an array (second operand). The array index is a signed integer in the specified register. If the operand-size attribute is 16, the array operand is a memory location containing a pair of signed word-integers; if the operand-size attribute is 32, the array operand is a pair of signed doubleword-integers. The first word or doubleword specifies the lower bound of the array and the second word or doubleword specifies the upper bound.

The array index must be greater than or equal to the lower bound and less than or equal to the upper bound. If the index is not within the specified bounds, the processor generates a BOUND range-exceeded exception (#BR).

The bounds of an array, consisting of two words or doublewords containing the lower and upper limits of the array, usually reside in a data structure just before the array itself, making the limits addressable through a constant offset from the beginning of the array. With the address of the array in a register, this practice reduces the number of bus cycles required to determine the effective address of the array bounds.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

Mnemonic	Opcode	Description
BOUND *reg16*, *mem16&mem16*	62 */r*	Test whether a 16-bit array index is within the bounds specified by the two 16-bit values in mem16&mem16. (Invalid in 64-bit mode.)
BOUND *reg32*, *mem32&mem32*	62 */r*	Test whether a 32-bit array index is within the bounds specified by the two 32-bit values in mem32&mem32. (Invalid in 64-bit mode.)

Related Instructions

INT, INT3, INTO

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Bound range, #BR	X	X	X	The bound range was exceeded.
Invalid opcode, #UD	X	X	X	The source operand was a register.
			X	Instruction was executed in 64-bit mode.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit
General protection, #GP	X	X	X	A memory address exceeded a data segment limit.
			X	A null data segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

BSF Bit Scan Forward

Searches the value in a register or a memory location (second operand) for the least-significant set bit. If a set bit is found, the instruction clears the zero flag (ZF) and stores the index of the least-significant set bit in a destination register (first operand). If the second operand contains 0, the instruction sets ZF to 1 and does not change the contents of the destination register. The bit index is an unsigned offset from bit 0 of the searched value.

Mnemonic	Opcode	Description
BSF *reg16*, *reg/mem16*	0F BC /r	Bit scan forward on the contents of *reg/mem16*.
BSF *reg32*, *reg/mem32*	0F BC /r	Bit scan forward on the contents of *reg/mem32*.
BSF *reg64*, *reg/mem64*	0F BC /r	Bit scan forward on the contents of *reg/mem64*

Related Instructions

BSR

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	M	U	U	U
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

BSR Bit Scan Reverse

Searches the value in a register or a memory location (second operand) for the most-significant set bit. If a set bit is found, the instruction clears the zero flag (ZF) and stores the index of the most-significant set bit in a destination register (first operand). If the second operand contains 0, the instruction sets ZF to 1 and does not change the contents of the destination register. The bit index is an unsigned offset from bit 0 of the searched value.

Mnemonic	Opcode	Description
BSR *reg16*, *reg/mem16*	0F BD /r	Bit scan reverse on the contents of *reg/mem16*.
BSR *reg32*, *reg/mem32*	0F BD /r	Bit scan reverse on the contents of *reg/mem32*.
BSR *reg64*, *reg/mem64*	0F BD /r	Bit scan reverse on the contents of *reg/mem64*.

Related Instructions

BSF

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	M	U	U	U
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded the data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

BSWAP Byte Swap

Reverses the byte order of the specified register. This action converts the contents of the register from little endian to big endian or vice versa. In a doubleword, bits 7:0 are exchanged with bits 31:24, and bits 15:8 are exchanged with bits 23:16. In a quadword, bits 7:0 are exchanged with bits 63:56, bits 15:8 with bits 55:48, bits 23:16 with bits 47:40, and bits 31:24 with bits 39:32. A subsequent use of the BSWAP instruction with the same operand restores the original value of the operand.

The result of applying the BSWAP instruction to a 16-bit register is undefined. To swap the bytes of a 16-bit register, use the XCHG instruction and specify the respective byte halves of the 16-bit register as the two operands. For example, to swap the bytes of AX, use `XCHG AL, AH`.

Mnemonic	Opcode	Description
BSWAP *reg32*	0F C8 +*rd*	Reverse the byte order of *reg32*.
BSWAP *reg64*	0F C8 +*rq*	Reverse the byte order of *reg64*.

Related Instructions

XCHG

rFLAGS Affected

None

Exceptions

None

BT Bit Test

Copies a bit, specified by a bit index in a register or 8-bit immediate value (second operand), from a bit string (first operand), also called the bit base, to the carry flag (CF) of the rFLAGS register.

If the bit base operand is a register, the instruction uses the modulo 16, 32, or 64 (depending on the operand size) of the bit index to select a bit in the register.

If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string. If the bit index is in a register, the instruction selects a bit position relative to the bit base in the range -2^{63} to $+2^{63} - 1$ if the operand size is 64, -2^{31} to $+2^{31} - 1$, if the operand size is 32, and -2^{15} to $+2^{15} - 1$ if the operand size is 16. If the bit index is in an immediate value, the bit selected is that value modulo 16, 32, or 64, depending on operand size.

When the instruction attempts to copy a bit from memory, it accesses 2, 4, or 8 bytes starting from the specified memory address for 16-bit, 32-bit, or 64-bit operand sizes, respectively, using the following formula:

Effective Address + ($NumBytes_i$ * (BitOffset DIV $NumBits_{i*8}$))

When using this bit addressing mechanism, avoid referencing areas of memory close to address space holes, such as references to memory-mapped I/O registers. Instead, use a MOV instruction to load a register from such an address and use a register form of the BT instruction to manipulate the data.

Mnemonic	Opcode	Description
BT *reg/mem16*, *reg16*	0F A3 /r	Copy the value of the selected bit to the carry flag.
BT *reg/mem32, reg32*	0F A3 /r	Copy the value of the selected bit to the carry flag.
BT *reg/mem64, reg64*	0F A3 /r	Copy the value of the selected bit to the carry flag.
BT *reg/mem16*, *imm8*	0F BA /4 *ib*	Copy the value of the selected bit to the carry flag.
BT *reg/mem32, imm8*	0F BA /4 *ib*	Copy the value of the selected bit to the carry flag.
BT *reg/mem64, imm8*	0F BA /4 *ib*	Copy the value of the selected bit to the carry flag.

Related Instructions

BTC, BTR, BTS

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	U	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

BTC Bit Test and Complement

Copies a bit, specified by a bit index in a register or 8-bit immediate value (second operand), from a bit string (first operand), also called the bit base, to the carry flag (CF) of the rFLAGS register, and then complements (toggles) the bit in the bit string.

If the bit base operand is a register, the instruction uses the modulo 16, 32, or 64 (depending on the operand size) of the bit index to select a bit in the register.

If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string. If the bit index is in a register, the instruction selects a bit position relative to the bit base in the range -2^{63} to $+2^{63} - 1$ if the operand size is 64, -2^{31} to $+2^{31} - 1$, if the operand size is 32, and -2^{15} to $+2^{15} - 1$ if the operand size is 16. If the bit index is in an immediate value, the bit selected is that value modulo 16, 32, or 64, depending the operand size.

This instruction is useful for implementing semaphores in concurrent operating systems. Such an application should precede this instruction with the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
BTC *reg/mem16*, *reg16*	0F BB /r	Copy the value of the selected bit to the carry flag, then complement the selected bit.
BTC *reg/mem32, reg32*	0F BB /r	Copy the value of the selected bit to the carry flag, then complement the selected bit.
BTC *reg/mem64, reg64*	0F BB /r	Copy the value of the selected bit to the carry flag, then complement the selected bit.
BTC *reg/mem16*, *imm8*	0F BA /7 *ib*	Copy the value of the selected bit to the carry flag, then complement the selected bit.
BTC *reg/mem32, imm8*	0F BA /7 *ib*	Copy the value of the selected bit to the carry flag, then complement the selected bit.
BTC *reg/mem64, imm8*	0F BA /7 *ib*	Copy the value of the selected bit to the carry flag, then complement the selected bit.

Related Instructions

BT, BTR, BTS

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	U	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

BTR Bit Test and Reset

Copies a bit, specified by a bit index in a register or 8-bit immediate value (second operand), from a bit string (first operand), also called the bit base, to the carry flag (CF) of the rFLAGS register, and then clears the bit in the bit string to 0.

If the bit base operand is a register, the instruction uses the modulo 16, 32, or 64 (depending on the operand size) of the bit index to select a bit in the register.

If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string. If the bit index is in a register, the instruction selects a bit position relative to the bit base in the range -2^{63} to $+2^{63} - 1$ if the operand size is 64, -2^{31} to $+2^{31} - 1$, if the operand size is 32, and -2^{15} to $+2^{15} - 1$ if the operand size is 16. If the bit index is in an immediate value, the bit selected is that value modulo 16, 32, or 64, depending on the operand size.

This instruction is useful for implementing semaphores in concurrent operating systems. Such applications should precede this instruction with the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
BTR *reg/mem16*, *reg16*	0F B3 /r	Copy the value of the selected bit to the carry flag, then clear the selected bit.
BTR *reg/mem32, reg32*	0F B3 /r	Copy the value of the selected bit to the carry flag, then clear the selected bit.
BTR *reg/mem64, reg64*	0F B3 /r	Copy the value of the selected bit to the carry flag, then clear the selected bit.
BTR *reg/mem16*, *imm8*	0F BA /6 *ib*	Copy the value of the selected bit to the carry flag, then clear the selected bit.
BTR *reg/mem32, imm8*	0F BA /6 *ib*	Copy the value of the selected bit to the carry flag, then clear the selected bit.
BTR *reg/mem64, imm8*	0F BA /6 *ib*	Copy the value of the selected bit to the carry flag, then clear the selected bit.

Related Instructions

BT, BTC, BTS

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	U	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

BTS Bit Test and Set

Copies a bit, specified by bit index in a register or 8-bit immediate value (second operand), from a bit string (first operand), also called the bit base, to the carry flag (CF) of the rFLAGS register, and then sets the bit in the bit string to 1.

If the bit base operand is a register, the instruction uses the modulo 16, 32, or 64 (depending on the operand size) of the bit index to select a bit in the register.

If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string. If the bit index is in a register, the instruction selects a bit position relative to the bit base in the range -2^{63} to $+2^{63} - 1$ if the operand size is 64, -2^{31} to $+2^{31} - 1$, if the operand size is 32, and -2^{15} to $+2^{15} - 1$ if the operand size is 16. If the bit index is in an immediate value, the bit selected is that value modulo 16, 32, or 64, depending on the operand size.

This instruction is useful for implementing semaphores in concurrent operating systems. Such applications should precede this instruction with the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
BTS *reg/mem16*, *reg16*	0F AB /r	Copy the value of the selected bit to the carry flag, then set the selected bit.
BTS *reg/mem32, reg32*	0F AB /r	Copy the value of the selected bit to the carry flag, then set the selected bit.
BTS *reg/mem64, reg64*	0F AB /r	Copy the value of the selected bit to the carry flag, then set the selected bit.
BTS *reg/mem16*, *imm8*	0F BA /5 *ib*	Copy the value of the selected bit to the carry flag, then set the selected bit.
BTS *reg/mem32, imm8*	0F BA /5 *ib*	Copy the value of the selected bit to the carry flag, then set the selected bit.
BTS *reg/mem64, imm8*	0F BA /5 *ib*	Copy the value of the selected bit to the carry flag, then set the selected bit.

Related Instructions

BT, BTC, BTR

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	U	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

BZHI Zero High Bits

Copies bits, left to right, from the first source operand starting with the bit position specified by the second source operand (*index*), writes these bits to the destination, and clears all the bits in positions greater than or equal to *index*.

This instruction has three operands:

BZHI *dest*, *src, index*

In 64-bit mode, the operand size (*op_size*) is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register. The first source operand (*src*) is either a general purpose register or a memory operand. The second source operand is a general purpose register. Bits [7:0] of this register, treated as an unsigned 8-bit integer, specify the index of the most-significant bit of the first source operand to be copied to the corresponding bit of the destination. Bits [*op_size*-1:*index*] of the destination are cleared.

If the value of *index* is greater than or equal to the operand size, *index* is set to (*op_size*-1). In this case, the CF flag is set.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
BZHI *reg32*, *reg/mem32, reg32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{index}}$.0.00	F5 /r
BZHI *reg64, reg/mem64, reg64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{index}}$.0.00	F5 /r

Related Instructions

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

CALL (Near) Near Procedure Call

Pushes the offset of the next instruction onto the stack and branches to the target address, which contains the first instruction of the called procedure. The target operand can specify a register, a memory location, or a label. A procedure accessed by a near CALL is located in the same code segment as the CALL instruction.

If the CALL target is specified by a register or memory location, then a 16-, 32-, or 64-bit rIP is read from the operand, depending on the operand size. A 16- or 32-bit rIP is zero-extended to 64 bits.

If the CALL target is specified by a displacement, the signed displacement is added to the rIP (of the following instruction), and the result is truncated to 16, 32, or 64 bits, depending on the operand size. The signed displacement is 16 or 32 bits, depending on the operand size.

In all cases, the rIP of the instruction after the CALL is pushed on the stack, and the size of the stack push (16, 32, or 64 bits) depends on the operand size of the CALL instruction.

For near calls in 64-bit mode, the operand size defaults to 64 bits. The E8 opcode results in RIP = RIP + 32-bit signed displacement and the FF /2 opcode results in RIP = 64-bit offset from register or memory. No prefix is available to encode a 32-bit operand size in 64-bit mode.

At the end of the called procedure, RET is used to return control to the instruction following the original CALL. When RET is executed, the rIP is popped off the stack, which returns control to the instruction after the CALL.

See CALL (Far) for information on far calls—calls to procedures located outside of the current code segment. For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
CALL *rel16off*	E8 *iw*	Near call with the target specified by a 16-bit relative displacement.
CALL *rel32off*	E8 *id*	Near call with the target specified by a 32-bit relative displacement.
CALL *reg/mem16*	FF /2	Near call with the target specified by *reg/mem16*.
CALL *reg/mem32*	FF /2	Near call with the target specified by *reg/mem32*. (There is no prefix for encoding this in 64-bit mode.)
CALL *reg/mem64*	FF /2	Near call with the target specified by *reg/mem64*.

For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Action

```
// For function ShadowStacksEnabled()
// see "Pseudocode Definition" on page 57

CALLN_START:

IF (OPCODE == calln abs [mem] )     // CALLN, abs indirect
   temp_RIP = READ_MEM.z [mem]
ELSE                                // CALLN, rel/abs direct
   temp_RIP = z-sized instruction offset field, zero-extended to 64 bits

IF (OPCODE == calln rel )           // if relative, add offset to rIP
   temp_RIP = temp_RIP + RIP.v

IF  (stack is not large enough for a v-sized push)
         EXCEPTION[#SS(0)]

PUSH.v next_RIP

IF ((64BIT_MODE) && (temp_RIP is non-canonical) ||
   (!64BIT_MODE) && (temp_RIP > CS.limit))
         EXCEPTION[#GP(0)]

IF ((ShadowStacksEnabled(current CPL)) && (OPCODE != calln +0))
   {
   IF (v == 2)        // operand size = 16
      {
      SSTK_WRITE_MEM.d [SSP-4]  = next_IP
      SSP = SSP - 4
      }
   ELSEIF (v == 4)    // operand size = 32
      {
      SSTK_WRITE_MEM.d [SSP-4]  = next_EIP
      SSP = SSP - 4
      }
   ELSE // (v == 8)   // operand size = 64
      {
      SSTK_WRITE_MEM.q [SSP-8]  = next_RIP
      SSP = SSP - 8
      }
   } // end shadow stacks enabled

RIP = temp_RIP

EXIT
```

Related Instructions

CALL(Far), RET(Near), RET(Far)

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Alignment Check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
Page Fault, #PF		X	X	A page fault resulted from the execution of the instruction.

CALL (Far) Far Procedure Call

Pushes procedure linking information onto the stack and branches to the target address, which contains the first instruction of the called procedure. The operand specifies a target selector and offset.

The instruction can specify the target directly, by including the far pointer in the immediate and displacement fields of the instruction, or indirectly, by referencing a far pointer in memory. In 64-bit mode, only indirect far calls are allowed; executing a direct far call (opcode 9A) generates an undefined opcode exception. For both direct and indirect far calls, if the CALL (Far) operand-size is 16 bits, the instruction's operand is a 16-bit offset followed by a 16-bit selector. If the operand-size is 32 or 64 bits, the operand is a 32-bit offset followed by a 16-bit selector.

The target selector used by the instruction can be a code selector in all modes. Additionally, the target selector can reference a call gate in protected mode, or a task gate or TSS selector in legacy protected mode.

- *Target is a code selector*—The CS:rIP of the next instruction is pushed to the stack, using operand-size stack pushes. Then code is executed from the target CS:rIP. In this case, the target offset can only be a 16- or 32-bit value, depending on operand-size, and is zero-extended to 64 bits. No CPL change is allowed.
- *Target is a call gate*—The call gate specifies the actual target code segment and offset. Call gates allow calls to the same or more privileged code. If the target segment is at the same CPL as the current code segment, the CS:rIP of the next instruction is pushed to the stack.

  If the CALL (Far) changes privilege level, then a stack-switch occurs, using an inner-level stack pointer from the TSS. The CS:rIP of the next instruction is pushed to the new stack. If the mode is legacy mode and the param-count field in the call gate is non-zero, then up to 31 operands are copied from the caller's stack to the new stack. Finally, the caller's SS:rSP is pushed to the new stack.

  When calling through a call gate, the stack pushes are 16-, 32-, or 64-bits, depending on the size of the call gate. The size of the target rIP is also 16, 32, or 64 bits, depending on the size of the call gate. If the target rIP is less than 64 bits, it is zero-extended to 64 bits. Long mode only allows 64-bit call gates that must point to 64-bit code segments.
- *Target is a task gate or a TSS*—If the mode is legacy protected mode, then a task switch occurs. See “Hardware Task-Management in Legacy Mode” in volume 2 for details about task switches. Hardware task switches are not supported in long mode.

See CALL (Near) for information on near calls—calls to procedures located inside the current code segment. For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
CALL FAR *pntr16:16*	9A *cd*	Far call direct, with the target specified by a far pointer contained in the instruction. (Invalid in 64-bit mode.)
CALL FAR *pntr16:32*	9A *cp*	Far call direct, with the target specified by a far pointer contained in the instruction. (Invalid in 64-bit mode.)
CALL FAR *mem16:16*	FF /3	Far call indirect, with the target specified by a far pointer in memory.
CALL FAR *mem16:32*	FF /3	Far call indirect, with the target specified by a far pointer in memory.

Action

```
// For functions READ_DESCRIPTOR, READ_INNER_LEVEL_SP,
// ShadowStacksEnabled and SET_TOKEN_BUSY see "Pseudocode Definition"
// on page 57

CALLF_START:

IF (REAL_MODE)
   CALLF_REAL_OR_VIRTUAL    // CALLF real mode
ELSEIF (PROTECTED_MODE)
   CALLF_PROTECTED          // CALLF protected mode
ELSE // virtual mode
   CALLF_REAL_OR_VIRTUAL    // CALLF virtual mode


CALLF_REAL_OR_VIRTUAL:

IF (OPCODE == callf [mem] ) // CALLF real mode, indirect
   {
   temp_RIP = READ_MEM.z [mem]
   temp_CS  = READ_MEM.w [mem+Z]
   }
ELSE                        // CALLF real mode, direct
   {
   temp_RIP = z-sized instruction offset field, zero-extended to 64 bits
   temp_CS  = selector specified in the instruction
   }
PUSH.v old_CS
PUSH.v next_RIP

IF (temp_RIP > CS.limit)
      EXCEPTION [#GP(0)]

CS.sel  = temp_CS
CS.base = temp_CS SHL 4
RIP     = temp_RIP

EXIT  // end CALLF real or virtual
```

```
CALLF_PROTECTED:

IF (OPCODE == callf [mem])    // CALLF protected mode, indirect
   {
   temp_offset = READ_MEM.z [mem]
   temp_sel    = READ_MEM.w [mem+Z]
   }
ELSE                          // CALLF protected mode, direct
   {
   IF (64BIT_MODE)
         EXCEPTION [#UD]      // CALLF direct is illegal in 64-bit mode.
   temp_offset = z-sized instruction offset field, zero-extended to 64 bits
   temp_sel    = selector specified in the instruction
   }

temp_desc = READ_DESCRIPTOR (temp_sel, cs_chk)

IF (temp_desc.attr.type == 'available_tss')
   TASK_SWITCH // Using temp_sel as the target TSS
ELSEIF (temp_desc.attr.type == 'taskgate')
   TASK_SWITCH // Using the TSS selector in the task gate as the target TSS
ELSEIF (temp_desc.attr.type == 'callgate')
   CALLF_CALLGATE   // CALLF through callgate
ELSE // (temp_desc.attr.type == 'code')
   {           // the selector refers to a code descriptor
   temp_RIP = temp_offset   // the target RIP is the instruction offset field
   CS = temp_desc
   PUSH.v old_CS
   PUSH.v next_RIP

   IF ((!64BIT_MODE) && (temp_RIP > CS.limit))
         EXCEPTION [#GP(0)] // temp_RIP can't be non-canonical because its' a
                            // 16- or 32-bit offset, zero-extended to 64 bits
   RIP = temp_RIP

   IF ShadowStacksEnabled at current CPL
      {
      IF (v == 2)
         temp_LIP = CS.base + next_IP  // operand size = 16
      ELSEIF (v == 4)
         temp_LIP = CS.base + next_EIP // operand size = 32
      ELSE // (v == 8)
         temp_LIP = next_RIP           // operand size = 64

      IF EFER.LMA && (temp_desc.attr.L == 0) && (SSP[63:32] != 0)
         EXCEPTION [#GP(0)]       // SSP must be <4 GB

      Align SSP to 8B boundary, storing 4B of 0 if needed
      old_SSP = SSP
      SSTK_WRITE_MEM.q [SSP-16] = old_CS   // push CS, LIP, SSP
```

```
      SSTK_WRITE_MEM.q [SSP-8]  = temp_LIP // onto the shadow stack
      SSTK_WRITE_MEM.q [SSP]    = old_SSP
      SSP = SSP - 24
      }

   EXIT
   }   // end CALLF selector=code segment


CALLF_CALLGATE:

IF (LONG_MODE)   // the gate size controls the size of the stack pushes
   v=8-byte  // Long mode only uses 64-bit call gates, force 8-byte opsize
ELSEIF (temp_desc.attr.type == 'callgate32')
   v=4-byte  // Legacy mode, using a 32-bit call-gate, force 4-byte
ELSE   // (temp_desc.attr.type == 'callgate16')
   v=2-byte  // Legacy mode, using a 16-bit call-gate, force 2-byte opsize

// the target CS and RIP both come from the call gate.
temp_RIP = temp_desc.offset

IF (LONG_MODE)
   {                                    // read 2nd half of 16-byte call-gate
   temp_upper = READ_MEM.q [temp_sel+8] // to get upper 32 bits of target RIP
   IF (temp_upper's extended attribute bits != 0)
      EXCEPTION [#GP(temp_sel)]
   temp_RIP = tempRIP + (temp_upper SHL 32) // Concatenate both halves of RIP
   }

CS = READ_DESCRIPTOR (temp_desc.segment, callgate_check)

IF ((64BIT_MODE) && (temp_RIP is non-canonical) ||
   (!64BIT_MODE) && (temp_RIP > CS.limit))
         EXCEPTION[#GP(0)]

IF (CS.attr.conforming == 1)
   temp_CPL = CPL
ELSE
   temp_CPL = CS.attr.dpl

IF (CPL == temp_CPL)   // CALLF through gate, to same privilege
   {
   PUSH.v old_CS
   PUSH.v next_RIP
   RIP = temp_RIP

   IF (ShadowStacksEnabled at current CPL)
      {
      IF (v == 2)
         temp_LIP = CS.base + next_IP  // operand size = 16
      ELSEIF (v == 4)
```

```
         temp_LIP = CS.base + next_EIP // operand size = 32
      ELSE // (v == 8)
         temp_LIP = next_RIP           // operand size = 64

      IF ((EFER.LMA && (temp_desc.attr.L == 0)) && (SSP[63:32] != 0))
            EXCEPTION [#GP(0)]       // SSP must be <4 GB
      Align SSP to next 8B boundary, storing 4B of 0 if needed
      old_SSP = SSP
      SSTK_WRITE_MEM.q [SSP-24]  = old_CS   // push CS, LIP, SSP
      SSTK_WRITE_MEM.q [SSP-16]  = temp_LIP // onto the shadow stack
      SSTK_WRITE_MEM.q [SSP-8]   = old_SSP
      SSP = SSP - 24
      } // end shadow stacks enabled
   EXIT  // end CALLF through gate, to same privilege
   }
ELSE     // CALLF through gate, to more privilege
   {
   old_CPL = CPL
   CPL = temp_CPL
   temp_ist = 0      // CALLF doesn't use IST pointers.
   temp_SS_desc:temp_RSP = READ_INNER_LEVEL_SP(CPL,temp_ist)
   RSP.q = temp_RSP
   SS = temp_SS_desc

   PUSH.v old_SS  // #SS on this or next pushes use SS.sel as error code
   PUSH.v old_RSP

   IF (LEGACY_MODE) // Legacy-mode call gates have a param_count field
      temp_PARAM_COUNT = temp_desc.attr.param_count
      FOR (I=temp_PARAM_COUNT; I>0; I--)
         {
         temp_DATA = READ_MEM.v [old_SS:(old_RSP+I*V)]
         PUSH.v temp_DATA
         }

   PUSH.v old_CS
   PUSH.v next_RIP
   RIP = temp_RIP

   IF ((ShadowStacksEnabled at CPL=3) && (old_CPL == 3))
      PL3_SSP = SSP

   IF (ShadowStacksEnabled at new CPL)
      {
      old_SSP = SSP
      SSP     = PLn_SSP   // where n=new CPL

      SET_SSTK_TOKEN_BUSY(SSP) // check for valid token and set busy bit

      IF old_CPL != 3
         {
```

```
        // push CS,LIP,SSP onto sstk
        SSTK_WRITE_MEM.q [SSP-24] = old_CS   // push CS
        SSTK_WRITE_MEM.q [SSP-16] = temp_LIP // LIP and
        SSTK_WRITE_MEM.q [SSP-8]  = old_SSP  // SSP to the shadow stack
        SSP = SSP - 24
        }
    } // end shadow stacks enabled at new CPL
EXIT
} // end CALLF to more priv
```

Related Instructions

CALL (Near), RET (Near), RET (Far)

rFLAGS Affected

None, unless a task switch occurs, in which case all flags are modified.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The far CALL indirect opcode (FF /3) had a register operand.
			X	The far CALL direct opcode (9A) was executed in 64-bit mode.
Invalid TSS, #TS (selector)			X	As part of a stack switch, the target stack segment selector or rSP in the TSS was beyond the TSS limit.
			X	As part of a stack switch, the target stack segment selector in the TSS was a null selector.
			X	As part of a stack switch, the target stack selector's TI bit was set, but LDT selector was a null selector.
			X	As part of a stack switch, the target stack segment selector in the TSS was beyond the limit of the GDT or LDT descriptor table.
			X	As part of a stack switch, the target stack segment selector in the TSS contained a RPL that was not equal to its DPL.
			X	As part of a stack switch, the target stack segment selector in the TSS contained a DPL that was not equal to the CPL of the code segment selector.
			X	As part of a stack switch, the target stack segment selector in the TSS was not a writable segment.
Segment not present, #NP (selector)			X	The accessed code segment, call gate, task gate, or TSS was not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical, and no stack switch occurred.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS (selector)			X	After a stack switch, a memory access exceeded the stack segment limit or was non-canonical.
			X	As part of a stack switch, the SS register was loaded with a non-null segment selector and the segment was marked not present.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
General protection, #GP (selector)			X	The target code segment selector was a null selector.
			X	A code, call gate, task gate, or TSS descriptor exceeded the descriptor table limit.
			X	A segment selector's TI bit was set but the LDT selector was a null selector.
			X	The segment descriptor specified by the instruction was not a code segment, task gate, call gate or available TSS in legacy mode, or not a 64-bit code segment or a 64-bit call gate in long mode.
			X	The RPL of the non-conforming code segment selector specified by the instruction was greater than the CPL, or its DPL was not equal to the CPL.
			X	The DPL of the conforming code segment descriptor specified by the instruction was greater than the CPL.
			X	The DPL of the callgate, taskgate, or TSS descriptor specified by the instruction was less than the CPL, or less than its own RPL.
			X	The segment selector specified by the call gate or task gate was a null selector.
			X	The segment descriptor specified by the call gate was not a code segment in legacy mode, or not a 64-bit code segment in long mode.
			X	The DPL of the segment descriptor specified by the call gate was greater than the CPL.
			X	The 64-bit call gate's extended attribute bits were not zero.
			X	The TSS descriptor was found in the LDT.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

CBW
CWDE
CDQE
Convert to Sign-Extended

Copies the sign bit in the AL or eAX register to the upper bits of the rAX register. The effect of this instruction is to convert a signed byte, word, or doubleword in the AL or eAX register into a signed word, doubleword, or quadword in the rAX register. This action helps avoid overflow problems in signed number arithmetic.

The CDQE mnemonic is meaningful only in 64-bit mode.

Mnemonic	Opcode	Description
CBW	98	Sign-extend AL into AX.
CWDE	98	Sign-extend AX into EAX.
CDQE	98	Sign-extend EAX into RAX.

Related Instructions

CWD, CDQ, CQO

rFLAGS Affected

None

Exceptions

None

CWD
CDQ
CQO

Convert to Sign-Extended

Copies the sign bit in the rAX register to all bits of the rDX register. The effect of this instruction is to convert a signed word, doubleword, or quadword in the rAX register into a signed doubleword, quadword, or double-quadword in the rDX:rAX registers. This action helps avoid overflow problems in signed number arithmetic.

The CQO mnemonic is meaningful only in 64-bit mode.

Mnemonic	Opcode	Description
CWD	99	Sign-extend AX into DX:AX.
CDQ	99	Sign-extend EAX into EDX:EAX.
CQO	99	Sign-extend RAX into RDX:RAX.

Related Instructions

CBW, CWDE, CDQE

rFLAGS Affected

None

Exceptions

None

CLC Clear Carry Flag

Clears the carry flag (CF) in the rFLAGS register to zero.

Mnemonic	Opcode	Description
CLC	F8	Clear the carry flag (CF) to zero.

Related Instructions

STC, CMC

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
																0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

None

CLD Clear Direction Flag

Clears the direction flag (DF) in the rFLAGS register to zero. If the DF flag is 0, each iteration of a string instruction increments the data pointer (index registers rSI or rDI). If the DF flag is 1, the string instruction decrements the pointer. Use the CLD instruction before a string instruction to make the data pointer increment.

Mnemonic	Opcode	Description
CLD	FC	Clear the direction flag (DF) to zero.

Related Instructions

CMPS*x*, INSx, LODSx, MOVSx, OUTSx, SCASx, STD, STOSx

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
									0							
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

None

CLFLUSH Cache Line Flush

Flushes the cache line specified by the *mem8* linear-address. The instruction checks all levels of the cache hierarchy—internal caches and external caches—and invalidates the cache line in every cache in which it is found. If a cache contains a dirty copy of the cache line (that is, the cache line is in the *modified* or *owned* MOESI state), the line is written back to memory before it is invalidated. The instruction sets the cache-line MOESI state to *invalid*.

The instruction also checks the physical address corresponding to the linear-address operand against the processor's write-combining buffers. If the write-combining buffer holds data intended for that physical address, the instruction writes the entire contents of the buffer to memory. This occurs even though the data is not cached in the cache hierarchy. In a multiprocessor system, the instruction checks the write-combining buffers only on the processor that executed the CLFLUSH instruction.

On processors that do not support the CLFLUSHOPT instruction, (CPUID Fn 0000_0007_EBX_x0[CLFLOPT]=0), the CLFLUSH instruction is weakly ordered with respect to other instructions that operate on memory. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around a CLFLUSH instruction. Such reordering can invalidate a speculatively prefetched cache line, unintentionally defeating the prefetch operation. The only way to avoid this situation is to use the MFENCE instruction after the CLFLUSH instruction to force strong-ordering of the CLFLUSH instruction with respect to subsequent memory operations. The CLFLUSH instruction may also take effect on a cache line while stores from previous store instructions are still pending in the store buffer. To ensure that such stores are included in the cache line that is flushed, use an MFENCE instruction ahead of the CLFLUSH instruction. Such stores would otherwise cause the line to be re-cached and modified after the CLFLUSH completed. The LFENCE, SFENCE, and serializing instructions are *not* ordered with respect to CLFLUSH.

On processors that support CLFLUSHOPT, (CPUID Fn 0000_0007_EBX_x0[CLFLOPT]=1), CLFLUSH is ordered with respect to locked operations, fence instructions, and CLFLUSHOPT, CLFLUSH and write instructions that touch the same cache line. CLFLUSH is not ordered with CLFLUSHOPT, CLFLUSH and write instructions to other cache lines.

The CLFLUSH instruction behaves like a load instruction with respect to setting the page-table accessed and dirty bits. That is, it sets the page-table accessed bit to 1, but does not set the page-table dirty bit.

The CLFLUSH instruction executes at any privilege level. CLFLUSH performs all the segmentation and paging checks that a 1-byte read would perform, except that it also allows references to execute-only segments.

The CLFLUSH instruction is supported if the feature flag CPUID Fn0000_0001_EDX[CLFSH] is set. The 8-bit field CPUID Fn 0000_0001_EBX[CLFlush] returns the size of the cacheline in quadwords.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, "Instruction Subsets and CPUID Feature Flags," on page 587.

Mnemonic	Opcode	Description
CLFLUSH *mem8*	0F AE /7	flush cache line containing *mem8.*

Related Instructions

INVD, WBINVD, CLFLUSHOPT, CLZERO

rFLAGS Affected

None

Exceptions

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	CLFLUSH instruction is not supported, as indicated by CPUID Fn0000_0001_EDX[CLFSH] = 0.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.

CLFLUSHOPT Optimized Cache Line Flush

Flushes the cache line specified by the mem8 linear-address. The instruction checks all levels of the cache hierarchy-internal caches and external caches-and invalidates the cache line in every cache in which it is found. If a cache contains a dirty copy of the cache line (that is, the cache line is in the modified or owned MOESI state), the line is written back to memory before it is invalidated. The instruction sets the cache-line MOESI state to invalid.

The instruction also checks the physical address corresponding to the linear-address operand against the processor's write-combining buffers. If the write-combining buffer holds data intended for that physical address, the instruction writes the entire contents of the buffer to memory. This occurs even though the data is not cached in the cache hierarchy. In a multiprocessor system, the instruction checks the write-combining buffers only on the processor that executed the CLFLUSHOPT instruction.

The CLFLUSHOPT instruction is ordered with respect to fence instructions and locked operations. CLFLUSHOPT is also ordered with writes, CLFLUSH, and CLFLUSHOPT instructions that reference the same cache line as the CLFLUSHOPT. CLFLUSHOPT is not ordered with writes, CLFLUSH, and CLFLUSHOPT to other cache lines. To enforce ordering in that situation, a SFENCE instruction or stronger should be used.

Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around a CLFLUSHOPT instruction. Such reordering can invalidate a speculatively prefetched cache line, unintentionally defeating the prefetch operation.

The only way to avoid this situation is to use the MFENCE instruction after the CLFLUSHOPT instruction to force strong ordering of the CLFLUSHOPT instruction with respect to subsequent memory operations.

The CLFLUSHOPT instruction behaves like a load instruction with respect to setting the page-table accessed and dirty bits. That is, it sets the page-table accessed bit to 1, but does not set the page-table dirty bit.

The CLFLUSHOPT instruction executes at any privilege level. CLFLUSHOPT performs all the segmentation and paging checks that a 1-byte read would perform, except that it also allows references to execute-only segments.

The CLFLUSHOPT instruction is supported if the feature flag CPUID Fn0000_0007_EBX_x0[CLFLOPT]is set. The 8-bit field CPUID Fn 0000_0001_EBX[CLFlush] returns the size of the cacheline in quadwords.

Mnemonic	Opcode	Description
CLFLUSHOPT mem8	66 0F AE /7	Flush cache line containing mem8

Related Instructions

CLFLUSH

rFLAGS Affected

None

Exceptions

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	CLFLUSH instruction is not supported, as indicated by CPUID Fn0000_0001_EDX[CLFSH] = 0.
	X	X	X	Instruction not supported by CPUID Fn0000_0007_EBX_x0[CLFLUSHOPT] = 0
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.

CLWB Cache Line Write Back and Retain

Flushes the cache line specified by the *mem8* linear address. The instruction checks all levels of the cache hierarchy—internal caches and external caches—and causes the cache line, if dirty, to be written to memory. The cache line may be retained in the cache where found in a non-dirty state.

The CLWB instruction is weakly ordered with respect to other instructions that operate on memory. Speculative loads initiated by the processor, or specified explicitly using cache prefetch instructions, can be reordered around a CLWB instruction. CLWB is ordered naturally with older stores to the same address on the same logical processor. To create strict ordering of CLWB use a store-ordering instruction such as SFENCE.

The CLWB instruction behaves like a load instruction with respect to setting the page table accessed and dirty bits. That is, it sets the page table accessed bit to 1, but does not set the page table dirty bit.

The CLWB instruction executes at any privilege level. CLWB performs all the segmentation and paging checks that a 1-byte read would perform, except that it also allows references to execute only segments.

The CLWB instruction is supported if the feature flag CPUID Fn0000_0007-EBX[24]=1.

The 8-bit field CPUID Fn 0000_0001_EBX[CLFlush] returns the size of the cacheline in quadwords.

Mnemonic	Opcode	Description
CLWB	66 0F AE /6	Cache line write-back.

Related Instructions

CLFLUSH, CLFLUSHOPT, WBINVD, WBNOINVD

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID Fn0000_0007_EBX[24] = 0
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.

CLZERO Zero Cache Line

Clears the cache line specified by the logical address in rAX by writing a zero to every byte in the line. The instruction uses an implied non temporal memory type, similar to a streaming store, and uses the write combining protocol to minimize cache pollution.

CLZERO is weakly-ordered with respect to other instructions that operate on memory. Software should use an SFENCE or stronger to enforce memory ordering of CLZERO with respect to other store instructions.

The CLZERO instruction executes at any privilege level. CLZERO performs all the segmentation and paging checks that a store of the specified cache line would perform.

The CLZERO instruction is supported if the feature flag CPUID Fn8000_0008_EBX[CLZERO] is set. The 8-bit field CPUID Fn 0000_0001_EBX[CLFlush] returns the size of the cacheline in quadwords.

Mnemonic	Opcode	Description
CLZERO rAX	0F 01 FC	Clears cache line containing rAX

Related Instructions

CLFLUSH

rFLAGS Affected

None

Exceptions

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID Fn8000_0008_EBX[CLZERO] = 0
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.

CMC Complement Carry Flag

Complements (toggles) the carry flag (CF) bit of the rFLAGS register.

Mnemonic	Opcode	Description
CMC	F5	Complement the carry flag (CF).

Related Instructions

CLC, STC

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
																M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

None

CMOV*cc* Conditional Move

Conditionally moves a 16-bit, 32-bit, or 64-bit value in memory or a general-purpose register (second operand) into a register (first operand), depending upon the settings of condition flags in the rFLAGS register. If the condition is not satisfied, the destination register is not modified. For the memory-based forms of CMOVcc, memory-related exceptions may be reported even if the condition is false. In 64-bit mode, CMOVcc with a 32-bit operand size will clear the upper 32 bits of the destination register even if the condition is false.

The mnemonics of CMOVcc instructions denote the condition that must be satisfied. Most assemblers provide instruction mnemonics with A (above) and B (below) tags to supply the semantics for manipulating unsigned integers. Those with G (greater than) and L (less than) tags deal with signed integers. Many opcodes may be represented by synonymous mnemonics. For example, the CMOVL instruction is synonymous with the CMOVNGE instruction and denote the instruction with the opcode 0F 4C.

The feature flag CPUID Fn0000_0001_EDX[CMOV] or CPUID Fn8000_0001_EDX[CMOV] =1 indicates support for CMOV*cc* instructions on a particular processor implementation.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Opcode	Description
CMOVO *reg16*, *reg/mem16* CMOVO *reg32*, *reg/mem32* CMOVO *reg64*, *reg/mem64*	0F 40 */r*	Move if overflow (OF = 1).
CMOVNO *reg16*, *reg/mem16* CMOVNO *reg32*, *reg/mem32* CMOVNO *reg64*, *reg/mem64*	0F 41 */r*	Move if not overflow (OF = 0).
CMOVB *reg16*, *reg/mem16* CMOVB *reg32*, *reg/mem32* CMOVB *reg64*, *reg/mem64*	0F 42 */r*	Move if below (CF = 1).
CMOVC *reg16*, *reg/mem16* CMOVC *reg32*, *reg/mem32* CMOVC *reg64*, *reg/mem64*	0F 42 */r*	Move if carry (CF = 1).
CMOVNAE *reg16*, *reg/mem16* CMOVNAE *reg32*, *reg/mem32* CMOVNAE *reg64*, *reg/mem64*	0F 42 */r*	Move if not above or equal (CF = 1).
CMOVNB *reg16,reg/mem16* CMOVNB *reg32,reg/mem32* CMOVNB *reg64,reg/mem64*	0F 43 */r*	Move if not below (CF = 0).
CMOVNC *reg16,reg/mem16* CMOVNC *reg32,reg/mem32* CMOVNC *reg64,reg/mem64*	0F 43 */r*	Move if not carry (CF = 0).

Mnemonic	Opcode	Description
CMOVAE *reg16*, *reg/mem16* CMOVAE *reg32*, *reg/mem32* CMOVAE *reg64*, *reg/mem64*	0F 43 */r*	Move if above or equal (CF = 0).
CMOVZ *reg16*, *reg/mem16* CMOVZ *reg32*, *reg/mem32* CMOVZ *reg64*, *reg/mem64*	0F 44 */r*	Move if zero (ZF = 1).
CMOVE *reg16*, *reg/mem16* CMOVE *reg32*, *reg/mem32* CMOVE *reg64*, *reg/mem64*	0F 44 */r*	Move if equal (ZF =1).
CMOVNZ *reg16*, *reg/mem16* CMOVNZ *reg32*, *reg/mem32* CMOVNZ *reg64*, *reg/mem64*	0F 45 */r*	Move if not zero (ZF = 0).
CMOVNE *reg16*, *reg/mem16* CMOVNE *reg32*, *reg/mem32* CMOVNE *reg64*, *reg/mem64*	0F 45 */r*	Move if not equal (ZF = 0).
CMOVBE *reg16*, *reg/mem16* CMOVBE *reg32*, *reg/mem32* CMOVBE *reg64*, *reg/mem64*	0F 46 */r*	Move if below or equal (CF = 1 or ZF = 1).
CMOVNA *reg16*, *reg/mem16* CMOVNA *reg32*, *reg/mem32* CMOVNA *reg64*, *reg/mem64*	0F 46 */r*	Move if not above (CF = 1 or ZF = 1).
CMOVNBE *reg16, reg/mem16* CMOVNBE *reg32,reg/mem32* CMOVNBE *reg64,reg/mem64*	0F 47 */r*	Move if not below or equal (CF = 0 and ZF = 0).
CMOVA *reg16*, *reg/mem16* CMOVA *reg32*, *reg/mem32* CMOVA *reg64*, *reg/mem64*	0F 47 */r*	Move if above (CF = 0 and ZF = 0).
CMOVS *reg16*, *reg/mem16* CMOVS *reg32*, *reg/mem32* CMOVS *reg64*, *reg/mem64*	0F 48 */r*	Move if sign (SF =1).
CMOVNS *reg16*, *reg/mem16* CMOVNS *reg32*, *reg/mem32* CMOVNS *reg64*, *reg/mem64*	0F 49 */r*	Move if not sign (SF = 0).
CMOVP *reg16*, *reg/mem16* CMOVP *reg32*, *reg/mem32* CMOVP *reg64*, *reg/mem64*	0F 4A */r*	Move if parity (PF = 1).
CMOVPE *reg16*, *reg/mem16* CMOVPE *reg32*, *reg/mem32* CMOVPE *reg64*, *reg/mem64*	0F 4A */r*	Move if parity even (PF = 1).
CMOVNP *reg16*, *reg/mem16* CMOVNP *reg32*, *reg/mem32* CMOVNP *reg64*, *reg/mem64*	0F 4B */r*	Move if not parity (PF = 0).
CMOVPO *reg16*, *reg/mem16* CMOVPO *reg32*, *reg/mem32* CMOVPO *reg64*, *reg/mem64*	0F 4B */r*	Move if parity odd (PF = 0).

Mnemonic	Opcode	Description
CMOVL *reg16*, *reg/mem16* CMOVL *reg32*, *reg/mem32* CMOVL *reg64*, *reg/mem64*	0F 4C */r*	Move if less (SF <> OF).
CMOVNGE *reg16*, *reg/mem16* CMOVNGE *reg32*, *reg/mem32* CMOVNGE *reg64*, *reg/mem64*	0F 4C */r*	Move if not greater or equal (SF <> OF).
CMOVNL *reg16*, *reg/mem16* CMOVNL *reg32*, *reg/mem32* CMOVNL *reg64*, *reg/mem64*	0F 4D */r*	Move if not less (SF = OF).
CMOVGE *reg16*, *reg/mem16* CMOVGE *reg32*, *reg/mem32* CMOVGE *reg64*, *reg/mem64*	0F 4D */r*	Move if greater or equal (SF = OF).
CMOVLE *reg16*, *reg/mem16* CMOVLE *reg32*, *reg/mem32* CMOVLE *reg64*, *reg/mem64*	0F 4E */r*	Move if less or equal (ZF = 1 or SF <> OF).
CMOVNG *reg16*, *reg/mem16* CMOVNG *reg32*, *reg/mem32* CMOVNG *reg64*, *reg/mem64*	0F 4E */r*	Move if not greater (ZF = 1 or SF <> OF).
CMOVNLE *reg16*, *reg/mem16* CMOVNLE *reg32*, *reg/mem32* CMOVNLE *reg64*, *reg/mem64*	0F 4F */r*	Move if not less or equal (ZF = 0 and SF = OF).
CMOVG *reg16*, *reg/mem16* CMOVG *reg32*, *reg/mem32* CMOVG *reg64*, *reg/mem64*	0F 4F */r*	Move if greater (ZF = 0 and SF = OF).

Related Instructions

MOV

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	CMOVcc instruction is not supported, as indicated by CPUID Fn0000_0001_EDX[CMOV] or Fn8000_0001_EDX[CMOV] = 0.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

CMP Compare

Compares the contents of a register or memory location (first operand) with an immediate value or the contents of a register or memory location (second operand), and sets or clears the status flags in the rFLAGS register to reflect the results. To perform the comparison, the instruction subtracts the second operand from the first operand and sets the status flags in the same manner as the SUB instruction, but does not alter the first operand. If the second operand is an immediate value, the instruction sign-extends the value to the length of the first operand.

Use the CMP instruction to set the condition codes for a subsequent conditional jump (J*cc*), conditional move (CMOV*cc*), or conditional SET*cc* instruction. Appendix F, “Instruction Effects on RFLAGS” shows how instructions affect the rFLAGS status flags.

Mnemonic	Opcode	Description
CMP AL, *imm8*	3C *ib*	Compare an 8-bit immediate value with the contents of the AL register.
CMP AX, *imm16*	3D *iw*	Compare a 16-bit immediate value with the contents of the AX register.
CMP EAX, *imm32*	3D *id*	Compare a 32-bit immediate value with the contents of the EAX register.
CMP RAX, *imm32*	3D *id*	Compare a 32-bit immediate value with the contents of the RAX register.
CMP *reg/mem8*, *imm8*	80 /7 *ib*	Compare an 8-bit immediate value with the contents of an 8-bit register or memory operand.
CMP *reg/mem16*, *imm16*	81 /7 *iw*	Compare a 16-bit immediate value with the contents of a 16-bit register or memory operand.
CMP *reg/mem32*, *imm32*	81 /7 *id*	Compare a 32-bit immediate value with the contents of a 32-bit register or memory operand.
CMP *reg/mem64*, *imm32*	81 /7 *id*	Compare a 32-bit signed immediate value with the contents of a 64-bit register or memory operand.
CMP *reg/mem16*, *imm8*	83 /7 *ib*	Compare an 8-bit signed immediate value with the contents of a 16-bit register or memory operand.
CMP *reg/mem32*, *imm8*	83 /7 *ib*	Compare an 8-bit signed immediate value with the contents of a 32-bit register or memory operand.
CMP *reg/mem64*, *imm8*	83 /7 *ib*	Compare an 8-bit signed immediate value with the contents of a 64-bit register or memory operand.
CMP *reg/mem8*, *reg8*	38 */r*	Compare the contents of an 8-bit register or memory operand with the contents of an 8-bit register.
CMP *reg/mem16*, *reg16*	39 */r*	Compare the contents of a 16-bit register or memory operand with the contents of a 16-bit register.
CMP *reg/mem32*, *reg32*	39 */r*	Compare the contents of a 32-bit register or memory operand with the contents of a 32-bit register.
CMP *reg/mem64*, *reg64*	39 */r*	Compare the contents of a 64-bit register or memory operand with the contents of a 64-bit register.

Mnemonic	Opcode	Description
CMP *reg8*, *reg/mem8*	3A */r*	Compare the contents of an 8-bit register with the contents of an 8-bit register or memory operand.
CMP *reg16*, *reg/mem16*	3B */r*	Compare the contents of a 16-bit register with the contents of a 16-bit register or memory operand.
CMP *reg32*, *reg/mem32*	3B */r*	Compare the contents of a 32-bit register with the contents of a 32-bit register or memory operand.
CMP *reg64*, *reg/mem64*	3B */r*	Compare the contents of a 64-bit register with the contents of a 64-bit register or memory operand.

When interpreting operands as unsigned, flag settings are as follows:

Operands	CF	ZF
dest > source	0	0
dest = source	0	1
dest < source	1	0

When interpreting operands as signed, flag settings are as follows:

Operands	OF	ZF
dest > source	SF	0
dest = source	0	1
dest < source	NOT SF	0

Related Instructions

SUB, CMPS*x*, SCAS*x*

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

CMPS
CMPSB
CMPSW
CMPSD
CMPSQ

Compare Strings

Compares the bytes, words, doublewords, or quadwords pointed to by the rSI and rDI registers, sets or clears the status flags of the rFLAGS register to reflect the results, and then increments or decrements the rSI and rDI registers according to the state of the DF flag in the rFLAGS register. To perform the comparison, the instruction subtracts the second operand from the first operand and sets the status flags in the same manner as the SUB instruction, but does not alter the first operand. The two operands must be the same size.

If the DF flag is 0, the instruction increments rSI and rDI; otherwise, it decrements the pointers. It increments or decrements the pointers by 1, 2, 4, or 8, depending on the size of the operands.

The forms of the CMPSx instruction with explicit operands address the first operand at *seg*:[rSI]. The value of *seg* defaults to the DS segment, but may be overridden by a segment prefix. These instructions always address the second operand at ES:[rDI]. ES may not be overridden. The explicit operands serve only to specify the type (size) of the values being compared and the segment used by the first operand.

The no-operands forms of the instruction use the DS:[rSI] and ES:[rDI] registers to point to the values to be compared. The mnemonic determines the size of the operands.

Do not confuse this CMPSD instruction with the same-mnemonic CMPSD (compare scalar double-precision floating-point) instruction in the 128-bit media instruction set. Assemblers can distinguish the instructions by the number and type of operands.

For block comparisons, the CMPS instruction supports the REPE or REPZ prefixes (they are synonyms) and the REPNE or REPNZ prefixes (they are synonyms). For details about the REP prefixes, see “Repeat Prefixes” on page 12. If a conditional jump instruction like JL follows a CMPS*x* instruction, the jump occurs if the value of the *seg*:[rSI] operand is less than the ES:[rDI] operand. This action allows lexicographical comparisons of string or array elements. A CMPS*x* instruction can also operate inside a loop controlled by the LOOP*cc* instruction.

Mnemonic	Opcode	Description
CMPS *mem8*, *mem8*	A6	Compare the byte at DS:rSI with the byte at ES:rDI and then increment or decrement rSI and rDI.
CMPS *mem16*, *mem16*	A7	Compare the word at DS:rSI with the word at ES:rDI and then increment or decrement rSI and rDI.
CMPS *mem32*, *mem32*	A7	Compare the doubleword at DS:rSI with the doubleword at ES:rDI and then increment or decrement rSI and rDI.
CMPS *mem64*, *mem64*	A7	Compare the quadword at DS:rSI with the quadword at ES:rDI and then increment or decrement rSI and rDI.

Mnemonic	Opcode	Description
CMPSB	A6	Compare the byte at DS:rSI with the byte at ES:rDI and then increment or decrement rSI and rDI.
CMPSW	A7	Compare the word at DS:rSI with the word at ES:rDI and then increment or decrement rSI and rDI.
CMPSD	A7	Compare the doubleword at DS:rSI with the doubleword at ES:rDI and then increment or decrement rSI and rDI.
CMPSQ	A7	Compare the quadword at DS:rSI with the quadword at ES:rDI and then increment or decrement rSI and rDI.

Related Instructions

CMP, SCAS*x*

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

CMPXCHG Compare and Exchange

Compares the value in the AL, AX, EAX, or RAX register with the value in a register or a memory location (first operand). If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the rFLAGS register to 1. Otherwise, it copies the value in the first operand to the AL, AX, EAX, or RAX register and clears the ZF flag to 0.

The OF, SF, AF, PF, and CF flags are set to reflect the results of the compare.

When the first operand is a memory operand, CMPXCHG always does a read-modify-write on the memory operand. If the compared operands were unequal, CMPXCHG writes the same value to the memory operand that was read.

The forms of the CMPXCHG instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
CMPXCHG *reg/mem8*, *reg8*	0F B0 */r*	Compare AL register with an 8-bit register or memory location. If equal, copy the second operand to the first operand. Otherwise, copy the first operand to AL.
CMPXCHG *reg/mem16*, *reg16*	0F B1 */r*	Compare AX register with a 16-bit register or memory location. If equal, copy the second operand to the first operand. Otherwise, copy the first operand to AX.
CMPXCHG *reg/mem32*, *reg32*	0F B1 */r*	Compare EAX register with a 32-bit register or memory location. If equal, copy the second operand to the first operand. Otherwise, copy the first operand to EAX.
CMPXCHG *reg/mem64*, *reg64*	0F B1 */r*	Compare RAX register with a 64-bit register or memory location. If equal, copy the second operand to the first operand. Otherwise, copy the first operand to RAX.

Related Instructions

CMPXCHG8B, CMPXCHG16B

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

CMPXCHG8B Compare and Exchange Eight Bytes
CMPXCHG16B Compare and Exchange Sixteen Bytes

Compares the value in the rDX:rAX registers with a 64-bit or 128-bit value in the specified memory location. If the values are equal, the instruction copies the value in the rCX:rBX registers to the memory location and sets the zero flag (ZF) of the rFLAGS register to 1. Otherwise, it copies the value in memory to the rDX:rAX registers and clears ZF to 0.

If the effective operand size is 16-bit or 32-bit, the CMPXCHG8B instruction is used. This instruction uses the EDX:EAX and ECX:EBX register operands and a 64-bit memory operand. If the effective operand size is 64-bit, the CMPXCHG16B instruction is used; this instruction uses RDX:RAX register operands and a 128-bit memory operand.

The CMPXCHG8B and CMPXCHG16B instructions always do a read-modify-write on the memory operand. If the compared operands were unequal, the instructions write the same value to the memory operand that was read.

The CMPXCHG8B and CMPXCHG16B instructions support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Support for the CMPXCHG8B and CMPXCHG16B instructions is implementation dependent. Support for the CMPXCHG8B instruction is indicated by CPUID Fn0000_0001_EDX[CMPXCHG8B] or Fn8000_0001_EDX[CMPXCHG8B] = 1. Support for the CMPXCHG16B instruction is indicated by CPUID Fn0000_0001_ECX[CMPXCHG16B] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

The memory operand used by CMPXCHG16B must be 16-byte aligned or else a general-protection exception is generated.

Mnemonic	Opcode	Description
CMPXCHG8B *mem64*	0F C7 /1 *m64*	Compare EDX:EAX register to 64-bit memory location. If equal, set the zero flag (ZF) to 1 and copy the ECX:EBX register to the memory location. Otherwise, copy the memory location to EDX:EAX and clear the zero flag.
CMPXCHG16B *mem128*	0F C7 /1 *m128*	Compare RDX:RAX register to 128-bit memory location. If equal, set the zero flag (ZF) to 1 and copy the RCX:RBX register to the memory location. Otherwise, copy the memory location to RDX:RAX and clear the zero flag.

Related Instructions

CMPXCHG

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
													M			
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	CMPXCHG8B instruction is not supported, as indicated by CPUID Fn0000_0001_EDX[CMPXCHG8B] or Fn8000_0001_EDX[CMPXCHG8B] = 0.
			X	CMPXCHG16B instruction is not supported, as indicated by CPUID Fn0000_0001_ECX[CMPXCHG16B] = 0.
	X	X	X	The operand was a register.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
			X	The memory operand for CMPXCHG16B was not aligned on a 16-byte boundary.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

CPUID Processor Identification

Provides information about the processor and its capabilities through a number of different functions. Software should load the number of the CPUID function to execute into the EAX register before executing the CPUID instruction. The processor returns information in the EAX, EBX, ECX, and EDX registers; the contents and format of these registers depend on the function.

The architecture supports CPUID information about *standard functions* and *extended functions*. The standard functions have numbers in the 0000_*xxxx*h series (for example, standard function 1). To determine the largest standard function number that a processor supports, execute CPUID function 0.

The extended functions have numbers in the 8000_*xxxx*h series (for example, extended function 8000_0001h). To determine the largest extended function number that a processor supports, execute CPUID extended function 8000_0000h. If the value returned in EAX is greater than 8000_0000h, the processor supports extended functions.

Software operating at any privilege level can execute the CPUID instruction to collect this information. In 64-bit mode, this instruction works the same as in legacy mode except that it zero-extends 32-bit register results to 64 bits.

CPUID is a serializing instruction.

Mnemonic	Opcode	Description
CPUID	0F A2	Returns information about the processor and its capabilities. EAX specifies the function number, and the data is returned in EAX, EBX, ECX, EDX.

Testing for the CPUID Instruction

To avoid an invalid-opcode exception (#UD) on those processor implementations that do not support the CPUID instruction, software must first test to determine if the CPUID instruction is supported. Support for the CPUID instruction is indicated by the ability to write the ID bit in the rFLAGS register. Normally, 32-bit software uses the PUSHFD and POPFD instructions in an attempt to write rFLAGS.ID. After reading the updated rFLAGS.ID bit, a comparison determines if the operation changed its value. If the value changed, the processor executing the code supports the CPUID instruction. If the value did not change, rFLAGS.ID is not writable, and the processor does not support the CPUID instruction.

The following code sample shows how to test for the presence of the CPUID instruction using 32-bit code.

```
pushfd                      ; save EFLAGS
pop     eax                 ; store EFLAGS in EAX
mov     ebx, eax            ; save in EBX for later testing
xor     eax, 00200000h      ; toggle bit 21
push    eax                 ; push to stack
popfd                       ; save changed EAX to EFLAGS
```

```
pushfd                  ; push EFLAGS to TOS
pop     eax             ; store EFLAGS in EAX
cmp     eax, ebx        ; see if bit 21 has changed
jz      NO_CPUID        ; if no change, no CPUID
```

Standard Function 0 and Extended Function 8000_0000h

CPUID standard function 0 loads the EAX register with the largest CPUID *standard* function number supported by the processor implementation; similarly, CPUID extended function 8000_0000h loads the EAX register with the largest *extended* function number supported.

Standard function 0 and extended function 8000_0000h both load a 12-character string into the EBX, EDX, and ECX registers identifying the processor vendor. For AMD processors, the string is `AuthenticAMD`. This string informs software that it should follow the AMD CPUID definition for subsequent CPUID function calls. If the function returns another vendor's string, software must use that vendor's CPUID definition when interpreting the results of subsequent CPUID function calls. Table 3-2 shows the contents of the EBX, EDX, and ECX registers after executing function 0 on an AMD processor.

Table 3-2. Processor Vendor Return Values

Register	Return Value	ASCII Characters
EBX	6874_7541h	"h t u A"
EDX	6974_6E65h	"i t n e"
ECX	444D_4163h	"D M A c"

For a description of all feature flags related to instruction subset support, see Appendix D, "Instruction Subsets and CPUID Feature Flags," on page 587. For a description of all defined feature numbers and return values, see Appendix E, "Obtaining Processor Information Via the CPUID Instruction," on page 593.

Related Instructions

None

rFLAGS Affected

None

Exceptions

None

CRC32 CRC32 Cyclical Redundancy Check

Performs one step of a 32-bit cyclic redundancy check.

The first source, which is also the destination, is a doubleword value in either a 32-bit or 64-bit GPR depending on the presence of a REX prefix and the value of the REX.W bit. The second source is a GPR or memory location of width 8, 16, or 32 bits. A vector of width 40, 48, or 64 bits is derived from the two operands as follows:

1. The low-order 32 bits of the first operand is bit-wise inverted and shifted left by the width of the second operand.
2. The second operand is bit-wise inverted and shifted left by 32 bits
3. The results of steps 1 and 2 are `xor`ed.

This vector is interpreted as a polynomial of degree 40, 48, or 64 over the field of two elements (i.e., bit *i* is interpreted as the coefficient of X^i). This polynomial is divided by the polynomial of degree 32 that is similarly represented by the vector 11EDC6F41h. (The division admits an efficient iterative implementation based on the `xor` operation.) The remainder is encoded as a 32-bit vector, which is bit-wise inverted and written to the destination. In the case of a 64-bit destination, the upper 32 bits are cleared.

In an application of the CRC algorithm, a data block is partitioned into byte, word, or doubleword segments and CRC32 is executed iteratively, once for each segment.

CRC32 is a SSE4.2 instruction. Support for SSE4.2 instructions is indicated by CPUID Fn0000_0001_ECX[SSE42] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Instruction Encoding

Mnemonic	Encoding	Notes
CRC32 *reg32*, *reg/mem8*	F2 0F 38 F0 /r	Perform CRC32 operation on 8-bit values
CRC32 *reg32*, *reg/mem8*	F2 REX 0F 38 F0 /r	Encoding using REX prefix allows access to GPR8–15
CRC32 *reg32*, *reg/mem16*	F2 0F 38 F1 /r	Effective operand size determines size of second operand.
CRC32 *reg32*, *reg/mem32*	F2 0F 38 F1 /r	
CRC32 *reg64*, *reg/mem8*	F2 REX.W 0F 38 F0 /r	REX.W = 1.
CRC32 *reg64*, *reg/mem64*	F2 REX.W 0F 38 F1 /r	REX.W = 1.

rFLAGS Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X	X	Lock prefix used
	X	X	X	SSE42 instructions are not supported as indicated by CPUID Fn0000_0001_ECX[SSE42] = 0.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

DAA Decimal Adjust after Addition

Adjusts the value in the AL register into a packed BCD result and sets the CF and AF flags in the rFLAGS register to indicate a decimal carry out of either nibble of AL.

Use this instruction to adjust the result of a byte ADD instruction that performed the binary addition of one 2-digit packed BCD values to another.

The instruction performs the adjustment by adding 06h to AL if the lower nibble is greater than 9 or if AF = 1. Then 60h is added to AL if the original AL was greater than 99h or if CF = 1.

If the lower nibble of AL was adjusted, the AF flag is set to 1. Otherwise AF is not modified. If the upper nibble of AL was adjusted, the CF flag is set to 1. Otherwise, CF is not modified. SF, ZF, and PF are set according to the final value of AL.

Using this instruction in 64-bit mode generates an invalid-opcode (#UD) exception.

Mnemonic	Opcode	Description
DAA	27	Decimal adjust AL. (Invalid in 64-bit mode.)

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	This instruction was executed in 64-bit mode.

DAS Decimal Adjust after Subtraction

Adjusts the value in the AL register into a packed BCD result and sets the CF and AF flags in the rFLAGS register to indicate a decimal borrow.

Use this instruction to adjust the result of a byte SUB instruction that performed a binary subtraction of one 2-digit, packed BCD value from another.

This instruction performs the adjustment by subtracting 06h from AL if the lower nibble is greater than 9 or if AF = 1. Then 60h is subtracted from AL if the original AL was greater than 99h or if CF = 1.

If the adjustment changes the lower nibble of AL, the AF flag is set to 1; otherwise AF is not modified. If the adjustment results in a borrow for either nibble of AL, the CF flag is set to 1; otherwise CF is not modified. The SF, ZF, and PF flags are set according to the final value of AL.

Using this instruction in 64-bit mode generates an invalid-opcode (#UD) exception.

Mnemonic	Opcode	Description
DAS	2F	Decimal adjusts AL after subtraction. (Invalid in 64-bit mode.)

Related Instructions

DAA

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	This instruction was executed in 64-bit mode.

DEC Decrement by 1

Subtracts 1 from the specified register or memory location. The CF flag is not affected.

The one-byte forms of this instruction (opcodes 48 through 4F) are used as REX prefixes in 64-bit mode. See “REX Prefix” on page 14.

The forms of the DEC instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.

Mnemonic	Opcode	Description
DEC *reg/mem8*	FE /1	Decrement the contents of an 8-bit register or memory location by 1.
DEC *reg/mem16*	FF /1	Decrement the contents of a 16-bit register or memory location by 1.
DEC *reg/mem32*	FF /1	Decrement the contents of a 32-bit register or memory location by 1.
DEC *reg/mem64*	FF /1	Decrement the contents of a 64-bit register or memory location by 1.
DEC *reg16*	48 +*rw*	Decrement the contents of a 16-bit register by 1. (See “REX Prefix” on page 14.)
DEC *reg32*	48 +*rd*	Decrement the contents of a 32-bit register by 1. (See “REX Prefix” on page 14.)

Related Instructions

INC, SUB

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded the data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

DIV Unsigned Divide

Divides the unsigned value in a register by the unsigned value in the specified register or memory location. The register to be divided depends on the size of the divisor.

When dividing a word, the dividend is in the AX register. The instruction stores the quotient in the AL register and the remainder in the AH register.

When dividing a doubleword, quadword, or double quadword, the most-significant word of the dividend is in the rDX register and the least-significant word is in the rAX register. After the division, the instruction stores the quotient in the rAX register and the remainder in the rDX register.

The following table summarizes the action of this instruction:

Division Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	reg/mem8	AL	AH	255
Doubleword/word	DX:AX	reg/mem16	AX	DX	65,535
Quadword/doubleword	EDX:EAX	reg/mem32	EAX	EDX	$2^{32} - 1$
Double quadword/ quadword	RDX:RAX	reg/mem64	RAX	RDX	$2^{64} - 1$

The instruction truncates non-integral results towards 0 and the remainder is always less than the divisor. An overflow generates a #DE (divide error) exception, rather than setting the CF flag.

Division by zero generates a divide-by-zero exception.

Mnemonic	Opcode	Description
DIV *reg/mem8*	F6 /6	Perform unsigned division of AX by the contents of an 8-bit register or memory location and store the quotient in AL and the remainder in AH.
DIV *reg/mem16*	F7 /6	Perform unsigned division of DX:AX by the contents of a 16-bit register or memory operand store the quotient in AX and the remainder in DX.
DIV *reg/mem32*	F7 /6	Perform unsigned division of EDX:EAX by the contents of a 32-bit register or memory location and store the quotient in EAX and the remainder in EDX.
DIV *reg/mem64*	F7 /6	Perform unsigned division of RDX:RAX by the contents of a 64-bit register or memory location and store the quotient in RAX and the remainder in RDX.

Related Instructions

MUL

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	U	U	U	U
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Divide by zero, #DE	X	X	X	The divisor operand was 0.
	X	X	X	The quotient was too large for the designated register.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

ENTER Create Procedure Stack Frame

Creates a stack frame for a procedure.

The first operand specifies the size of the stack frame allocated by the instruction.

The second operand specifies the nesting level (0 to 31—the value is automatically masked to 5 bits). For nesting levels of 1 or greater, the processor copies earlier stack frame pointers before adjusting the stack pointer. This action provides a called procedure with access points to other nested stack frames.

The 32-bit `enter N, 0` (a nesting level of 0) instruction is equivalent to the following 32-bit instruction sequence:

```
push  ebp         ; save current EBP
mov   ebp, esp    ; set stack frame pointer value
sub   esp, N      ; allocate space for local variables
```

The ENTER and LEAVE instructions provide support for block structured languages. The LEAVE instruction releases the stack frame on returning from a procedure.

In 64-bit mode, the operand size of ENTER defaults to 64 bits, and there is no prefix available for encoding a 32-bit operand size.

Mnemonic	Opcode	Description
ENTER *imm16*, 0	C8 *iw* 00	Create a procedure stack frame.
ENTER *imm16*, 1	C8 *iw* 01	Create a nested stack frame for a procedure.
ENTER *imm16, imm8*	C8 *iw ib*	Create a nested stack frame for a procedure.

Action

```
// See “Pseudocode Definition” on page 57.

ENTER_START:

    temp_ALLOC_SPACE = word-sized immediate specified in the instruction
                       (first operand), zero-extended to 64 bits
    temp_LEVEL = byte-sized immediate specified in the instruction
                       (second operand), zero-extended to 64 bits

    temp_LEVEL = temp_LEVEL AND 0x1f
                                 // only keep 5 bits of level count

    PUSH.v old_RBP

    temp_RBP = RSP             // This value of RSP will eventually be loaded
                               // into RBP.
    IF (temp_LEVEL>0)          // Push "temp_LEVEL" parameters to the stack.
    {
        FOR (I=1; I<temp_LEVEL; I++)
```

```
                                    // All but one of the parameters are copied
                                    // from higher up on the stack.
            {
                temp_DATA = READ_MEM.v [SS:old_RBP-I*V]
                PUSH.v temp_DATA
            }
            PUSH.v temp_RBP         // The last parameter is the offset of the old
                                    // value of RSP on the stack.
        }
        RSP.s = RSP - temp_ALLOC_SPACE   // Leave "temp_ALLOC_SPACE" free bytes on
                                         // the stack

        WRITE_MEM.v [SS:RSP.s] = temp_unused   // ENTER finishes with a memory
write
                                              // check on the final stack pointer,
                                               // but no write actually occurs.

        RBP.v = temp_RBP
        EXIT
```

Related Instructions

LEAVE

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack-segment limit or was non-canonical.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

IDIV Signed Divide

Divides the signed value in a register by the signed value in the specified register or memory location. The register to be divided depends on the size of the divisor.

When dividing a word, the dividend is in the AX register. The instruction stores the quotient in the AL register and the remainder in the AH register.

When dividing a doubleword, quadword, or double quadword, the most-significant word of the dividend is in the rDX register and the least-significant word is in the rAX register. After the division, the instruction stores the quotient in the rAX register and the remainder in the rDX register.

The following table summarizes the action of this instruction:

Division Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	reg/mem8	AL	AH	–128 to +127
Doubleword/word	DX:AX	reg/mem16	AX	DX	–32,768 to +32,767
Quadword/doubleword	EDX:EAX	reg/mem32	EAX	EDX	-2^{31} to $2^{31}-1$
Double quadword/ quadword	RDX:RAX	reg/mem64	RAX	RDX	-2^{63} to $2^{63}-1$

The instruction truncates non-integral results towards 0. The sign of the remainder is always the same as the sign of the dividend, and the absolute value of the remainder is less than the absolute value of the divisor. An overflow generates a #DE (divide error) exception, rather than setting the OF flag.

To avoid overflow problems, precede this instruction with a CBW, CWD, CDQ, or CQO instruction to sign-extend the dividend.

Mnemonic	Opcode	Description
IDIV *reg/mem8*	F6 /7	Perform signed division of AX by the contents of an 8-bit register or memory location and store the quotient in AL and the remainder in AH.
IDIV *reg/mem16*	F7 /7	Perform signed division of DX:AX by the contents of a 16-bit register or memory location and store the quotient in AX and the remainder in DX.
IDIV *reg/mem32*	F7 /7	Perform signed division of EDX:EAX by the contents of a 32-bit register or memory location and store the quotient in EAX and the remainder in EDX.
IDIV *reg/mem64*	F7 /7	Perform signed division of RDX:RAX by the contents of a 64-bit register or memory location and store the quotient in RAX and the remainder in RDX.

Related Instructions

IMUL

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	U	U	U	U
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Divide by zero, #DE	X	X	X	The divisor operand was 0.
	X	X	X	The quotient was too large for the designated register.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

IMUL Signed Multiply

Multiplies two signed operands. The number of operands determines the form of the instruction.

If a single operand is specified, the instruction multiplies the value in the specified general-purpose register or memory location by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and stores the product in AX, DX:AX, EDX:EAX, or RDX:RAX, respectively.

If two operands are specified, the instruction multiplies the value in a general-purpose register (first operand) by an immediate value or the value in a general-purpose register or memory location (second operand) and stores the product in the first operand location.

If three operands are specified, the instruction multiplies the value in a general-purpose register or memory location (second operand), by an immediate value (third operand) and stores the product in a register (first operand).

The IMUL instruction sign-extends an immediate operand to the length of the other register/memory operand.

The CF and OF flags are set if, due to integer overflow, the double-width multiplication result cannot be represented in the half-width destination register. Otherwise the CF and OF flags are cleared.

Mnemonic	Opcode	Description
IMUL *reg/mem8*	F6 /5	Multiply the contents of AL by the contents of an 8-bit memory or register operand and put the signed result in AX.
IMUL *reg/mem16*	F7 /5	Multiply the contents of AX by the contents of a 16-bit memory or register operand and put the signed result in DX:AX.
IMUL *reg/mem32*	F7 /5	Multiply the contents of EAX by the contents of a 32-bit memory or register operand and put the signed result in EDX:EAX.
IMUL *reg/mem64*	F7 /5	Multiply the contents of RAX by the contents of a 64-bit memory or register operand and put the signed result in RDX:RAX.
IMUL *reg16*, *reg/mem16*	0F AF */r*	Multiply the contents of a 16-bit destination register by the contents of a 16-bit register or memory operand and put the signed result in the 16-bit destination register.
IMUL *reg32*, *reg/mem32*	0F AF */r*	Multiply the contents of a 32-bit destination register by the contents of a 32-bit register or memory operand and put the signed result in the 32-bit destination register.
IMUL *reg64*, *reg/mem64*	0F AF */r*	Multiply the contents of a 64-bit destination register by the contents of a 64-bit register or memory operand and put the signed result in the 64-bit destination register.
IMUL *reg16*, *reg/mem16*, *imm8*	6B */r ib*	Multiply the contents of a 16-bit register or memory operand by a sign-extended immediate byte and put the signed result in the 16-bit destination register.

Mnemonic	Opcode	Description
IMUL *reg32*, *reg/mem32*, *imm8*	6B */r ib*	Multiply the contents of a 32-bit register or memory operand by a sign-extended immediate byte and put the signed result in the 32-bit destination register.
IMUL *reg64*, *reg/mem64*, *imm8*	6B */r ib*	Multiply the contents of a 64-bit register or memory operand by a sign-extended immediate byte and put the signed result in the 64-bit destination register.
IMUL *reg16*, *reg/mem16*, *imm16*	69 */r iw*	Multiply the contents of a 16-bit register or memory operand by a sign-extended immediate word and put the signed result in the 16-bit destination register.
IMUL *reg32*, *reg/mem32*, *imm32*	69 */r id*	Multiply the contents of a 32-bit register or memory operand by a sign-extended immediate double and put the signed result in the 32-bit destination register.
IMUL *reg64*, *reg/mem64*, *imm32*	69 */r id*	Multiply the contents of a 64-bit register or memory operand by a sign-extended immediate double and put the signed result in the 64-bit destination register.

Related Instructions

IDIV

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				U	U	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

IN Input from Port

Transfers a byte, word, or doubleword from an I/O port to the AL, AX, or EAX register. The port address is specified either by an 8-bit immediate value (00h to FFh) encoded in the instruction or a 16-bit value contained in the DX register (0000h to FFFFh). The processor's I/O address space is distinct from system memory addressing.

For two opcodes (E4h and ECh), the data size of the port is fixed at 8 bits. For the other opcodes (E5h and EDh), the effective operand-size determines the port size. If the effective operand size is 64 bits, IN reads only 32 bits from the I/O port.

If the CPL is higher than IOPL, or the mode is virtual mode, IN checks the I/O permission bitmap in the TSS before allowing access to the I/O port. (See Volume 2 for details on the TSS I/O permission bitmap.)

Mnemonic	Opcode	Description
IN AL, *imm8*	E4 *ib*	Input a byte from the port at the address specified by *imm8* and put it into the AL register.
IN AX, *imm8*	E5 *ib*	Input a word from the port at the address specified by *imm8* and put it into the AX register.
IN EAX, *imm8*	E5 *ib*	Input a doubleword from the port at the address specified by *imm8* and put it into the EAX register.
IN AL, DX	EC	Input a byte from the port at the address specified by the DX register and put it into the AL register.
IN AX, DX	ED	Input a word from the port at the address specified by the DX register and put it into the AX register.
IN EAX, DX	ED	Input a doubleword from the port at the address specified by the DX register and put it into the EAX register.

Related Instructions

INS*x*, OUT, OUTS*x*

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X		One or more I/O permission bits were set in the TSS for the accessed port.
			X	The CPL was greater than the IOPL and one or more I/O permission bits were set in the TSS for the accessed port.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.

INC Increment by 1

Adds 1 to the specified register or memory location. The CF flag is not affected, even if the operand is incremented to 0000.

The one-byte forms of this instruction (opcodes 40 through 47) are used as REX prefixes in 64-bit mode. See “REX Prefix” on page 14.

The forms of the INC instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

To perform an increment operation that updates the CF flag, use an ADD instruction with an immediate operand of 1.

Mnemonic	Opcode	Description
INC *reg/mem8*	FE /0	Increment the contents of an 8-bit register or memory location by 1.
INC *reg/mem16*	FF /0	Increment the contents of a 16-bit register or memory location by 1.
INC *reg/mem32*	FF /0	Increment the contents of a 32-bit register or memory location by 1.
INC *reg/mem64*	FF /0	Increment the contents of a 64-bit register or memory location by 1.
INC *reg16*	40 +*rw*	Increment the contents of a 16-bit register by 1. (These opcodes are used as REX prefixes in 64-bit mode. See “REX Prefix” on page 14.)
INC *reg32*	40 +*rd*	Increment the contents of a 32-bit register by 1. (These opcodes are used as REX prefixes in 64-bit mode. See “REX Prefix” on page 14.)

Related Instructions

ADD, DEC

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

INS
INSB
INSW
INSD

Input String

Transfers data from the I/O port specified in the DX register to an input buffer specified in the rDI register and increments or decrements the rDI register according to the setting of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments rDI by 1, 2, or 4, depending on the number of bytes read. If the DF flag is 1, it decrements the pointer by 1, 2, or 4.

In 16-bit and 32-bit mode, the INS instruction always uses ES as the data segment. The ES segment cannot be overridden with a segment override prefix. In 64-bit mode, INS always uses the unsegmented memory space.

The INS instructions use the explicit memory operand (first operand) to determine the size of the I/O port, but always use ES:[rDI] for the location of the input buffer. The explicit register operand (second operand) specifies the I/O port address and must always be DX.

The INSB, INSW, and INSD instructions copy byte, word, and doubleword data, respectively, from the I/O port (0000h to FFFFh) specified in the DX register to the input buffer specified in the ES:rDI registers.

If the operand size is 64-bits, the instruction behaves as if the operand size were 32-bits.

If the CPL is higher than the IOPL or the mode is virtual mode, INS*x* checks the I/O permission bitmap in the TSS before allowing access to the I/O port. (See volume 2 for details on the TSS I/O permission bitmap.)

The INSx instructions support the REP prefix for block input of rCX bytes, words, or doublewords. For details about the REP prefix, see “Repeat Prefixes” on page 12.

Mnemonic	Opcode	Description
INS *mem8*, DX	6C	Input a byte from the port specified by DX, put it into the memory location specified in ES:rDI, and then increment or decrement rDI.
INS *mem16*, DX	6D	Input a word from the port specified by DX register, put it into the memory location specified in ES:rDI, and then increment or decrement rDI.
INS *mem32*, DX	6D	Input a doubleword from the port specified by DX, put it into the memory location specified in ES:rDI, and then increment or decrement rDI.
INSB	6C	Input a byte from the port specified by DX, put it into the memory location specified in ES:rDI, and then increment or decrement rDI.

Mnemonic	Opcode	Description
INSW	6D	Input a word from the port specified by DX, put it into the memory location specified in ES:rDI, and then increment or decrement rDI.
INSD	6D	Input a doubleword from the port specified by DX, put it into the memory location specified in ES:rDI, and then increment or decrement rDI.

Related Instructions

IN, OUT, OUTS*x*

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
		X		One or more I/O permission bits were set in the TSS for the accessed port.
			X	The CPL was greater than the IOPL and one or more I/O permission bits were set in the TSS for the accessed port.
			X	A null data segment was used to reference memory.
			X	The destination operand was in a non-writable segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

INT Interrupt to Vector

Transfers execution to the interrupt handler specified by an 8-bit unsigned immediate value. This value is an interrupt vector number (00h to FFh), which the processor uses as an index into the interrupt-descriptor table (IDT).

For detailed descriptions of the steps performed by INT*n* instructions, see the following:

- *Legacy-Mode Interrupts:* "Virtual-8086 Mode Interrupt Control Transfers" in Volume 2.
- *Long-Mode Interrupts:* "Long-Mode Interrupt Control Transfers" in Volume 2.

See also the descriptions of the INT3 instruction on page 381 and the INTO instruction on page 195.

Mnemonic	Opcode	Description
INT *imm8*	CD *ib*	Call interrupt service routine specified by interrupt vector *imm8*.

Action

```
// For functions READ_IDT, READ_DESCRIPTOR, READ_INNER_LEVEL_SP,
// ShadowStacksEnabled and SET_TOKEN_BUSY see "Pseudocode Definition"
// on page 57

INT_N_START:

IF (REAL_MODE)
   INT_N_REAL           // INTn real mode
ELSEIF (PROTECTED_MODE)
   INT_N_PROTECTED      // INTn protected mode
ELSE  // (VIRTUAL_MODE)
   INT_N_VIRTUAL        // INTn virtual mode

INT_N_REAL:

temp_int_n_vector = byte-sized interrupt vector specified in
                    the instruction, zero-extended to 64 bits

// read target CS:RIP from the real-mode IDT
temp_RIP = READ_MEM.w [idt:temp_int_n_vector*4]
temp_CS  = READ_MEM.w [idt:temp_int_n_vector*4+2]

PUSH.w old_RFLAGS
PUSH.w old_CS
PUSH.w next_RIP

IF (temp_RIP > CS.limit)
   EXCEPTION [#GP(0)]

CS.sel  = temp_CS
CS.base = temp_CS SHL 4
```

```
RFLAGS.AC,TF,IF,RF cleared
RIP = temp_RIP

EXIT


INT_N_PROTECTED:

temp_int_n_vector = byte-sized interrupt vector specified in
                    the instruction, zero-extended to 64 bits
temp_idt_desc = READ_IDT (temp_int_n_vector)

IF (temp_idt_desc.attr.type == 'taskgate')
   TASK_SWITCH // using TSS selector in the task gate as the target TSS

// The size of the gate controls the size of the stack pushes
IF (LONG_MODE)
   v = 8-byte    // Long mode only uses 64-bit gates
ELSEIF ((temp_idt_desc.attr.type == 'intgate32') ||
        (temp_idt_desc.attr.type == 'trapgate32'))
   v = 4-byte    // Legacy mode, using a 32-bit gate
ELSE
   v = 2-byte    // Legacy mode, using a 16-bit gate

temp_RIP = temp_idt_desc.offset

IF (LONG_MODE)  // In long mode, read 2nd half of 16-byte interrupt-gate
   {            //  from the IDT to get the upper 32 bits of  target RIP
   temp_upper = READ_MEM.q [idt:temp_int_n_vector*16+8]
   temp_RIP   = temp_RIP + (temp_upper SHL 32) // form 64-bit target RIP
   }

CS = READ_DESCRIPTOR (temp_idt_desc.segment, intcs_chk)

IF (CS.attr.conforming == 1)
   temp_CPL = CPL
ELSE
   temp_CPL = CS.attr.dpl

IF (CPL == temp_CPL) // no privilege-level change
   {
   temp_CheckToken = FALSE
   IF (LONG_MODE)
      {
      IF (temp_idt_desc.ist != 0)
         {
         // IDT gate IST is non-zero, do stack switch
         RSP = READ_MEM.q [tss:ist_index*8+28]  // fetch new RSP
         RSP = RSP AND 0xFFFFFFFFFFFFFFF0       // ensure 16-byte alignment

         // fetch SSP from ISST if sstk enabled at current privilege
```

```
            IF (ShadowStacksEnabled(current CPL))
               {
                temp_isst_addr = INTERRUPT_SSP_TABLE_ADDR + (temp_idt_desc.ist*8)
                SSP = READ_MEM.q [tss:temp_isst_addr]
                IF (SSP[2:0] != 0)
                      EXCEPTION [#GP(0)] // new SSP must be 8-byte aligned
                temp_CheckToken = TRUE
               }
            }
         PUSH.q old_SS         // in long mode, save old SS:RSP to stack
         PUSH.q old_RSP
         }  // end long mode

      PUSH.v old_RFLAGS
      PUSH.v old_CS
      PUSH.v next_RIP

      IF (ShadowStacksEnabled(current CPL))
         {
         IF (temp_CheckToken == TRUE)
             SET_SSTK_TOKEN_BUSY(SSP)        // vaidate token, set busy
         Align SSP to next 8B boundary, storing 4B of 0 if needed
         SSTK_WRITE_MEM.q [SSP-24] = old_CS  // push CS,LIP,SSP to shadow stack
         SSTK_WRITE_MEM.q [SSP-16] = (CS.base + old_RIP)
         SSTK_WRITE_MEM.q [SSP-8]  = old_SSP
         SSP = SSP - 24
         } // end shadow stacks enabled @ CPL

      IF ((64BIT_MODE) && (temp_RIP is non-canonical) ||
          (!64BIT_MODE) && (temp_RIP > CS.limit))
               EXCEPTION [#GP(0)]
      RFLAGS.VM,NT,TF,RF cleared
      RFLAGS.IF cleared if interrupt gate
      RIP = temp_RIP
      EXIT
      }     // end of INTn to same privilege level

   ELSE     // INTn to more privileged level
      {
      // (CPL > temp_CPL), changing privilege so get inner level SS:RSP
      CPL = temp_CPL
      temp_SS_desc:temp_RSP = READ_INNER_LEVEL_SP(CPL, temp_idt_desc.ist)

      IF (LONG_MODE)
         temp_RSP = temp_RSP AND 0xFFFFFFFFFFFFFFF0   // force 16-byte alignment
      RSP = temp_RSP
      SS = temp_SS_desc

      IF (ShadowStacksEnabled(new CPL))
         {
         old_SSP = SSP
```

```
      IF ((temp_idt_desc.ist == 0) || (!LONG_MODE))
         SSP = PLn_SSP   // where n=new CPL
      ELSE
         {
         temp_isst_addr = INTERRUPT_SSP_TABLE_ADDR + (temp_idt_desc.ist*8)
         SSP = READ_MEM.q [tss:temp_isst_addr]
         }
      IF (SSP[2:0] != 0)  // new SSP must be 8-byte aligned
            EXCEPTION [#GP(0)]
      }

   // Any #SS from the following pushes uses SS.sel as error code
   PUSH.v old_SS
   PUSH.v old_RSP
   PUSH.v old_RFLAGS
   PUSH.v old_CS
   PUSH.v next_RIP

   IF ((ShadowStacksEnabled(CPL 3) && (old_CPL == 3))
      PL3_SSP = SSP

   IF (ShadowStacksEnabled(new CPL))
      {
      old_SSP = SSP
      SSP     = PLn_SSP        // where n=new CPL
      SET_SSTK_TOKEN_BUSY(SSP) // validate token, set busy
      IF (old_CPL != 3)
         SSTK_WRITE_MEM.q [SSP-24] = old_CS   // push CS, LIP, SSP
         SSTK_WRITE_MEM.q [SSP-16] = LIP      // onto the shadow stack
         SSTK_WRITE_MEM.q [SSP-8]  = old_SSP
         SSP = SSP - 24
      }  // end shadow stacks enabled at new CPL

   IF ((64BIT_MODE)  && (temp_RIP is non-canonical) ||
       (!64BIT_MODE) && (temp_RIP > CS.limit))
            EXCEPTION [#GP(0)]

   RFLAGS.VM,NT,TF,RF cleared
   RFLAGS.IF cleared if interrupt gate
   RIP = temp_RIP
   EXIT
   } end INTn to more privileged level


INT_N_VIRTUAL:

temp_int_n_vector = byte-sized interrupt vector specified in
                    the instruction, zero-extended to 64 bits

IF (CR4.VME == 0)   // VME isn't enabled
   IF (RFLAGS.IOPL == 3)
```

```
      INT_N_VIRTUAL_TO_PROTECTED
   ELSE
      EXCEPTION [#GP(0)]

temp_IRB_BASE = READ_MEM.w [tss:102] - 32

// Check the VME Interrupt Redirection Bitmap (IRB) to
// see if we should redirect to a virtual-mode handler
temp_VME_REDIRECTION = READ_BIT_ARRAY ([tss:temp_IRB_BASE], temp_int_n_vector)
IF (temp_VME_REDIRECTION == 1)
   { // continue with transition to protected mode
   IF (RFLAGS.IOPL==3)
      INT_N_VIRTUAL_TO_PROTECTED
   ELSE
      EXCEPTION [#GP(0)]
   }
ELSE
   { // INTn stays in virtual mode
   // redirect interrupt through virtual-mode IDT
   temp_RIP = READ_MEM.w [0:temp_int_n_vector*4]
   // read target CS:RIP from the virtual-mode IDT at linear address 0
   temp_CS = READ_MEM.w [0:temp_int_n_vector*4+2]
   IF (RFLAGS.IOPL < 3)
      old_RFLAGS = old_RFLAGS with VIF bit shifted into IF bit, and IOPL = 3
   PUSH.w old_RFLAGS
   PUSH.w old_CS
   PUSH.w next_RIP
   CS.sel = temp_CS
   CS.base = temp_CS SHL 4
   RFLAGS.TF,RF = 0
   IF (RFLAGS.IOPL == 3)
      RFLAGS.IF = 0
   ELSE
      RFLAGS.VIF = 0
   RIP = temp_RIP
   EXIT
   }


INT_N_VIRTUAL_TO_PROTECTED:

temp_idt_desc = READ_IDT (temp_int_n_vector)
IF (temp_idt_desc.attr.type == 'taskgate')
   TASK_SWITCH // using tss selector in the task gate as the target tss

// The size of the gate controls the size of the stack pushes
IF ((temp_idt_desc.attr.type == 'intgate32') ||
    (temp_idt_desc.attr.type == 'trapgate32'))
   v = 4-byte             // legacy mode, using a 32-bit gate
ELSE                      // gate is intgate16 or trapgate16
   v = 2-byte             // legacy mode, using a 16-bit gate
```

```
temp_RIP = temp_idt_desc.offset
old_CPL = CPL
CS = READ_DESCRIPTOR(temp_idt_desc.segment, intcs_chk)

IF (CS.attr.dpl !=0 )      // Handler must run at CPL 0.
      EXCEPTION [#GP(CS.sel)]

CPL = 0
temp_ist = 0               // Legacy mode doesn't use IST pointers
temp_SS_desc:temp_RSP = READ_INNER_LEVEL_SP(CPL, temp_ist)
RSP = temp_RSP
SS  = temp_SS_desc

// Any #SS from the following pushes uses SS.sel as error code
PUSH.v old_GS
PUSH.v old_FS
PUSH.v old_DS
PUSH.v old_ES
PUSH.v old_SS
PUSH.v old_RSP
PUSH.v old_RFLAGS // Pushed with RF = 0
PUSH.v old_CS
PUSH.v next_RIP

IF (temp_RIP > CS.limit)
      EXCEPTION [#GP(0)]

DS = NULL // can't use virtual-mode selectors in protected mode
ES = NULL // can't use virtual-mode selectors in protected mode
FS = NULL // can't use virtual-mode selectors in protected mode
GS = NULL // can't use virtual-mode selectors in protected mode
RFLAGS.VM,NT,TF,RF cleared
RFLAGS.IF cleared if interrupt gate
RIP = temp_RIP

IF (ShadowStacksEnabled(CPL 0))
   {
   old_SSP = SSP
   SSP     = PL0_SSP         // fetch new SSP
   SET_SSTK_TOKEN_BUSY(SSP)  // vaidate token, set busy
   IF (old_CPL) != 3
      {
      SSTK_WRITE_MEM.q [SSP-24] = old_CS   // push CS, LIP, SSP
      SSTK_WRITE_MEM.q [SSP-16] = LIP      // onto the shadow stack
      SSTK_WRITE_MEM.q [SSP-8]  = old_SSP
      SSP = SSP - 24
      }
   }

EXIT   // end INTn VIRTUAL_TO_PROTECTED
```

Related Instructions

INT 3, INTO, BOUND

rFLAGS Affected

If a task switch occurs, all flags are modified. Otherwise settings are as follows:

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		M	M	M	0	M				M	0					
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid TSS, #TS (selector)		X	X	As part of a stack switch, the target stack segment selector or rSP in the TSS was beyond the TSS limit.
		X	X	As part of a stack switch, the target stack segment selector in the TSS was a null selector.
		X	X	As part of a stack switch, the target stack segment selector's TI bit was set, but the LDT selector was a null selector.
		X	X	As part of a stack switch, the target stack segment selector in the TSS was beyond the limit of the GDT or LDT descriptor table.
		X	X	As part of a stack switch, the target stack segment selector in the TSS contained a RPL that was not equal to its DPL.
		X	X	As part of a stack switch, the target stack segment selector in the TSS contained a DPL that was not equal to the CPL of the code segment selector.
		X	X	As part of a stack switch, the target stack segment selector in the TSS was not a writable segment.
Segment not present, #NP (selector)		X	X	The accessed code segment, interrupt gate, trap gate, task gate, or TSS was not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical, and no stack switch occurred.
Stack, #SS (selector)		X	X	After a stack switch, a memory address exceeded the stack segment limit or was non-canonical.
		X	X	As part of a stack switch, the SS register was loaded with a non-null segment selector and the segment was marked not present.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
		X		The IOPL was less than 3 and CR4.VME was 0.
		X		IOPL was less than 3, CR4.VME was 1, and the corresponding bit in the VME interrupt redirection bitmap was 1.
General protection, #GP (selector)	X	X	X	The interrupt vector was beyond the limit of IDT.
		X	X	The descriptor in the IDT was not an interrupt, trap, or task gate in legacy mode or not a 64-bit interrupt or trap gate in long mode.
		X	X	The DPL of the interrupt, trap, or task gate descriptor was less than the CPL.
		X	X	The segment selector specified by the interrupt or trap gate had its TI bit set, but the LDT selector was a null selector.
		X	X	The segment descriptor specified by the interrupt or trap gate exceeded the descriptor table limit or was a null selector.
		X	X	The segment descriptor specified by the interrupt or trap gate was not a code segment in legacy mode, or not a 64-bit code segment in long mode.
			X	The DPL of the segment specified by the interrupt or trap gate was greater than the CPL.
		X		The DPL of the segment specified by the interrupt or trap gate pointed was not 0 or it was a conforming segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

INTO Interrupt to Overflow Vector

Checks the overflow flag (OF) in the rFLAGS register and calls the overflow exception (#OF) handler if the OF flag is set to 1. This instruction has no effect if the OF flag is cleared to 0. The INTO instruction detects overflow in signed number addition. See *AMD64 Architecture Programmer's Manual Volume 1: Application Programming* for more information on the OF flag.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

For detailed descriptions of the steps performed by INT instructions, see the following:

- *Legacy-Mode Interrupts:* "Legacy Protected-Mode Interrupt Control Transfers" in Volume 2.
- *Long-Mode Interrupts:* "Long-Mode Interrupt Control Transfers" in Volume 2.

Mnemonic	Opcode	Description
INTO	CE	Call overflow exception if the overflow flag is set. (Invalid in 64-bit mode.)

Action

```
IF (64BIT_MODE)
   EXCEPTION[#UD]
IF (RFLAGS.OF == 1)     // #OF is a trap, and pushes the rIP of the instruction
   EXCEPTION [#OF]     // following INTO.
EXIT
```

Related Instructions

INT, INT 3, BOUND

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Overflow, #OF	X	X	X	The INTO instruction was executed with 0F set to 1.
Invalid opcode, #UD			X	Instruction was executed in 64-bit mode.

Jcc Jump on Condition

Checks the status flags in the rFLAGS register and, if the flags meet the condition specified by the condition code in the mnemonic (*cc*), jumps to the target instruction located at the specified relative offset. Otherwise, execution continues with the instruction following the J*cc* instruction.

Unlike the unconditional jump (JMP), conditional jump instructions have only two forms—*short* and *near conditional jumps*. Different opcodes correspond to different forms of one instruction. For example, the JO instruction (jump if overflow) has opcode 0Fh 80h for its near form and 70h for its short form, but the mnemonic is the same for both forms. The only difference is that the near form has a 16- or 32-bit relative displacement, while the short form always has an 8-bit relative displacement.

Mnemonics are provided to deal with the programming semantics of both signed and unsigned numbers. Instructions tagged A (above) and B (below) are intended for use in unsigned integer code; those tagged G (greater) and L (less) are intended for use in signed integer code.

If the jump is taken, the signed displacement is added to the rIP (of the following instruction) and the result is truncated to 16, 32, or 64 bits, depending on operand size.

In 64-bit mode, the operand size defaults to 64 bits. The processor sign-extends the 8-bit or 32-bit displacement value to 64 bits before adding it to the RIP.

These instructions cannot perform far jumps (to other code segments). To create a far-conditional-jump code sequence corresponding to a high-level language statement like:

```
IF A == B THEN GOTO FarLabel
```

where FarLabel is located in another code segment, use the opposite condition in a conditional short jump before an unconditional far jump. Such a code sequence might look like:

```
      cmp   A,B               ; compare operands
      jne   NextInstr         ; continue program if not equal
      jmp   far FarLabel      ; far jump if operands are equal

   NextInstr:                 ; continue program
```

For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
JO *rel8off* JO *rel16off* JO *rel32off*	70 cb 0F 80 *cw* 0F 80 *cd*	Jump if overflow (OF = 1).
JNO *rel8off* JNO *rel16off* JNO *rel32off*	71 *cb* 0F 81 *cw* 0F 81 *cd*	Jump if not overflow (OF = 0).
JB *rel8off* JB *rel16off* JB *rel32off*	72 *cb* 0F 82 *cw* 0F 82 *cd*	Jump if below (CF = 1).

Mnemonic	Opcode	Description
JC *rel8off* JC *rel16off* JC *rel32off*	72 *cb* 0F 82 *cw* 0F 82 *cd*	Jump if carry (CF = 1).
JNAE *rel8off* JNAE *rel16off* JNAE *rel32off*	72 *cb* 0F 82 *cw* 0F 82 *cd*	Jump if not above or equal (CF = 1).
JNB *rel8off* JNB *rel16off* JNB *rel32off*	73 *cb* 0F 83 *cw* 0F 83 *cd*	Jump if not below (CF = 0).
JNC *rel8off* JNC *rel16off* JNC *rel32off*	73 *cb* 0F 83 *cw* 0F 83 *cd*	Jump if not carry (CF = 0).
JAE *rel8off* JAE *rel16off* JAE *rel32off*	73 *cb* 0F 83 *cw* 0F 83 *cd*	Jump if above or equal (CF = 0).
JZ *rel8off* JZ *rel16off* JZ *rel32off*	74 *cb* 0F 84 *cw* 0F 84 *cd*	Jump if zero (ZF = 1).
JE *rel8off* JE *rel16off* JE *rel32off*	74 *cb* 0F 84 *cw* 0F 84 *cd*	Jump if equal (ZF = 1).
JNZ *rel8off* JNZ *rel16off* JNZ *rel32off*	75 *cb* 0F 85 *cw* 0F 85 *cd*	Jump if not zero (ZF = 0).
JNE *rel8off* JNE *rel16off* JNE *rel32off*	75 *cb* 0F 85 *cw* 0F 85 *cd*	Jump if not equal (ZF = 0).
JBE *rel8off* JBE *rel16off* JBE *rel32off*	76 *cb* 0F 86 *cw* 0F 86 *cd*	Jump if below or equal (CF = 1 or ZF = 1).
JNA *rel8off* JNA *rel16off* JNA *rel32off*	76 *cb* 0F 86 *cw* 0F 86 *cd*	Jump if not above (CF = 1 or ZF = 1).
JNBE *rel8off* JNBE *rel16off* JNBE *rel32off*	77 *cb* 0F 87 *cw* 0F 87 *cd*	Jump if not below or equal (CF = 0 and ZF = 0).
JA *rel8off* JA *rel16off* JA *rel32off*	77 *cb* 0F 87 *cw* 0F 87 *cd*	Jump if above (CF = 0 and ZF = 0).
JS *rel8off* JS *rel16off* JS *rel32off*	78 *cb* 0F 88 *cw* 0F 88 *cd*	Jump if sign (SF = 1).
JNS *rel8off* JNS *rel16off* JNS *rel32off*	79 *cb* 0F 89 *cw* 0F 89 *cd*	Jump if not sign (SF = 0).

Mnemonic	Opcode	Description
JP *rel8off* JP *rel16off* JP *rel32off*	7A *cb* 0F 8A *cw* 0F 8A *cd*	Jump if parity (PF = 1).
JPE *rel8off* JPE *rel16off* JPE *rel32off*	7A *cb* 0F 8A *cw* 0F 8A *cd*	Jump if parity even (PF = 1).
JNP *rel8off* JNP *rel16off* JNP *rel32off*	7B *cb* 0F 8B *cw* 0F 8B *cd*	Jump if not parity (PF = 0).
JPO *rel8off* JPO *rel16off* JPO *rel32off*	7B *cb* 0F 8B *cw* 0F 8B *cd*	Jump if parity odd (PF = 0).
JL *rel8off* JL *rel16off* JL *rel32off*	7C *cb* 0F 8C *cw* 0F 8C *cd*	Jump if less (SF <> OF).
JNGE *rel8off* JNGE *rel16off* JNGE *rel32off*	7C *cb* 0F 8C *cw* 0F 8C *cd*	Jump if not greater or equal (SF <> OF).
JNL *rel8off* JNL *rel16off* JNL *rel32off*	7D *cb* 0F 8D *cw* 0F 8D *cd*	Jump if not less (SF = OF).
JGE *rel8off* JGE *rel16off* JGE *rel32off*	7D *cb* 0F 8D *cw* 0F 8D *cd*	Jump if greater or equal (SF = OF).
JLE *rel8off* JLE *rel16off* JLE *rel32off*	7E *cb* 0F 8E *cw* 0F 8E *cd*	Jump if less or equal (ZF = 1 or SF <> OF).
JNG *rel8off* JNG *rel16off* JNG *rel32off*	7E *cb* 0F 8E *cw* 0F 8E *cd*	Jump if not greater (ZF = 1 or SF <> OF).
JNLE *rel8off* JNLE *rel16off* JNLE *rel32off*	7F *cb* 0F 8F *cw* 0F 8F *cd*	Jump if not less or equal (ZF = 0 and SF = OF).
JG *rel8off* JG *rel16off* JG *rel32off*	7F *cb* 0F 8F *cw* 0F 8F *cd*	Jump if greater (ZF = 0 and SF = OF).

Related Instructions

JMP (Near), JMP (Far), JrCXZ

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	The target offset exceeded the code segment limit or was non-canonical.

JCXZ
JECXZ
JRCXZ
Jump if rCX Zero

Checks the contents of the count register (rCX) and, if 0, jumps to the target instruction located at the specified 8-bit relative offset. Otherwise, execution continues with the instruction following the J*r*CXZ instruction.

The size of the count register (CX, ECX, or RCX) depends on the address-size attribute of the JrCXZ instruction. Therefore, JRCXZ can only be executed in 64-bit mode and JCXZ cannot be executed in 64-bit mode.

If the jump is taken, the signed displacement is added to the rIP (of the following instruction) and the result is truncated to 16, 32, or 64 bits, depending on operand size.

In 64-bit mode, the operand size defaults to 64 bits. The processor sign-extends the 8-bit displacement value to 64 bits before adding it to the RIP.

For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
JCXZ *rel8off*	E3 *cb*	Jump short if the 16-bit count register (CX) is zero.
JECXZ *rel8off*	E3 *cb*	Jump short if the 32-bit count register (ECX) is zero.
JRCXZ *rel8off*	E3 *cb*	Jump short if the 64-bit count register (RCX) is zero.

Related Instructions

J*cc*, JMP (Near), JMP (Far)

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	The target offset exceeded the code segment limit or was non-canonical

JMP (Near) Near Jump

Unconditionally transfers control to a new address without saving the current rIP value. This form of the instruction jumps to an address in the current code segment and is called a *near jump*. The target operand can specify a register, a memory location, or a label.

If the JMP target is specified in a register or memory location, then a 16-, 32-, or 64-bit rIP is read from the operand, depending on operand size. This rIP is zero-extended to 64 bits.

If the JMP target is specified by a displacement in the instruction, the signed displacement is added to the rIP (of the following instruction), and the result is truncated to 16, 32, or 64 bits depending on operand size. The signed displacement can be 8 bits, 16 bits, or 32 bits, depending on the opcode and the operand size.

For near jumps in 64-bit mode, the operand size defaults to 64 bits. The E9 opcode results in RIP = RIP + 32-bit signed displacement, and the FF /4 opcode results in RIP = 64-bit offset from register or memory. No prefix is available to encode a 32-bit operand size in 64-bit mode.

See JMP (Far) for information on far jumps—jumps to procedures located outside of the current code segment. For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
JMP *rel8off*	EB *cb*	Short jump with the target specified by an 8-bit signed displacement.
JMP *rel16off*	E9 *cw*	Near jump with the target specified by a 16-bit signed displacement.
JMP *rel32off*	E9 *cd*	Near jump with the target specified by a 32-bit signed displacement.
JMP *reg/mem16*	FF /4	Near jump with the target specified *reg/mem16*.
JMP *reg/mem32*	FF /4	Near jump with the target specified *reg/mem32*. (No prefix for encoding in 64-bit mode.)
JMP *reg/mem64*	FF /4	Near jump with the target specified *reg/mem64*.

Related Instructions

JMP (Far), J*cc*, J*r*CX

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

JMP (Far) Far Jump

Unconditionally transfers control to a new address without saving the current CS:rIP values. This form of the instruction jumps to an address outside the current code segment and is called a *far jump*. The operand specifies a target selector and offset.

The target operand can be specified by the instruction directly, by containing the far pointer in the jmp far opcode itself, or indirectly, by referencing a far pointer in memory. In 64-bit mode, only indirect far jumps are allowed, executing a direct far jmp (opcode EA) will generate an undefined opcode exception. For both direct and indirect far jumps, if the JMP (Far) operand-size is 16 bits, the instruction's operand is a 16-bit selector followed by a 16-bit offset. If the operand-size is 32 or 64 bits, the operand is a 16-bit selector followed by a 32-bit offset.

In all modes, the target selector used by the instruction can be a code selector. Additionally, the target selector can also be a call gate in protected mode, or a task gate or TSS selector in legacy protected mode.

- *Target is a code segment*—Control is transferred to the target CS:rIP. In this case, the target offset can only be a 16 or 32 bit value, depending on operand-size, and is zero-extended to 64 bits; 64-bit offsets are only available via call gates. No CPL change is allowed.
- *Target is a call gate*—The call gate specifies the actual target code segment and offset, and control is transferred to the target CS:rIP. When jumping through a call gate, the size of the target rIP is 16, 32, or 64 bits, depending on the size of the call gate. If the target rIP is less than 64 bits, it's zero-extended to 64 bits. In long mode, only 64-bit call gates are allowed, and they must point to 64-bit code segments. No CPL change is allowed.
- *Target is a task gate or a TSS*—If the mode is legacy protected mode, then a task switch occurs. See “Hardware Task-Management in Legacy Mode” in volume 2 for details about task switches. Hardware task switches are not supported in long mode.

See JMP (Near) for information on near jumps—jumps to procedures located inside the current code segment. For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
JMP FAR *pntr16:16*	EA *cd*	Far jump direct, with the target specified by a far pointer contained in the instruction. (Invalid in 64-bit mode.)
JMP FAR *pntr16:32*	EA *cp*	Far jump direct, with the target specified by a far pointer contained in the instruction. (Invalid in 64-bit mode.)
JMP FAR *mem16:16*	FF /5	Far jump indirect, with the target specified by a far pointer in memory (16-bit operand size).
JMP FAR *mem16:32*	FF /5	Far jump indirect, with the target specified by a far pointer in memory (32- and 64-bit operand size).

Action

```
// Far jumps (JMPF)
// See “Pseudocode Definition” on page 57.

JMPF_START:

IF (REAL_MODE)
    JMPF_REAL_OR_VIRTUAL
ELSIF (PROTECTED_MODE)
    JMPF_PROTECTED
ELSE // (VIRTUAL_MODE)
    JMPF_REAL_OR_VIRTUAL


JMPF_REAL_OR_VIRTUAL:

    IF (OPCODE == jmpf [mem]) //JMPF Indirect
    {
        temp_RIP = READ_MEM.z [mem]
        temp_CS  = READ_MEM.w [mem+Z]
    }
    ELSE // (OPCODE == jmpf direct)
    {
        temp_RIP = z-sized offset specified in the instruction,
                   zero-extended to 64 bits
        temp_CS  = selector specified in the instruction
    }

    IF (temp_RIP>CS.limit)
        EXCEPTION [#GP(0)]

    CS.sel  = temp_CS
    CS.base = temp_CS SHL 4
    RIP = temp_RIP
    EXIT

JMPF_PROTECTED:
    IF (OPCODE == jmpf [mem]) // JMPF Indirect
    {
        temp_offset = READ_MEM.z [mem]
        temp_sel    = READ_MEM.w [mem+Z]
    }
    ELSE // (OPCODE == jmpf direct)
    {
        IF (64BIT_MODE)
           EXCEPTION [#UD]          // ’jmpf direct’ is illegal in 64-bit mode

        temp_offset = z-sized offset specified in the instruction,
                      zero-extended to 64 bits
        temp_sel    = selector specified in the instruction
    }
```

```
temp_desc = READ_DESCRIPTOR (temp_sel, cs_chk)
                    // read descriptor, perform protection and type checks

IF (temp_desc.attr.type == 'available_tss')
    TASK_SWITCH      // using temp_sel as the target tss selector
ELSIF (temp_desc.attr.type == 'taskgate')
    TASK_SWITCH      // using the tss selector in the task gate as the
                     // target tss
ELSIF (temp_desc.attr.type == 'code')
                     // if the selector refers to a code descriptor, then
                     // the offset we read is the target RIP
{
    temp_RIP = temp_offset
    CS = temp_desc
    IF ((!64BIT_MODE) && (temp_RIP > CS.limit))
                     // temp_RIP can't be non-canonical because
                   // it's a 16- or 32-bit offset, zero-extended to 64 bits
    {
        EXCEPTION [#GP(0)]
    }
    RIP = temp_RIP
    EXIT
}
ELSE
{
       // (temp_desc.attr.type == 'callgate')
       // if the selector refers to a call gate, then
       // the target CS and RIP both come from the call gate
    temp_RIP = temp_desc.offset

    IF (LONG_MODE)
    {
      // in long mode, we need to read the 2nd half of a 16-byte call-gate
       // from the gdt/ldt to get the upper 32 bits of the target RIP
        temp_upper = READ_MEM.q [temp_sel+8]
        IF (temp_upper's extended attribute bits != 0)
            EXCEPTION [#GP(temp_sel)]       // Make sure the extended
                                           // attribute bits are all zero.

        temp_RIP = tempRIP + (temp_upper SHL 32)
                         // concatenate both halves of RIP
    }
    CS = READ_DESCRIPTOR (temp_desc.segment, clg_chk)
                         // set up new CS base, attr, limits
    IF ((64BIT_MODE) && (temp_RIP is non-canonical)
       || (!64BIT_MODE) && (temp_RIP > CS.limit))
        EXCEPTION [#GP(0)]
    RIP = temp_RIP
    EXIT
}
```

Related Instructions

JMP (Near), J*cc*, J*r*CX

rFLAGS Affected

None, unless a task switch occurs, in which case all flags are modified.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The far JUMP indirect opcode (FF /5) had a register operand.
			X	The far JUMP direct opcode (EA) was executed in 64-bit mode.
Segment not present, #NP (selector)			X	The accessed code segment, call gate, task gate, or TSS was not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
			X	A null data segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP (selector)			X	The target code segment selector was a null selector.
			X	A code, call gate, task gate, or TSS descriptor exceeded the descriptor table limit.
			X	A segment selector's TI bit was set, but the LDT selector was a null selector.
			X	The segment descriptor specified by the instruction was not a code segment, task gate, call gate or available TSS in legacy mode, or not a 64-bit code segment or a 64-bit call gate in long mode.
			X	The RPL of the non-conforming code segment selector specified by the instruction was greater than the CPL, or its DPL was not equal to the CPL.
			X	The DPL of the conforming code segment descriptor specified by the instruction was greater than the CPL.
			X	The DPL of the callgate, taskgate, or TSS descriptor specified by the instruction was less than the CPL or less than its own RPL.
			X	The segment selector specified by the call gate or task gate was a null selector.
			X	The segment descriptor specified by the call gate was not a code segment in legacy mode or not a 64-bit code segment in long mode.
			X	The DPL of the segment descriptor specified the call gate was greater than the CPL and it is a conforming segment.
			X	The DPL of the segment descriptor specified by the callgate was not equal to the CPL and it is a non-conforming segment.
			X	The 64-bit call gate's extended attribute bits were not zero.
			X	The TSS descriptor was found in the LDT.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

LAHF Load Status Flags into AH Register

Loads the lower 8 bits of the rFLAGS register, including sign flag (SF), zero flag (ZF), auxiliary carry flag (AF), parity flag (PF), and carry flag (CF), into the AH register.

The instruction sets the reserved bits 1, 3, and 5 of the rFLAGS register to 1, 0, and 0, respectively, in the AH register.

The LAHF instruction is available in 64-bit mode if CPUID Fn8000_0001_ECX[LahfSahf] = 1. It is always available in the other operating modes (including compatibility mode)

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Opcode	Description
LAHF	9F	Load the SF, ZF, AF, PF, and CF flags into the AH register.

Related Instructions

SAHF

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	The LAHF instruction is not supported in 64-bit mode, as indicated by CPUID Fn8000_0001_ECX[LahfSahf] = 0.

LDS
LES
LFS
LGS
LSS

Load Far Pointer

Loads a far pointer from a memory location (second operand) into a segment register (mnemonic) and general-purpose register (first operand). The instruction stores the 16-bit segment selector of the pointer into the segment register and the 16-bit or 32-bit offset portion into the general-purpose register. The operand-size attribute determines whether the pointer loaded is 32 or 48 bits in length. A 64-bit operand is not supported.

These instructions load associated segment-descriptor information into the hidden portion of the specified segment register.

Mnemonic	Opcode	Description
LDS *reg16, mem16:16*	C5 */r*	Load DS:reg16 with a far pointer from memory. [Redefined as VEX (2-byte prefix) in 64-bit mode.]
LDS *reg32, mem16:32*	C5 */r*	Load DS:reg32 with a far pointer from memory. [Redefined as VEX (2-byte prefix) in 64-bit mode.]
LES *reg16, mem16:16*	C4 */r*	Load ES:reg16 with a far pointer from memory. [Redefined as VEX (3-byte prefix) in 64-bit mode.]
LES *reg32, mem16:32*	C4 */r*	Load ES:reg32 with a far pointer from memory. [Redefined as VEX (3-byte prefix) in 64-bit mode.]
LFS *reg16, mem16:16*	0F B4 */r*	Load FS:reg16 with a 32-bit far pointer from memory.
LFS *reg32, mem16:32*	0F B4 */r*	Load FS:reg32 with a 48-bit far pointer from memory.
LGS *reg16, mem16:16*	0F B5 */r*	Load GS:reg16 with a 32-bit far pointer from memory.
LGS *reg32, mem16:32*	0F B5 */r*	Load GS:reg32 with a 48-bit far pointer from memory.
LSS *reg16, mem16:16*	0F B2 */r*	Load SS:*reg16* with a 32-bit far pointer from memory.
LSS *reg32, mem16:32*	0F B2 */r*	Load SS:*reg32* with a 48-bit far pointer from memory.

Related Instructions

None

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The source operand was a register.
			X	LDS or LES was executed in 64-bit mode and not subject to interpretation as a VEX prefix.
Segment not present, #NP (selector)			X	The DS, ES, FS, or GS register was loaded with a non-null segment selector and the segment was marked not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Stack, #SS (selector)			X	The SS register was loaded with a non-null segment selector and the segment was marked not present.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
General protection, #GP (selector)			X	A segment register was loaded, but the segment descriptor exceeded the descriptor table limit.
			X	A segment register was loaded and the segment selector's TI bit was set, but the LDT selector was a null selector.
			X	The SS register was loaded with a null segment selector in non-64-bit mode or while CPL = 3.
			X	The SS register was loaded and the segment selector RPL and the segment descriptor DPL were not equal to the CPL.
			X	The SS register was loaded and the segment pointed to was not a writable data segment.
			X	The DS, ES, FS, or GS register was loaded and the segment pointed to was a data or non-conforming code segment, but the RPL or CPL was greater than the DPL.
			X	The DS, ES, FS, or GS register was loaded and the segment pointed to was not a data segment or readable code segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

LEA Load Effective Address

Computes the effective address of a memory location (second operand) and stores it in a general-purpose register (first operand).

The address size of the memory location and the size of the register determine the specific action taken by the instruction, as follows:

- If the address size and the register size are the same, the instruction stores the effective address as computed.
- If the address size is longer than the register size, the instruction truncates the effective address to the size of the register.
- If the address size is shorter than the register size, the instruction zero-extends the effective address to the size of the register.

If the second operand is a register, an undefined-opcode exception occurs.

The LEA instruction is related to the MOV instruction, which copies data from a memory location to a register, but LEA takes the address of the source operand, whereas MOV takes the contents of the memory location specified by the source operand. In the simplest cases, LEA can be replaced with MOV. For example:

```
lea eax, [ebx]
```

has the same effect as:

```
mov eax, ebx
```

However, LEA allows software to use any valid ModRM and SIB addressing mode for the source operand. For example:

```
lea eax, [ebx+edi]
```

loads the sum of the EBX and EDI registers into the EAX register. This could not be accomplished by a single MOV instruction.

The LEA instruction has a limited capability to perform multiplication of operands in general-purpose registers using scaled-index addressing. For example:

```
lea eax, [ebx+ebx*8]
```

loads the value of the EBX register, multiplied by 9, into the EAX register. Possible values of multipliers are 2, 4, 8, 3, 5, and 9.

The LEA instruction is widely used in string-processing and array-processing to initialize an index register (rSI or rDI) before performing string instructions such as MOVSx. It is also used to initialize the rBX register before performing the XLAT instruction in programs that perform character translations. In data structures, the LEA instruction can calculate addresses of operands stored in memory, and in particular, addresses of array or string elements.

Mnemonic	Opcode	Description
LEA *reg16*, *mem*	8D */r*	Store effective address in a 16-bit register.
LEA *reg32, mem*	8D */r*	Store effective address in a 32-bit register.
LEA *reg64, mem*	8D */r*	Store effective address in a 64-bit register.

Related Instructions

MOV

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The source operand was a register.

LEAVE Delete Procedure Stack Frame

Releases a stack frame created by a previous ENTER instruction. To release the frame, it copies the frame pointer (in the rBP register) to the stack pointer register (rSP), and then pops the old frame pointer from the stack into the rBP register, thus restoring the stack frame of the calling procedure.

The 32-bit LEAVE instruction is equivalent to the following 32-bit operation:

```
MOV ESP,EBP
POP EBP
```

To return program control to the calling procedure, execute a RET instruction after the LEAVE instruction.

In 64-bit mode, the LEAVE operand size defaults to 64 bits, and there is no prefix available for encoding a 32-bit operand size.

Mnemonic	Opcode	Description
LEAVE	C9	Set the stack pointer register SP to the value in the BP register and pop BP.
LEAVE	C9	Set the stack pointer register ESP to the value in the EBP register and pop EBP. (No prefix for encoding this in 64-bit mode.)
LEAVE	C9	Set the stack pointer register RSP to the value in the RBP register and pop RBP.

Related Instructions

ENTER

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

LFENCE Load Fence

Acts as a barrier to force strong memory ordering (serialization) between load instructions preceding the LFENCE and load instructions that follow the LFENCE. Loads from differing memory types may be performed out of order, in particular between WC/WC+ and other memory types. The LFENCE instruction assures that the system completes all previous loads before executing subsequent loads.

The LFENCE instruction is weakly-ordered with respect to store instructions, data and instruction prefetches, and the SFENCE instruction. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around an LFENCE.

In addition to load instructions, the LFENCE instruction is strongly ordered with respect to other LFENCE instructions, as well as MFENCE and other serializing instructions. Further details on the use of MFENCE to order accesses among differing memory types may be found in *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, section 7.4 "Memory Types" on page 172.

LFENCE is an SSE2 instruction. Support for SSE2 instructions is indicated by CPUID Fn0000_0001_EDX[SSE2] = 1.

In some systems, LFENCE may be configured to be dispatch serializing. In systems where CPUID Fn8000_0021_EAX[LFenceAlwaysSerializing](bit 2) = 1, LFENCE is always dispatch serializing.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, "Instruction Subsets and CPUID Feature Flags," on page 587.

Mnemonic	Opcode	Description
LFENCE	0F AE E8	Force strong ordering of (serialize) load operations.

Related Instructions

MFENCE, SFENCE, MCOMMIT

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.

LLWPCB Load Lightweight Profiling Control Block Address

Parses the Lightweight Profiling Control Block at the address contained in the specified register. If the LWPCB is valid, writes the address into the LWP_CBADDR MSR and enables Lightweight Profiling.

See Volume 2, Chapter 13, for an overview of the lightweight profiling facility.

The LWPCB must be in memory that is readable and writable in user mode. For better performance, it should be aligned on a 64-byte boundary in memory and placed so that it does not cross a page boundary, though neither of these suggestions is required.

The LWPCB address in the register is truncated to 32 bits if the operand size is 32.

Action

1. If LWP is not available or if the machine is not in protected mode, LLWPCB immediately causes a #UD exception.
2. If LWP is already enabled, the processor flushes the LWP state to memory in the old LWPCB. See description of the SLWPCB instruction on page 340 for details on saving the active LWP state.

   If the flush causes a #PF exception, LWP remains enabled with the old LWPCB still active. Note that the flush is done before LWP attempts to access the new LWPCB.
3. If the specified LWPCB address is 0, LWP is disabled and the execution of LLWPCB is complete.
4. The LWPCB address is non-zero. LLWPCB validates it as follows:
   - If any part of the LWPCB or the ring buffer is beyond the data segment limit, LLWPCB causes a #GP exception.
   - If the ring buffer size is below the implementation’s minimum ring buffer size, LLWPCB causes a #GP exception.
   - While doing these checks, LWP reads and writes the LWPCB, which may cause a #PF exception.

   If any of these exceptions occurs, LLWPCB aborts and LWP is left disabled. Usually, the operating system will handle a #PF exception by making the memory available and returning to retry the LLWPCB instruction. The #GP exceptions indicate application programming errors.
5. LWP converts the LWPCB address and the ring buffer address to linear address form by adding the DS base address and stores the addresses internally.
6. LWP examines the LWPCB.Flags field to determine which events should be enabled and whether threshold interrupts should be taken. It clears the bits for any features that are not available and stores the result back to LWPCB.Flags to inform the application of the actual LWP state.
7. For each event being enabled, LWP examines the EventInterval*n* value and, if necessary, sets it to an implementation-defined minimum. (The minimum event interval for LWPVAL is zero.) It loads its internal counter for the event from the value in EventCounter*n*. A zero or negative value

in EventCounter*n* means that the next event of that type will cause an event record to be stored. To count every j^{th} event, a program should set EventInterval*n* to *j-1* and EventCounter*n* to some starting value (where *j-1* is a good initial count). If the counter value is larger than the interval, the first event record will be stored after a larger number of events than subsequent records.

8. LWP is started. The execution of LLWPCB is complete.

Notes

If none of the bits in the LWPCB.Flags specifies an available event, LLWPCB still enables LWP to allow the use of the LWPINS instruction. However, no other event records will be stored.

A program can temporarily disable LWP by executing SLWPCB to obtain the current LWPCB address, saving that value, and then executing LLWPCB with a register containing 0. It can later re-enable LWP by executing LLWPCB with a register containing the saved address.

When LWP is enabled, it is typically an error to execute LLWPCB with the address of the active LWPCB. When the hardware flushes the existing LWP state into the LWPCB, it may overwrite fields that the application may have set to new LWP parameter values. The flushed values will then be loaded as LWP is restarted. To reuse an LWPCB, an application should stop LWP by passing a zero to LLWPCB, then prepare the LWPCB with new parameters and execute LLWPCB again to restart LWP.

Internally, LWP keeps the linear address of the LWPCB and the ring buffer. If the application changes the value of DS, LWP will continue to collect samples even if the new DS value would no longer allow access the LWPCB or the ring buffer. However, a #GP fault will occur if the application uses XRSTOR to restore LWP state saved by XSAVE. Programs should avoid using XSAVE/XRSTOR on LWP state if DS has changed. This only applies when the CPL != 0; kernel mode operation of XRSTOR is unaffected by changes to DS. See instruction listing for XSAVE in Volume 4 for details.

Operating system and hypervisor code that runs when CPL ≠ 3 should use XSAVE and XRSTOR to control LWP rather than using LLWPCB. Use WRMSR to write 0 to the LWP_CBADDR MSR to immediately stop LWP without saving its current state.

It is possible to execute LLWPCB when the CPL != 3 or when SMM is active, but the system software must ensure that the LWPCB and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Furthermore, if LWP is enabled when a kernel executes LLWPCB, both the old and new control blocks and ring buffers must be accessible. Using LLWPCB in these situations is not recommended.

LLWPCB is an LWP instruction. Support for LWP instructions is indicated by CPUID Fn8000_0001_ECX[LWP] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
LLWPCB *reg32*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	12 /0
LLWPCB *reg64*	8F	$\overline{\text{RXB}}$.09	1.1111.0.00	12 /0

ModRM.reg augments the opcode and is assigned the value 0. ModRM.r/m (augmented by XOP.R) specifies the register containing the effective address of the LWPCB. ModRM.mod is 11b.

Related Instructions

SLWPCB, LWPVAL, LWPINS

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	LWP instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[LWP] = 0.
	X	X		The system is not in protected mode.
			X	LWP is not available, or mod != 11b, or vvvv != 1111b.
General protection, #GP			X	Any part of the LWPCB or the event ring buffer is beyond the DS segment limit.
			X	Any restrictions on the contents of the LWPCB are violated
Page fault, #PF			X	A page fault resulted from reading or writing the LWPCB.
			X	LWP was already enabled and a page fault resulted from reading or writing the old LWPCB.
			X	LWP was already enabled and a page fault resulted from flushing an event to the old ring buffer.

LODS
LODSB
LODSW
LODSD
LODSQ

Load String

Copies the byte, word, doubleword, or quadword in the memory location pointed to by the DS:rSI registers to the AL, AX, EAX, or RAX register, depending on the size of the operand, and then increments or decrements the rSI register according to the state of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments rSI; otherwise, it decrements rSI. It increments or decrements rSI by 1, 2, 4, or 8, depending on the number of bytes being loaded.

The forms of the LODS instruction with an explicit operand address the operand at *seg*:[rSI]. The value of *seg* defaults to the DS segment, but may be overridden by a segment prefix. The explicit operand serves only to specify the type (size) of the value being copied and the specific registers used.

The no-operands forms of the instruction always use the DS:[rSI] registers to point to the value to be copied (they do not allow a segment prefix). The mnemonic determines the size of the operand and the specific registers used.

The LODS*x* instructions support the REP prefixes. For details about the REP prefixes, see “Repeat Prefixes” on page 12. More often, software uses the LODS*x* instruction inside a loop controlled by a LOOP*cc* instruction as a more efficient replacement for instructions like:

```
mov eax, dword ptr ds:[esi]
add esi, 4
```

The LODSQ instruction can only be used in 64-bit mode.

Mnemonic	Opcode	Description
LODS *mem8*	AC	Load byte at DS:rSI into AL and then increment or decrement rSI.
LODS *mem16*	AD	Load word at DS:rSI into AX and then increment or decrement rSI.
LODS *mem32*	AD	Load doubleword at DS:rSI into EAX and then increment or decrement rSI.
LODS *mem64*	AD	Load quadword at DS:rSI into RAX and then increment or decrement rSI.
LODSB	AC	Load byte at DS:rSI into AL and then increment or decrement rSI.
LODSW	AD	Load the word at DS:rSI into AX and then increment or decrement rSI.

Mnemonic	Opcode	Description
LODSD	AD	Load doubleword at DS:rSI into EAX and then increment or decrement rSI.
LODSQ	AD	Load quadword at DS:rSI into RAX and then increment or decrement rSI.

Related Instructions

MOVS*x*, STOS*x*

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

LOOP
LOOPE
LOOPNE
LOOPNZ
LOOPZ

Loop

Decrements the count register (rCX) by 1, then, if rCX is not 0 and the ZF flag meets the condition specified by the mnemonic, it jumps to the target instruction specified by the signed 8-bit relative offset. Otherwise, it continues with the next instruction after the LOOP*cc* instruction.

The size of the count register used (CX, ECX, or RCX) depends on the address-size attribute of the LOOP*cc* instruction.

The LOOP instruction ignores the state of the ZF flag.

The LOOPE and LOOPZ instructions jump if rCX is not 0 and the ZF flag is set to 1. In other words, the instruction exits the loop (falls through to the next instruction) if rCX becomes 0 or ZF = 0.

The LOOPNE and LOOPNZ instructions jump if rCX is not 0 and ZF flag is cleared to 0. In other words, the instruction exits the loop if rCX becomes 0 or ZF = 1.

The LOOP*cc* instruction does not change the state of the ZF flag. Typically, the loop contains a compare instruction to set or clear the ZF flag.

If the jump is taken, the signed displacement is added to the rIP (of the following instruction) and the result is truncated to 16, 32, or 64 bits, depending on operand size.

In 64-bit mode, the operand size defaults to 64 bits without the need for a REX prefix, and the processor sign-extends the 8-bit offset before adding it to the RIP.

Mnemonic	Opcode	Description
LOOP *rel8off*	E2 *cb*	Decrement rCX, then jump short if rCX is not 0.
LOOPE *rel8off*	E1 *cb*	Decrement rCX, then jump short if rCX is not 0 and ZF is 1.
LOOPNE *rel8off*	E0 *cb*	Decrement rCX, then Jump short if rCX is not 0 and ZF is 0.
LOOPNZ *rel8off*	E0 *cb*	Decrement rCX, then Jump short if rCX is not 0 and ZF is 0.
LOOPZ *rel8off*	E1 *cb*	Decrement rCX, then Jump short if rCX is not 0 and ZF is 1.

Related Instructions

None

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	The target offset exceeded the code segment limit or was non-canonical.

LWPINS Lightweight Profiling Insert Record

Inserts programmed event record into the LWP event ring buffer in memory and advances the ring buffer pointer.

Refer to the description of the programmed event record in Volume 2, Chapter 13. The record has an EventId of 255. The value in the register specified by vvvv (first operand) is stored in the Data2 field at bytes 23–16 (zero extended if the operand size is 32). The value in a register or memory location (second operand) is stored in the Data1 field at bytes 7–4. The immediate value (third operand) is truncated to 16 bits and stored in the Flags field at bytes 3–2.

If the ring buffer is not full, or if LWP is running in Continuous Mode, the head pointer is advanced and the CF flag is cleared. If the ring buffer threshold is exceeded and threshold interrupts are enabled, an interrupt is signaled. If LWP is in Continuous Mode and the new head pointer equals the tail pointer, the MissedEvents counter is incremented to indicate that the buffer wrapped.

If the ring buffer is full and LWP is running in Synchronized Mode, the event record overwrites the last record in the buffer, the MissedEvents counter in the LWPCB is incremented, the head pointer is not advanced, and the CF flag is set.

LWPINS generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

LWPINS simply clears CF if LWP is not enabled. This allows LWPINS instructions to be harmlessly ignored if profiling is turned off.

It is possible to execute LWPINS when the CPL $\neq 3$ or when SMM is active, but the system software must ensure that the memory operand (if present), the LWPCB, and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Using LWPINS in these situations is not recommended.

LWPINS can be used by a program to mark significant events in the ring buffer as they occur. For instance, a program might capture information on changes in the process' address space such as library loads and unloads, or changes in the execution environment such as a change in the state of a user-mode thread of control.

Note that when the LWPINS instruction finishes writing a event record in the event ring buffer, it counts as an instruction retired. If the Instructions Retired event is active, this might cause that counter to become negative and immediately store another event record with the same instruction address (but different EventId values).

LWPINS is an LWP instruction. Support for LWP instructions is indicated by CPUID Fn8000_0001_ECX[LWP] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, "Instruction Subsets and CPUID Feature Flags," on page 587.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
LWPINS *reg32.vvvv, reg/mem32, imm32*	8F	$\overline{\text{RXB}}$.0A	0.$\overline{\text{src1}}$.0.00	12 /0 /imm32
LWPINS *reg64.vvvv, reg/mem32, imm32*	8F	$\overline{\text{RXB}}$.0A	1.$\overline{\text{src1}}$.0.00	12 /0 /imm32

ModRM.reg augments the opcode and is assigned the value 0. The {mod, r/m} field of the ModRM byte (augmented by XOP.R) encodes the second operand. A 4-byte immediate field follows ModRM.

Related Instructions

LLWPCB, SLWPCB, LWPVAL

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
																M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	LWP instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[LWP] = 0.
	X	X		The system is not in protected mode.
			X	LWP is not available.
Page fault, #PF			X	A page fault resulted from reading or writing the LWPCB.
			X	A page fault resulted from writing the event to the ring buffer.
			X	A page fault resulted from reading a modrm operand from memory.
General protection, #GP			X	A modrm operand in memory exceeded the segment limit.

LWPVAL Lightweight Profiling Insert Value

Decrements the event counter associated with the programmed value sample event (see “Programmed Value Sample” in Volume 2, Chapter 13). If the resulting counter value is negative, inserts an event record into the LWP event ring buffer in memory and advances the ring buffer pointer.

Refer to the description of the programmed value sample record in Volume 2, Chapter 13. The event record has an EventId of 1. The value in the register specified by vvvv (first operand) is stored in the Data2 field at bytes 23–16 (zero extended if the operand size is 32). The value in a register or memory location (second operand) is stored in the Data1 field at bytes 7–4. The immediate value (third operand) is truncated to 16 bits and stored in the Flags field at bytes 3–2.

If the programmed value sample record is not written to the event ring buffer, the memory location of the second operand (assuming it is memory-based) is not accessed.

If the ring buffer is not full or if LWP is running in continuous mode, the head pointer is advanced and the event counter is reset to the interval for the event (subject to randomization). If the ring buffer threshold is exceeded and threshold interrupts are enabled, an interrupt is signaled. If LWP is in Continuous Mode and the new head pointer equals the tail pointer, the MissedEvents counter is incremented to indicate that the buffer wrapped.

If the ring buffer is full and LWP is running in Synchronized Mode, the event record overwrites the last record in the buffer, the MissedEvents counter in the LWPCB is incremented, and the head pointer is not advanced.

LWPVAL generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

LWPVAL does nothing if LWP is not enabled or if the Programmed Value Sample event is not enabled in LWPCB.Flags. This allows LWPVAL instructions to be harmlessly ignored if profiling is turned off.

It is possible to execute LWPVAL when the CPL != 3 or when SMM is active, but the system software must ensure that the memory operand (if present), the LWPCB, and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Using LWPVAL in these situations is not recommended.

LWPVAL can be used by a program to perform value profiling. This is the technique of sampling the value of some program variable at a predetermined frequency. For example, a managed runtime might use LWPVAL to sample the value of the divisor for a frequently executed divide instruction in order to determine whether to generate specialized code for a common division. It might sample the target location of an indirect branch or call to see if one destination is more frequent than others. Since LWPVAL does not modify any registers or condition codes, it can be inserted harmlessly between any instructions.

Note

When LWPVAL completes (whether or not it stored an event record in the event ring buffer), it counts as an instruction retired. If the Instructions Retired event is active, this might cause that counter to become negative and immediately store an event record. If LWPVAL also stored an event record, the buffer will contain two records with the same instruction address (but different EventId values).

LWPVAL is an LWP instruction. Support for LWP instructions is indicated by CPUID Fn8000_0001_ECX[LWP] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
LWPVAL *reg32.vvvv, reg/mem32, imm32*	8F	$\overline{\text{RXB}}$.0A	0.$\overline{\text{src1}}$.0.00	12 /1 /imm32
LWPVAL *reg64.vvvv, reg/mem32, imm32*	8F	$\overline{\text{RXB}}$.0A	1.$\overline{\text{src1}}$.0.00	12 /1 /imm32

ModRM.reg augments the opcode and is assigned the value 001b. The {mod, r/m} field of the ModRM byte (augmented by XOP.R) encodes the second operand. A four-byte immediate field follows ModRM.

Related Instructions

LLWPCB, SLWPCB, LWPINS

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	LWP instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[LWP] = 0.
	X	X		The system is not in protected mode.
			X	LWP is not available.
Page fault, #PF			X	A page fault resulted from reading or writing the LWPCB.
			X	A page fault resulted from writing the event to the ring buffer.
			X	A page fault resulted from reading a modrm operand from memory.
General protection, #GP			X	A modrm operand in memory exceeded the segment limit.

LZCNT Count Leading Zeros

Counts the number of leading zero bits in the 16-, 32-, or 64-bit general purpose register or memory source operand. Counting starts downward from the most significant bit and stops when the highest bit having a value of 1 is encountered or when the least significant bit is encountered. The count is written to the destination register.

This instruction has two operands:

LZCNT dest, src

If the input operand is zero, CF is set to 1 and the size (in bits) of the input operand is written to the destination register. Otherwise, CF is cleared.

If the most significant bit is a one, the ZF flag is set to 1, zero is written to the destination register. Otherwise, ZF is cleared.

LZCNT is an Advanced Bit Manipulation (ABM) instruction. Support for the LZCNT instruction is indicated by CPUID Fn8000_0001_ECX[ABM] = 1. If the LZCNT instruction is not available, the encoding is interpreted as the BSR instruction. Software MUST check the CPUID bit once per program or library initialization before using the LZCNT instruction, or inconsistent behavior may result.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic		Opcode	Description
LZCNT	*reg16*, *reg/mem16*	F3 0F BD /r	Count the number of leading zeros in reg/mem16.
LZCNT	*reg32*, *reg/mem32*	F3 0F BD /r	Count the number of leading zeros in reg/mem32.
LZCNT	*reg64*, *reg/mem64*	F3 0F BD /r	Count the number of leading zeros in reg/mem64.

Related Instructions

ANDN, BEXTR, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, POPCNT, T1MSKC, TZCNT, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MCOMMIT Commit Stores to Memory

MCOMMIT provides a fencing and error detection capability for stores to system memory components that have delayed error reporting. Execution of MCOMMIT ensures that any preceding stores in the thread to such memory components have completed (target locations written, unless inhibited by an error condition) and that any errors encountered by those stores have been signaled to associated error logging resources. If any such errors are present, MCOMMIT will clear rFLAGS.CF to zero, otherwise it will set rFLAGS.CF to one.

These errors are specific to the design of the platform and are reported only via MCOMMIT and in associated error logging registers on the platform; they are not visible to the Machine Check Architecture. Execution of MCOMMIT does not change any state in the error logging resources. Any error indications will need to be cleared by privileged software before MCOMMIT can return an error-free indication. Details on the error logging mechanisms may be found in the Processor Programming Reference manual for any product that supports this technology and the MCOMMIT instruction.

The MCOMMIT instruction is supported if the feature flag CPUID Fn8000_0008_EBX[MCOMMIT] =1 (bit 8). The MCOMMIT instruction must be explicitly enabled by the OS by setting EFER.MCOMMIT=1 (EFER bit 17), otherwise attempted execution of MCOMMIT will result in a #UD exception.

MCOMMIT uses the same ordering rules as the SFENCE instruction. It may be executed at any privilege level.

Instruction Encoding

Mnemonic	Opcode	Description
MCOMMIT	F3 0F 01 FA	Commit stores to memory

Related Instructions

LFENCE, SFENCE, MFENCE

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	0	0	0	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

MFENCE Memory Fence

Acts as a barrier to force strong memory ordering (serialization) between load and store instructions preceding the MFENCE, and load and store instructions that follow the MFENCE. The processor may perform loads out of program order with respect to non-conflicting stores for certain memory types. The MFENCE instruction ensures that the system completes all previous memory accesses before executing subsequent accesses.

The MFENCE instruction is weakly-ordered with respect to data and instruction prefetches. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around an MFENCE.

In addition to load and store instructions, the MFENCE instruction is strongly ordered with respect to other MFENCE instructions, LFENCE instructions, SFENCE instructions, serializing instructions, and CLFLUSH instructions. Further details on the use of MFENCE to order accesses among differing memory types may be found in *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, see 7.4 "Memory Types" on page 172.

The MFENCE instruction is a serializing instruction.

MFENCE is an SSE2 instruction. Support for SSE2 instructions is indicated by CPUID Fn0000_0001_EDX[SSE2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, "Instruction Subsets and CPUID Feature Flags," on page 587.

Instruction Encoding

Mnemonic	Opcode	Description
MFENCE	0F AE F0	Force strong ordering of (serialized) load and store operations.

Related Instructions

LFENCE, SFENCE, MCOMMIT

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.

MONITORX Setup Monitor Address

Establishes a linear address range of memory for hardware to monitor and puts the processor in the monitor event pending state. When in the monitor event pending state, the monitoring hardware detects stores to the specified linear address range and causes the processor to exit the monitor event pending state. The MWAIT and MWAITX instructions use the state of the monitor hardware.

The address range should be a write-back memory type. Executing MONITORX on an address range for a non-write-back memory type is not guaranteed to cause the processor to enter the monitor event pending state. The size of the linear address range that is established by the MONITORX instruction can be determined by CPUID function 0000_0005h.

The rAX register provides the effective address. The DS segment is the default segment used to create the linear address. Segment overrides may be used with the MONITORX instruction.

The ECX register specifies optional extensions for the MONITORX instruction. There are currently no extensions defined and setting any bits in ECX will result in a #GP exception. The ECX register operand is implicitly 32-bits.

The EDX register specifies optional hints for the MONITORX instruction. There are currently no hints defined and EDX is ignored by the processor. The EDX register operand is implicitly 32-bits.

The MONITORX instruction can be executed at any privilege level and MSR `C001_0015h[MonMwaitUserEn]` has no effect on MONITORX.

MONITORX performs the same segmentation and paging checks as a 1-byte read.

Support for the MONITORX instruction is indicated by CPUID Fn8000_0001_ECX[MONITORX] (bit 29) = 1.

Software must check the CPUID bit once per program or library initialization before using the MONITORX instruction, or inconsistent behavior may result.

The following pseudo-code shows typical usage of a MONITORX/MWAITX pair:

```
EAX = Linear_Address_to_Monitor;
ECX = 0; // Extensions
EDX = 0; // Hints
while (!matching_store_done){
      MONITORX EAX, ECX, EDX
IF (!matching_store_done) {
      MWAITX EAX, ECX
                }
}
```

Mnemonic	Opcode	Description
MONITORX	0F 01 FA	Establishes a range to be monitored

Related Instructions

MWAITX, MONITOR, MWAIT

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	MONITORX/MWAITX instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[MONITORX] =0
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical
	X	X	X	ECX was non-zero
			X	A null data segment was used to reference memory
Page Fault, #PF		X	X	A page fault resulted from the execution of the instruction

MOV Move

Copies an immediate value or the value in a general-purpose register, segment register, or memory location (second operand) to a general-purpose register, segment register, or memory location. The source and destination must be the same size (byte, word, doubleword, or quadword) and cannot both be memory locations.

In opcodes A0 through A3, the memory offsets (called *moffsets*) are address sized. In 64-bit mode, memory offsets default to 64 bits. Opcodes A0–A3, in 64-bit mode, are the only cases that support a 64-bit offset value. (In all other cases, offsets and displacements are a maximum of 32 bits.) The B8 through BF (B8 +*rq*) opcodes, in 64-bit mode, are the only cases that support a 64-bit immediate value (in all other cases, immediate values are a maximum of 32 bits).

When reading segment-registers with a 32-bit operand size, the processor zero-extends the 16-bit selector results to 32 bits. When reading segment-registers with a 64-bit operand size, the processor zero-extends the 16-bit selector to 64 bits. If the destination operand specifies a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector.

It is possible to move a null segment selector value (0000–0003h) into the DS, ES, FS, or GS register. This action does not cause a general protection fault, but a subsequent reference to such a segment *does* cause a #GP exception. For more information about segment selectors, see “Segment Selectors and Registers” in Volume 2.

When the MOV instruction is used to load the SS register, the processor blocks external interrupts until after the execution of the following instruction. This action allows the following instruction to be a MOV instruction to load a stack pointer into the ESP register (`MOV ESP,val`) before an interrupt occurs. However, the LSS instruction provides a more efficient method of loading SS and ESP.

Attempting to use the MOV instruction to load the CS register generates an invalid opcode exception (#UD). Use the far JMP, CALL, or RET instructions to load the CS register.

To initialize a register to 0, rather than using a MOV instruction, it may be more efficient to use the XOR instruction with identical destination and source operands.

Mnemonic	Opcode	Description
MOV *reg/mem8*, *reg8*	88 */r*	Move the contents of an 8-bit register to an 8-bit destination register or memory operand.
MOV *reg/mem16*, *reg16*	89 */r*	Move the contents of a 16-bit register to a 16-bit destination register or memory operand.
MOV *reg/mem32*, *reg32*	89 */r*	Move the contents of a 32-bit register to a 32-bit destination register or memory operand.
MOV *reg/mem64*, *reg64*	89 */r*	Move the contents of a 64-bit register to a 64-bit destination register or memory operand.
MOV *reg8*, *reg/mem8*	8A */r*	Move the contents of an 8-bit register or memory operand to an 8-bit destination register.

Mnemonic	Opcode	Description
MOV *reg16*, *reg/mem16*	8B */r*	Move the contents of a 16-bit register or memory operand to a 16-bit destination register.
MOV *reg32*, *reg/mem32*	8B */r*	Move the contents of a 32-bit register or memory operand to a 32-bit destination register.
MOV *reg64*, *reg/mem64*	8B */r*	Move the contents of a 64-bit register or memory operand to a 64-bit destination register.
MOV *reg16/32/64/mem16*, *segReg*	8C */r*	Move the contents of a segment register to a 16-bit, 32-bit, or 64-bit destination register or to a 16-bit memory operand.
MOV *segReg, reg/mem16*	8E */r*	Move the contents of a 16-bit register or memory operand to a segment register.
MOV AL, *moffset8*	A0	Move 8-bit data at a specified memory offset to the AL register.
MOV AX, *moffset16*	A1	Move 16-bit data at a specified memory offset to the AX register.
MOV EAX, *moffset32*	A1	Move 32-bit data at a specified memory offset to the EAX register.
MOV RAX, *moffset64*	A1	Move 64-bit data at a specified memory offset to the RAX register.
MOV *moffset8*, AL	A2	Move the contents of the AL register to an 8-bit memory offset.
MOV *moffset16*, AX	A3	Move the contents of the AX register to a 16-bit memory offset.
MOV *moffset32*, EAX	A3	Move the contents of the EAX register to a 32-bit memory offset.
MOV *moffset64*, RAX	A3	Move the contents of the RAX register to a 64-bit memory offset.
MOV *reg8*, *imm8*	B0 *+rb ib*	Move an 8-bit immediate value into an 8-bit register.
MOV *reg16*, *imm16*	B8 *+rw iw*	Move a 16-bit immediate value into a 16-bit register.
MOV *reg32*, *imm32*	B8 *+rd id*	Move an 32-bit immediate value into a 32-bit register.
MOV *reg64*, *imm64*	B8 *+rq iq*	Move an 64-bit immediate value into a 64-bit register.
MOV *reg/mem8, imm8*	C6 /0 *ib*	Move an 8-bit immediate value to an 8-bit register or memory operand.
MOV *reg/mem16, imm16*	C7 /0 *iw*	Move a 16-bit immediate value to a 16-bit register or memory operand.
MOV *reg/mem32, imm32*	C7 /0 *id*	Move a 32-bit immediate value to a 32-bit register or memory operand.
MOV *reg/mem64, imm32*	C7 /0 *id*	Move a 32-bit signed immediate value to a 64-bit register or memory operand.

Related Instructions

MOV CR*n*, MOV DR*n*, MOVD, MOVSX, MOVZX, MOVSXD, MOVS*x*

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	An attempt was made to load the CS register.
Segment not present, #NP (selector)			X	The DS, ES, FS, or GS register was loaded with a non-null segment selector and the segment was marked not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Stack, #SS (selector)			X	The SS register was loaded with a non-null segment selector, and the segment was marked not present.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
General protection, #GP (selector)			X	A segment register was loaded, but the segment descriptor exceeded the descriptor table limit.
			X	A segment register was loaded and the segment selector's TI bit was set, but the LDT selector was a null selector.
			X	The SS register was loaded with a null segment selector in non-64-bit mode or while CPL = 3.
			X	The SS register was loaded and the segment selector RPL and the segment descriptor DPL were not equal to the CPL.
			X	The SS register was loaded and the segment pointed to was not a writable data segment.
			X	The DS, ES, FS, or GS register was loaded and the segment pointed to was a data or non-conforming code segment, but the RPL or CPL was greater than the DPL.
			X	The DS, ES, FS, or GS register was loaded and the segment pointed to was not a data segment or readable code segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MOVBE Move Big Endian

Loads or stores a general purpose register while swapping the byte order. Operates on 16-bit, 32-bit, or 64-bit values. Converts big-endian formatted memory data to little-endian format when loading a register and reverses the conversion when storing a GPR to memory.

The load form reads a 16-, 32-, or 64-bit value from memory, swaps the byte order, and places the reordered value in a general-purpose register. When the operand size is 16 bits, the upper word of the destination register remains unchanged. In 64-bit mode, when the operand size is 32 bits, the upper doubleword of the destination register is cleared.

The store form takes a 16-, 32-, or 64-bit value from a general-purpose register, swaps the byte order, and stores the reordered value in the specified memory location. The contents of the source GPR remains unchanged.

In the 16-bit swap, the upper and lower bytes are exchanged. In the doubleword swap operation, bits 7:0 are exchanged with bits 31:24 and bits 15:8 are exchanged with bits 23:16. In the quadword swap operation, bits 7:0 are exchanged with bits 63:56, bits 15:8 with bits 55:48, bits 23:16 with bits 47:40, and bits 31:24 with bits 39:32.

Support for the MOVBE instruction is indicated by CPUID Fn0000_0001_ECX[MOVBE] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Instruction Encoding

Mnemonic	Opcode	Description
MOVBE *reg16*, *mem16*	0F 38 F0 */r*	Load the low word of a general-purpose register from a 16-bit memory location while swapping the bytes.
MOVBE *reg32*, *mem32*	0F 38 F0 */r*	Load the low doubleword of a general-purpose register from a 32-bit memory location while swapping the bytes.
MOVBE *reg64*, *mem64*	0F 38 F0 */r*	Load a 64-bit register from a 64-bit memory location while swapping the bytes.
MOVBE *mem16*, *reg16*	0F 38 F1 */r*	Store the low word of a general-purpose register to a 16-bit memory location while swapping the bytes.
MOVBE *mem32*, *reg32*	0F 38 F1 */r*	Store the low doubleword of a general-purpose register to a 32-bit memory location while swapping the bytes.
MOVBE *mem64*, *reg64*	0F 38 F1 */r*	Store the contents of a 64-bit general-purpose register to a 64-bit memory location while swapping the bytes.

Related Instruction

BSWAP

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported as indicated by CPUID Fn0000_0001_ECX[MOVBE] = 0.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MOVD Move Doubleword or Quadword

Moves a 32-bit or 64-bit value in one of the following ways:

- from a 32-bit or 64-bit general-purpose register or memory location to the low-order 32 or 64 bits of an XMM register, with zero-extension to 128 bits
- from the low-order 32 or 64 bits of an XMM to a 32-bit or 64-bit general-purpose register or memory location
- from a 32-bit or 64-bit general-purpose register or memory location to the low-order 32 bits (with zero-extension to 64 bits) or the full 64 bits of an MMX register
- from the low-order 32 or the full 64 bits of an MMX register to a 32-bit or 64-bit general-purpose register or memory location

Figure 3-1 on page 239 illustrates the operation of the MOVD instruction.

The MOVD instruction form that moves data to or from MMX registers is part of the MMX instruction subset. Support for MMX instructions is indicated by CPUID Fn0000_0001_EDX[MMX] or Fn0000_0001_EDX[MMX] = 1.

The MOVD instruction form that moves data to or from XMM registers is part of the SSE2 instruction subset. Support for SSE2 instructions is indicated by CPUID Fn0000_0001_EDX[SSE2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

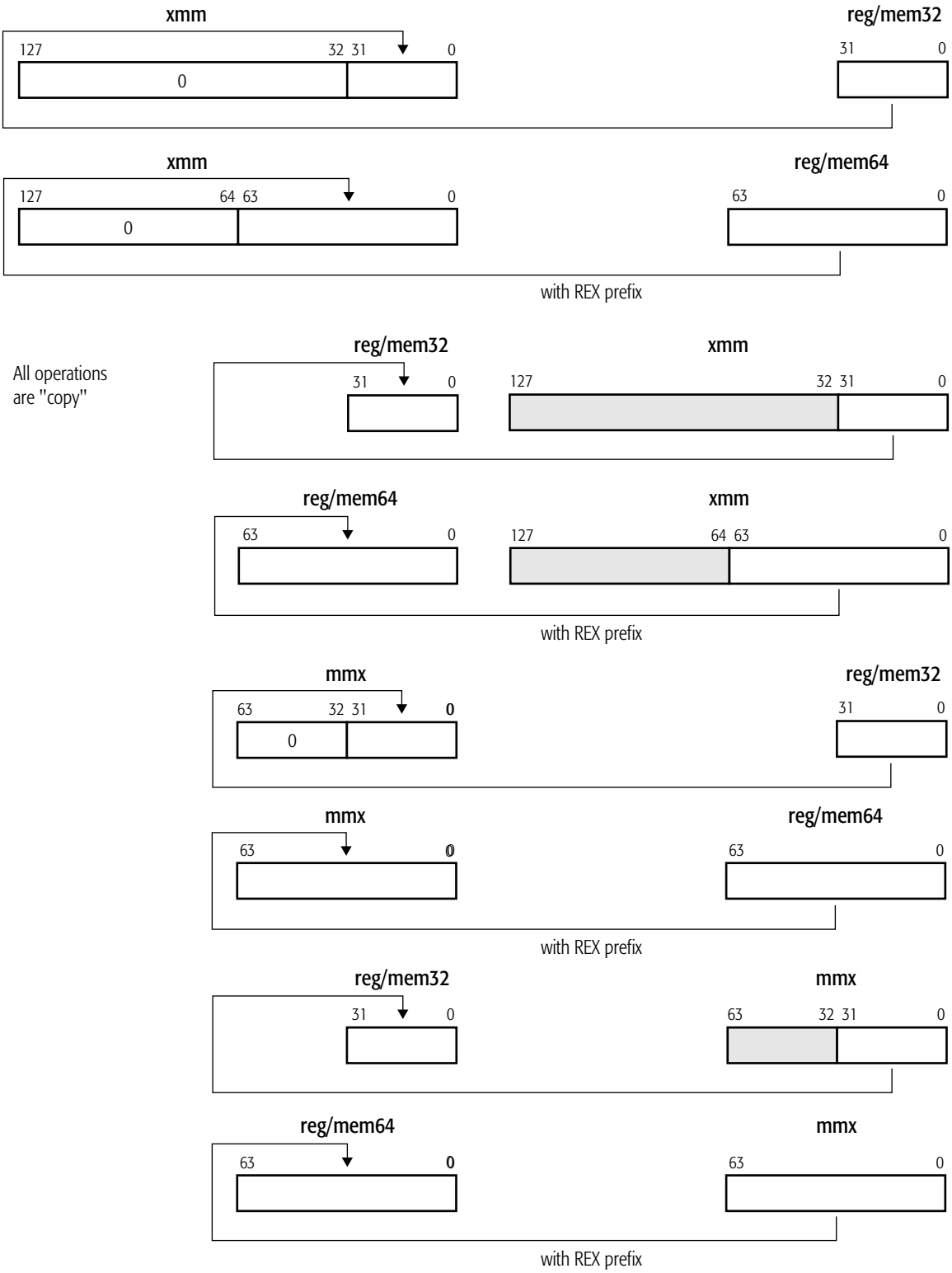


Figure 3-1. MOVD Instruction Operation

Instruction Encoding

Mnemonic	Opcode	Description
MOVD *xmm*, *reg/mem32*	66 0F 6E */r*	Move 32-bit value from a general-purpose register or 32-bit memory location to an XMM register.
MOVD[1] *xmm*, *reg/mem64*	66 0F 6E */r*	Move 64-bit value from a general-purpose register or 64-bit memory location to an XMM register.
MOVD *reg/mem32*, *xmm*	66 0F 7E */r*	Move 32-bit value from an XMM register to a 32-bit general-purpose register or memory location.
MOVD[1] *reg/mem64*, *xmm*	66 0F 7E */r*	Move 64-bit value from an XMM register to a 64-bit general-purpose register or memory location.
MOVD *mmx*, *reg/mem32*	0F 6E */r*	Move 32-bit value from a general-purpose register or 32-bit memory location to an MMX register.
MOVD *mmx*, *reg/mem64*	0F 6E */r*	Move 64-bit value from a general-purpose register or 64-bit memory location to an MMX register.
MOVD *reg/mem32*, *mmx*	0F 7E */r*	Move 32-bit value from an MMX register to a 32-bit general-purpose register or memory location.
MOVD *reg/mem64*, *mmx*	0F 7E */r*	Move 64-bit value from an MMX register to a 64-bit general-purpose register or memory location.

Note: *1. Also known as MOVQ in some developer tools.*

Related Instructions

MOVDQA, MOVDQU, MOVDQ2Q, MOVQ, MOVQ2DQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

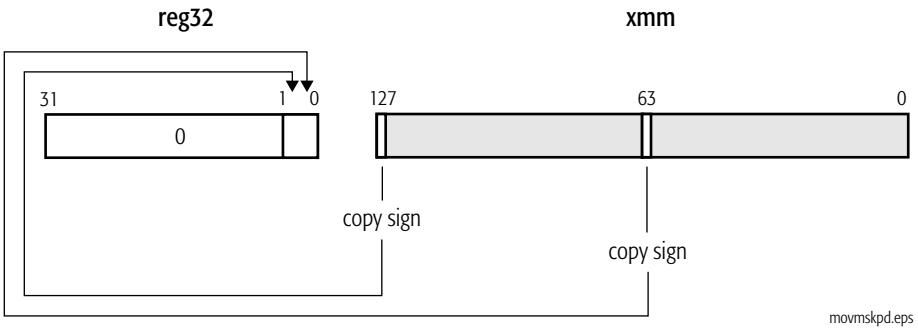
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode, #UD	X	X	X	MMX instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[MMX] or Fn0000_0001_EDX[MMX] = 0.
	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The instruction used XMM registers while CR4.OSFXSR = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.

Exception	Real	Virtual 8086	Protected	Description
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An x87 floating-point exception was pending and the instruction referenced an MMX register.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MOVMSKPD Extract Packed Double-Precision Floating-Point Sign Mask

Moves the sign bits of two packed double-precision floating-point values in an XMM register (second operand) to the two low-order bits of a general-purpose register (first operand) with zero-extension.

The function of the MOVMSKPD instruction is illustrated by the diagram below:



The MOVMSKPD instruction is an SSE2 instruction. Support for SSE2 instructions is indicated by CPUID Fn0000_0001_EDX[SSE2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Instruction Encoding

Mnemonic	Opcode	Description
MOVMSKPD *reg32*, *xmm*	66 0F 50 */r*	Move sign bits 127 and 63 in an XMM register to a 32-bit general-purpose register.

Related Instructions

MOVMSKPS, PMOVMSKB

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.

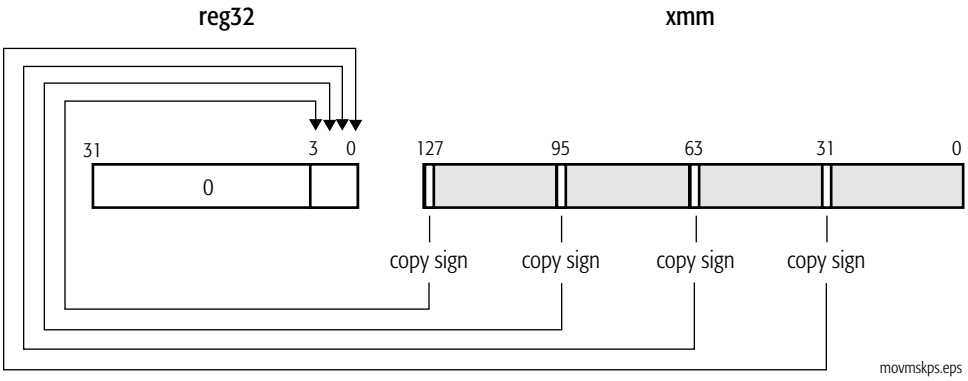
MOVMSKPS Extract Packed Single-Precision Floating-Point Sign Mask

Moves the sign bits of four packed single-precision floating-point values in an XMM register (second operand) to the four low-order bits of a general-purpose register (first operand) with zero-extension.

The MOVMSKPD instruction is an SSE2 instruction. Support for SSE2 instructions is indicated by CPUID Fn0000_0001_EDX[SSE2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Opcode	Description
MOVMSKPS *reg32*, *xmm*	0F 50 */r*	Move sign bits 127, 95, 63, 31 in an XMM register to a 32-bit general-purpose register.



Related Instructions

MOVMSKPD, PMOVMSKB

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.

MOVNTI Move Non-Temporal Doubleword or Quadword

Stores a value in a 32-bit or 64-bit general-purpose register (second operand) in a memory location (first operand). This instruction indicates to the processor that the data is non-temporal and is unlikely to be used again soon. The processor treats the store as a write-combining (WC) memory write, which minimizes cache pollution. The exact method by which cache pollution is minimized depends on the hardware implementation of the instruction. For further information, see “Memory Optimization” in Volume 1.

The MOVNTI instruction is weakly-ordered with respect to other instructions that operate on memory. Software should use an SFENCE instruction to force strong memory ordering of MOVNTI with respect to other stores.

The MOVNTI instruction is an SSE2 instruction. Support for SSE2 instructions is indicated by CPUID Fn0000_0001_EDX[SSE2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Opcode	Description
MOVNTI *mem32*, *reg32*	0F C3 */r*	Stores a 32-bit general-purpose register value into a 32-bit memory location, minimizing cache pollution.
MOVNTI *mem64*, *reg64*	0F C3 */r*	Stores a 64-bit general-purpose register value into a 64-bit memory location, minimizing cache pollution.

Related Instructions

MOVNTDQ, MOVNTPD, MOVNTPS, MOVNTQ

rFLAGS Affected

None

Exceptions

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
			X	The destination operand was in a non-writable segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MOVS
MOVSB
MOVSW
MOVSD
MOVSQ

Move String

Moves a byte, word, doubleword, or quadword from the memory location pointed to by DS:rSI to the memory location pointed to by ES:rDI, and then increments or decrements the rSI and rDI registers according to the state of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments both pointers; otherwise, it decrements them. It increments or decrements the pointers by 1, 2, 4, or 8, depending on the size of the operands.

The forms of the MOVS*x* instruction with explicit operands address the first operand at *seg*:[rSI]. The value of *seg* defaults to the DS segment, but can be overridden by a segment prefix. These instructions always address the second operand at ES:[rDI] (ES may not be overridden). The explicit operands serve only to specify the type (size) of the value being moved.

The no-operands forms of the instruction use the DS:[rSI] and ES:[rDI] registers to point to the value to be moved (they do not allow a segment prefix). The mnemonic determines the size of the operands.

Do not confuse this MOVSD instruction with the same-mnemonic MOVSD (move scalar double-precision floating-point) instruction in the 128-bit media instruction set. Assemblers can distinguish the instructions by the number and type of operands.

The MOVS*x* instructions support the REP prefixes. For details about the REP prefixes, see “Repeat Prefixes” on page 12.

Mnemonic	Opcode	Description
MOVS *mem8*, *mem8*	A4	Move byte at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.
MOVS *mem16*, *mem16*	A5	Move word at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.
MOVS *mem32*, *mem32*	A5	Move doubleword at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.
MOVS *mem64*, *mem64*	A5	Move quadword at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.
MOVSB	A4	Move byte at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.
MOVSW	A5	Move word at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.

Mnemonic	Opcode	Description
MOVSD	A5	Move doubleword at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.
MOVSQ	A5	Move quadword at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.

Related Instructions

MOV, LODSx, STOS*x*

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MOVSX Move with Sign-Extension

Copies the value in a register or memory location (second operand) into a register (first operand), extending the most significant bit of an 8-bit or 16-bit value into all higher bits in a 16-bit, 32-bit, or 64-bit register.

Mnemonic	Opcode	Description
MOVSX *reg16*, *reg/mem8*	0F BE */r*	Move the contents of an 8-bit register or memory location to a 16-bit register with sign extension.
MOVSX *reg32*, *reg/mem8*	0F BE */r*	Move the contents of an 8-bit register or memory location to a 32-bit register with sign extension.
MOVSX *reg64*, *reg/mem8*	0F BE */r*	Move the contents of an 8-bit register or memory location to a 64-bit register with sign extension.
MOVSX *reg32*, *reg/mem16*	0F BF */r*	Move the contents of an 16-bit register or memory location to a 32-bit register with sign extension.
MOVSX *reg64*, *reg/mem16*	0F BF */r*	Move the contents of an 16-bit register or memory location to a 64-bit register with sign extension.

Related Instructions

MOVSXD, MOVZX

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MOVSXD Move with Sign-Extend Doubleword

Copies the 32-bit value in a register or memory location (second operand) into a 64-bit register (first operand), extending the most significant bit of the 32-bit value into all higher bits of the 64-bit register.

This instruction requires the REX prefix 64-bit operand size bit (REX.W) to be set to 1 to sign-extend a 32-bit source operand to a 64-bit result. Without the REX operand-size prefix, the operand size will be 32 bits, the default for 64-bit mode, and the source is zero-extended into a 64-bit register. With a 16-bit operand size, only 16 bits are copied, without modifying the upper 48 bits in the destination.

This instruction is available only in 64-bit mode. In legacy or compatibility mode this opcode is interpreted as ARPL.

Mnemonic	Opcode	Description
MOVSXD *reg64*, *reg/mem32*	63 */r*	Move the contents of a 32-bit register or memory operand to a 64-bit register with sign extension.

Related Instructions

MOVSX, MOVZX

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS			X	A memory address was non-canonical.
General protection, #GP			X	A memory address was non-canonical.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

MOVZX Move with Zero-Extension

Copies the value in a register or memory location (second operand) into a register (first operand), zero-extending the value to fit in the destination register. The operand-size attribute determines the size of the zero-extended value.

Mnemonic	Opcode	Description
MOVZX *reg16*, *reg/mem8*	0F B6 */r*	Move the contents of an 8-bit register or memory operand to a 16-bit register with zero-extension.
MOVZX *reg32*, *reg/mem8*	0F B6 */r*	Move the contents of an 8-bit register or memory operand to a 32-bit register with zero-extension.
MOVZX *reg64*, *reg/mem8*	0F B6 */r*	Move the contents of an 8-bit register or memory operand to a 64-bit register with zero-extension.
MOVZX *reg32*, *reg/mem16*	0F B7 */r*	Move the contents of a 16-bit register or memory operand to a 32-bit register with zero-extension.
MOVZX *reg64*, *reg/mem16*	0F B7 */r*	Move the contents of a 16-bit register or memory operand to a 64-bit register with zero-extension.

Related Instructions

MOVSXD, MOVSX

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MUL Unsigned Multiply

Multiplies the unsigned byte, word, doubleword, or quadword value in the specified register or memory location by the value in AL, AX, EAX, or RAX and stores the result in AX, DX:AX, EDX:EAX, or RDX:RAX (depending on the operand size). It puts the high-order bits of the product in AH, DX, EDX, or RDX.

If the upper half of the product is non-zero, the instruction sets the carry flag (CF) and overflow flag (OF) both to 1. Otherwise, it clears CF and OF to 0. The other arithmetic flags (SF, ZF, AF, PF) are undefined.

Mnemonic	Opcode	Description
MUL *reg/mem8*	F6 /4	Multiplies an 8-bit register or memory operand by the contents of the AL register and stores the result in the AX register.
MUL *reg/mem16*	F7 /4	Multiplies a 16-bit register or memory operand by the contents of the AX register and stores the result in the DX:AX register.
MUL *reg/mem32*	F7 /4	Multiplies a 32-bit register or memory operand by the contents of the EAX register and stores the result in the EDX:EAX register.
MUL *reg/mem64*	F7 /4	Multiplies a 64-bit register or memory operand by the contents of the RAX register and stores the result in the RDX:RAX register.

Related Instructions

DIV

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				U	U	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference is performed while alignment checking was enabled.

MULX Multiply Unsigned

Computes the unsigned product of the specified source operand and the implicit source operand rDX. Writes the upper half of the product to the first destination and the lower half to the second. Does not affect the arithmetic flags.

This instruction has three operands:

MULX *dest1*, *dest2*, *src*

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The first and second operands (*dest1* and *dest2*) are general purpose registers. The specified source operand (*src*) is either a general purpose register or a memory operand. If the first and second operands specify the same register, the register receives the upper half of the product.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
MULX *reg32, reg32*, *reg/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{dest2}}$.0.11	F6 /r
MULX *reg64, reg64, reg/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{dest2}}$.0.11	F6 /r

Related Instructions

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

MWAITX Monitor Wait with Timeout

Used in conjunction with the MONITORX instruction to cause a processor to wait until a store occurs to a specific linear address range from another processor or the timer expires. The previously executed MONITORX instruction causes the processor to enter the monitor event pending state. The MWAITX instruction may enter an implementation dependent power state until the monitor event pending state is exited. The MWAITX instruction has the same effect on architectural state as the NOP instruction.

Events that cause an exit from the monitor event pending state include:

- A store from another processor matches the address range established by the MONITORX instruction.
- The timer expires.
- Any unmasked interrupt, including INTR, NMI, SMI, INIT.
- RESET.
- Any far control transfer that occurs between the MONITORX and the MWAITX.

EAX specifies optional hints for the MWAITX instruction. Optimized C-state request is communicated through EAX[7:4]. The processor C-state is EAX[7:4]+1, so to request C0 is to place the value F in EAX[7:4] and to request C1 is to place the value 0 in EAX[7:4]. All other components of EAX should be zero when making the C1 request. Setting a reserved bit in EAX is ignored by the processor. This is implicitly a 32-bit operand.

ECX specifies optional extensions for the MWAITX instruction. The extensions currently defined for ECX are:

- Bit 0: When set, allows interrupts to wake MWAITX, even when eFLAGS.IF = 0. Support for this extension is indicated by a feature flag returned by the CPUID instruction.
- Bit 1: When set, EBX contains the maximum wait time expressed in Software P0 clocks, the same clocks counted by the TSC. Setting bit 1 but passing in a value of zero on EBX is equivalent to setting bit 1 to a zero. The timer will not be an exit condition.
- Bit 31-2: When non-zero, results in a #GP(0) exception.

This is implicitly a 32-bit operand.

CPUID Function 0000_0005h indicates support for extended features of MONITORX/MWAITX as well as MONITOR/MWAIT:

- CPUID Fn0000_0005_ECX[EMX] = 1 indicates support for enumeration of MONITOR/MWAIT/MONITORX/MWAITX extensions.
- CPUID Fn0000_0005_ECX[IBE] = 1 indicates that MWAIT/MWAITX can set ECX[0] to allow interrupts to cause an exit from the monitor event pending state even when eFLAGS.IF = 0.

The MWAITX instruction can be executed at any privilege level and MSR C001_0015h[MonMwaitUserEn] has no effect on MWAITX.

Support for the MWAITX instruction is indicated by CPUID Fn8000_0001_ECX[MONITORX] (bit 29)= 1.

Software must check the CPUID bit once per program or library initialization before using the MWAITX instruction, or inconsistent behavior may result.

The use of the MWAITX instruction is contingent upon the satisfaction of the following coding requirements:

- MONITORX must precede the MWAITX and occur in the same loop.
- MWAITX must be conditionally executed only if the awaited store has not already occurred. (This prevents a race condition between the MONITORX instruction arming the monitoring hardware and the store intended to trigger the monitoring hardware.)

There is no indication after exiting MWAITX of why the processor exited or if the timer expired. It is up to software to check whether the awaiting store has occurred, and if not, determining how much time has elapsed if it wants to re-establish the MONITORX with a new timer value.

Mnemonic	Opcode	Description
MWAITX	0F 01 FB	Causes the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events

Related Instructions

MONITORX, MONITOR, MWAIT

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	MONITORX/MWAITX instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[MONITORX] =0
General protection, #GP	X	X	X	Unsupported extension bits in ECX

NEG Two's Complement Negation

Performs the two's complement negation of the value in the specified register or memory location by subtracting the value from 0. Use this instruction only on signed integer numbers.

If the value is 0, the instruction clears the CF flag to 0; otherwise, it sets CF to 1. The OF, SF, ZF, AF, and PF flag settings depend on the result of the operation.

The forms of the NEG instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see "Lock Prefix" on page 11.

Mnemonic	Opcode	Description
NEG *reg/mem8*	F6 /3	Performs a two's complement negation on an 8-bit register or memory operand.
NEG *reg/mem16*	F7 /3	Performs a two's complement negation on a 16-bit register or memory operand.
NEG *reg/mem32*	F7 /3	Performs a two's complement negation on a 32-bit register or memory operand.
NEG *reg/mem64*	F7 /3	Performs a two's complement negation on a 64-bit register or memory operand.

Related Instructions

AND, NOT, OR, XOR

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand is in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

NOP No Operation

Does nothing. This instruction increments the rIP to point to next instruction, but does not affect the machine state in any other way.

The single-byte variant is an alias for `XCHG rAX,rAX`.

Mnemonic	Opcode	Description
NOP	90	Performs no operation.
NOP *reg/mem16*	0F 1F /0	Performs no operation on a 16-bit register or memory operand.
NOP *reg/mem32*	0F 1F /0	Performs no operation on a 32-bit register or memory operand.
NOP *reg/mem64*	0F 1F /0	Performs no operation on a 64-bit register or memory operand.

Related Instructions

None

rFLAGS Affected

None

Exceptions

None

NOT One's Complement Negation

Performs the one's complement negation of the value in the specified register or memory location by inverting each bit of the value.

The memory-operand forms of the NOT instruction support the LOCK prefix. For details about the LOCK prefix, see "Lock Prefix" on page 11.

Mnemonic	Opcode	Description
NOT *reg/mem8*	F6 /2	Complements the bits in an 8-bit register or memory operand.
NOT *reg/mem16*	F7 /2	Complements the bits in a 16-bit register or memory operand.
NOT *reg/mem32*	F7 /2	Complements the bits in a 32-bit register or memory operand.
NOT *reg/mem64*	F7 /2	Compliments the bits in a 64-bit register or memory operand.

Related Instructions

AND, NEG, OR, XOR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference is performed while alignment checking was enabled.

OR — Logical OR

Performs a logical or on the bits in a register, memory location, or immediate value (second operand) and a register or memory location (first operand) and stores the result in the first operand location. The two operands cannot both be memory locations.

If both corresponding bits are 0, the corresponding bit of the result is 0; otherwise, the corresponding result bit is 1.

The forms of the OR instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
OR AL, *imm8*	0C *ib*	or the contents of AL with an immediate 8-bit value.
OR AX, *imm16*	0D *iw*	or the contents of AX with an immediate 16-bit value.
OR EAX, *imm32*	0D *id*	or the contents of EAX with an immediate 32-bit value.
OR RAX, *imm32*	0D *id*	or the contents of RAX with a sign-extended immediate 32-bit value.
OR *reg/mem8*, *imm8*	80 /1 *ib*	or the contents of an 8-bit register or memory operand and an immediate 8-bit value.
OR *reg/mem16*, *imm16*	81 /1 *iw*	or the contents of a 16-bit register or memory operand and an immediate 16-bit value.
OR *reg/mem32*, *imm32*	81 /1 *id*	or the contents of a 32-bit register or memory operand and an immediate 32-bit value.
OR *reg/mem64*, *imm32*	81 /1 *id*	or the contents of a 64-bit register or memory operand and sign-extended immediate 32-bit value.
OR *reg/mem16*, *imm8*	83 /1 *ib*	or the contents of a 16-bit register or memory operand and a sign-extended immediate 8-bit value.
OR *reg/mem32*, *imm8*	83 /1 *ib*	or the contents of a 32-bit register or memory operand and a sign-extended immediate 8-bit value.
OR *reg/mem64*, *imm8*	83 /1 *ib*	or the contents of a 64-bit register or memory operand and a sign-extended immediate 8-bit value.
OR *reg/mem8*, reg8	08 */r*	or the contents of an 8-bit register or memory operand with the contents of an 8-bit register.
OR *reg/mem16*, *reg16*	09 */r*	or the contents of a 16-bit register or memory operand with the contents of a 16-bit register.
OR *reg/mem32*, *reg32*	09 */r*	or the contents of a 32-bit register or memory operand with the contents of a 32-bit register.
OR *reg/mem64*, *reg64*	09 */r*	or the contents of a 64-bit register or memory operand with the contents of a 64-bit register.

Mnemonic	Opcode	Description
OR *reg8*, *reg/mem8*	0A */r*	or the contents of an 8-bit register with the contents of an 8-bit register or memory operand.
OR *reg16*, *reg/mem16*	0B */r*	or the contents of a 16-bit register with the contents of a 16-bit register or memory operand.
OR *reg32*, *reg/mem32*	0B */r*	or the contents of a 32-bit register with the contents of a 32-bit register or memory operand.
OR *reg64*, *reg/mem64*	0B */r*	or the contents of a 64-bit register with the contents of a 64-bit register or memory operand.

The following chart summarizes the effect of this instruction:

X	Y	X or Y
0	0	0
0	1	1
1	0	1
1	1	1

Related Instructions

AND, NEG, NOT, XOR

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	M	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

OUT Output to Port

Copies the value from the AL, AX, or EAX register (second operand) to an I/O port (first operand). The port address can be a byte-immediate value (00h to FFh) or the value in the DX register (0000h to FFFFh). The source register used determines the size of the port (8, 16, or 32 bits).

If the operand size is 64 bits, OUT only writes to a 32-bit I/O port.

If the CPL is higher than the IOPL or the mode is virtual mode, OUT checks the I/O permission bitmap in the TSS before allowing access to the I/O port. See Volume 2 for details on the TSS I/O permission bitmap.

Mnemonic	Opcode	Description
OUT *imm8*, AL	E6 *ib*	Output the byte in the AL register to the port specified by an 8-bit immediate value.
OUT *imm8*, AX	E7 *ib*	Output the word in the AX register to the port specified by an 8-bit immediate value.
OUT *imm8*, EAX	E7 *ib*	Output the doubleword in the EAX register to the port specified by an 8-bit immediate value.
OUT DX, AL	EE	Output byte in AL to the output port specified in DX.
OUT DX, AX	EF	Output word in AX to the output port specified in DX.
OUT DX, EAX	EF	Output doubleword in EAX to the output port specified in DX.

Related Instructions

IN, INS*x*, OUTS*x*

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X		One or more I/O permission bits were set in the TSS for the accessed port.
			X	The CPL was greater than the IOPL and one or more I/O permission bits were set in the TSS for the accessed port.
Page fault (#PF)		X	X	A page fault resulted from the execution of the instruction.

OUTS
OUTSB
OUTSW
OUTSD

Output String

Copies data from the memory location pointed to by DS:rSI to the I/O port address (0000h to FFFFh) specified in the DX register, and then increments or decrements the rSI register according to the setting of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments rSI; otherwise, it decrements rSI. It increments or decrements the pointer by 1, 2, or 4, depending on the size of the value being copied.

The OUTS DX mnemonic uses an explicit memory operand (second operand) to determine the type (size) of the value being copied, but always uses DS:rSI for the location of the value to copy. The explicit register operand (first operand) specifies the I/O port address and must always be DX.

The no-operands forms of the mnemonic use the DS:rSI register pair to point to the memory data to be copied and the contents of the DX register as the destination I/O port address. The mnemonic specifies the size of the I/O port and the type (size) of the value being copied.

The OUTS*x* instruction supports the REP prefix. For details about the REP prefix, see “Repeat Prefixes” on page 12.

If the effective operand size is 64-bits, the instruction behaves as if the operand size were 32 bits.

If the CPL is higher than the IOPL or the mode is virtual mode, OUTS*x* checks the I/O permission bitmap in the TSS before allowing access to the I/O port. See Volume 2 for details on the TSS I/O permission bitmap.

Mnemonic	Opcode	Description
OUTS DX, *mem8*	6E	Output the byte in DS:rSI to the port specified in DX, then increment or decrement rSI.
OUTS DX, *mem16*	6F	Output the word in DS:rSI to the port specified in DX, then increment or decrement rSI.
OUTS DX, *mem32*	6F	Output the doubleword in DS:rSI to the port specified in DX, then increment or decrement rSI.
OUTSB	6E	Output the byte in DS:rSI to the port specified in DX, then increment or decrement rSI.
OUTSW	6F	Output the word in DS:rSI to the port specified in DX, then increment or decrement rSI.
OUTSD	6F	Output the doubleword in DS:rSI to the port specified in DX, then increment or decrement rSI.

Related Instructions

IN, INS*x*, OUT

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
		X		One or more I/O permission bits were set in the TSS for the accessed port.
			X	The CPL was greater than the IOPL and one or more I/O permission bits were set in the TSS for the accessed port.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference is performed while alignment checking was enabled.

PAUSE Pause

Improves the performance of spin loops, by providing a hint to the processor that the current code is in a spin loop. The processor may use this to optimize power consumption while in the spin loop.

Architecturally, this instruction behaves like a NOP instruction.

Processors that do not support PAUSE treat this opcode as a NOP instruction.

Mnemonic	Opcode	Description
PAUSE	F3 90	Provides a hint to processor that a spin loop is being executed.

Related Instructions

None

rFLAGS Affected

None

Exceptions

None

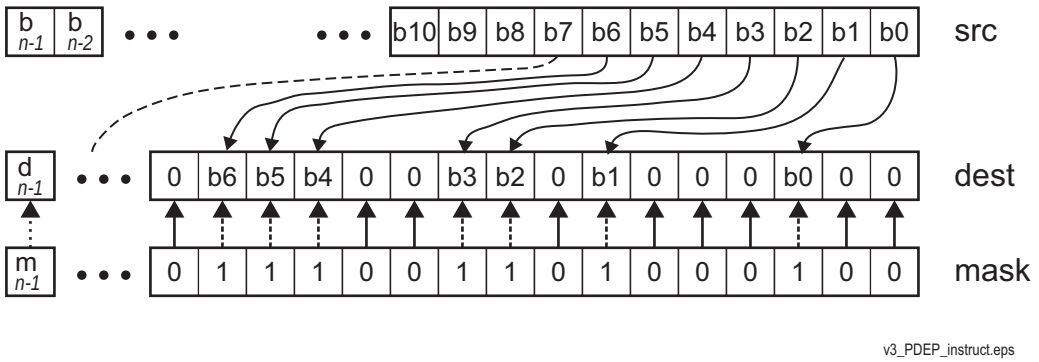
PDEP Parallel Deposit Bits

Scatters consecutive bits of the first source operand, starting at the least significant bit, to bit positions in the destination as specified by 1 bits in the second source operand (*mask*). Bit positions in the destination corresponding to 0 bits in the mask are cleared.

This instruction has three operands:

PDEP *dest*, *src, mask*

The following diagram illustrates the operation of this instruction.



If the mask is all ones, the execution of this instruction effectively copies the source to the destination.

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) and the source (*src*) are general-purpose registers. The second source operand (*mask*) is either a general-purpose register or a memory operand.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
PDEP *reg32*, *reg32, reg/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src}$.0.11	F5 /r
PDEP *reg64, reg64, reg/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src}$.0.11	F5 /r

Related Instructions

rFLAGS Affected

None.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

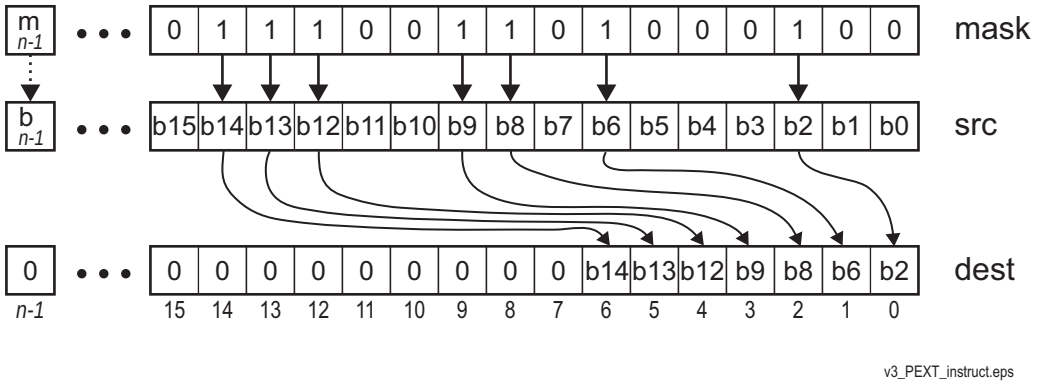
PEXT Parallel Extract Bits

Copies bits from the source operand, based on a mask, and packs them into the low-order bits of the destination. Clears all bits in the destination to the left of the most-significant bit copied.

This instruction has three operands:

PEXT *dest*, *src, mask*

The following diagram illustrates the operation of this instruction.



If the mask is all ones, the execution of this instruction effectively copies the source to the destination.

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) and the source (*src*) are general-purpose registers. The second source operand (*mask*) is either a general-purpose register or a memory operand.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
PEXT *reg32*, *reg32, reg/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src}$.0.10	F5 /r
PEXT *reg64, reg64, reg/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src}$.0.10	F5 /r

Related Instructions

rFLAGS Affected

None.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

POP Pop Stack

Copies the value pointed to by the stack pointer (SS:rSP) to the specified register or memory location and then increments the rSP by 2 for a 16-bit pop, 4 for a 32-bit pop, or 8 for a 64-bit pop.

The operand-size attribute determines the amount by which the stack pointer is incremented (2, 4 or 8 bytes). The stack-size attribute determines whether SP, ESP, or RSP is incremented.

For forms of the instruction that load a segment register (POP DS, POP ES, POP FS, POP GS, POP SS), the source operand must be a valid segment selector. When a segment selector is popped into a segment register, the processor also loads all associated descriptor information into the hidden part of the register and validates it.

It is possible to pop a null segment selector value (0000–0003h) into the DS, ES, FS, or GS register. This action does not cause a general protection fault, but a subsequent reference to such a segment *does* cause a #GP exception. For more information about segment selectors, see "Segment Selectors and Registers" in *Volume 2: System Programming*.

In 64-bit mode, the POP operand size defaults to 64 bits and there is no prefix available to encode a 32-bit operand size. Using POP DS, POP ES, or POP SS instruction in 64-bit mode generates an invalid-opcode exception.

This instruction cannot pop a value into the CS register. The RET (Far) instruction performs this function.

Mnemonic	Opcode	Description
POP reg/*mem16*	8F /0	Pop the top of the stack into a 16-bit register or memory location.
POP reg/*mem32*	8F /0	Pop the top of the stack into a 32-bit register or memory location. (No prefix for encoding this in 64-bit mode.)
POP reg/*mem64*	8F /0	Pop the top of the stack into a 64-bit register or memory location.
POP *reg16*	58 +*rw*	Pop the top of the stack into a 16-bit register.
POP *reg32*	58 +*rd*	Pop the top of the stack into a 32-bit register. (No prefix for encoding this in 64-bit mode.)
POP *reg64*	58 +*rq*	Pop the top of the stack into a 64-bit register.
POP DS	1F	Pop the top of the stack into the DS register. (Invalid in 64-bit mode.)
POP ES	07	Pop the top of the stack into the ES register. (Invalid in 64-bit mode.)
POP SS	17	Pop the top of the stack into the SS register. (Invalid in 64-bit mode.)

Mnemonic	Opcode	Description
POP FS	0F A1	Pop the top of the stack into the FS register.
POP GS	0F A9	Pop the top of the stack into the GS register.

Related Instructions

PUSH

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	POP DS, POP ES, or POP SS was executed in 64-bit mode.
Segment not present, #NP (selector)			X	The DS, ES, FS, or GS register was loaded with a non-null segment selector and the segment was marked not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Stack, #SS (selector)			X	The SS register was loaded with a non-null segment selector and the segment was marked not present.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
General protection, #GP (selector)			X	A segment register was loaded and the segment descriptor exceeded the descriptor table limit.
			X	A segment register was loaded and the segment selector's TI bit was set, but the LDT selector was a null selector.
			X	The SS register was loaded with a null segment selector in non-64-bit mode or while CPL = 3.
			X	The SS register was loaded and the segment selector RPL and the segment descriptor DPL were not equal to the CPL.
			X	The SS register was loaded and the segment pointed to was not a writable data segment.
			X	The DS, ES, FS, or GS register was loaded and the segment pointed to was a data or non-conforming code segment, but the RPL or the CPL was greater than the DPL.
			X	The DS, ES, FS, or GS register was loaded and the segment pointed to was not a data segment or readable code segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

POPA POPAD — POP All GPRs

Pops words or doublewords from the stack into the general-purpose registers in the following order: eDI, eSI, eBP, eSP (image is popped and discarded), eBX, eDX, eCX, and eAX. The instruction increments the stack pointer by 16 or 32, depending on the operand size.

Using the POPA or POPAD instructions in 64-bit mode generates an invalid-opcode exception.

Mnemonic	Opcode	Description
POPA	61	Pop the DI, SI, BP, SP, BX, DX, CX, and AX registers. (Invalid in 64-bit mode.)
POPAD	61	Pop the EDI, ESI, EBP, ESP, EBX, EDX, ECX, and EAX registers. (Invalid in 64-bit mode.)

Related Instructions

PUSHA, PUSHAD

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode (#UD)			X	This instruction was executed in 64-bit mode.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

POPCNT Bit Population Count

Counts the number of bits having a value of 1 in the source operand and places the result in the destination register. The source operand is a 16-, 32-, or 64-bit general purpose register or memory operand; the destination operand is a general purpose register of the same size as the source operand register.

If the input operand is zero, the ZF flag is set to 1 and zero is written to the destination register. Otherwise, the ZF flag is cleared. The other flags are cleared.

Support for the POPCNT instruction is indicated by CPUID Fn0000_0001_ECX[POPCNT] = 1. Software MUST check the CPUID bit once per program or library initialization before using the POPCNT instruction, or inconsistent behavior may result.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic		Opcode	Description
POPCNT	*reg16*, r*eg/mem16*	F3 0F B8 /r	Count the 1s in reg/mem16.
POPCNT	*reg32*, *reg/mem32*	F3 0F B8 /r	Count the 1s in reg/mem32.
POPCNT	*reg64*, *reg/mem64*	F3 0F B8 /r	Count the 1s in reg/mem64.

Related Instructions

BSF, BSR, LZCNT

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	M	0	0	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The POPCNT instruction is not supported, as indicated by CPUID Fn0000_0001_ECX[POPCNT].
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

POPF
POPFD
POPFQ

POP to rFLAGS

Pops a word, doubleword, or quadword from the stack into the rFLAGS register and then increments the stack pointer by 2, 4, or 8, depending on the operand size.

In protected or real mode, all the non-reserved flags in the rFLAGS register can be modified, except the VIP, VIF, and VM flags, which are unchanged. In protected mode, at a privilege level greater than 0 the IOPL is also unchanged. The instruction alters the interrupt flag (IF) only when the CPL is less than or equal to the IOPL.

In virtual-8086 mode, if IOPL field is less than 3, attempting to execute a POPF*x* or PUSHF*x* instruction while VME is not enabled, or the operand size is not 16-bit, generates a #GP exception.

In 64-bit mode, this instruction defaults to a 64-bit operand size; there is no prefix available to encode a 32-bit operand size.

Mnemonic	Opcode	Description
POPF	9D	Pop a word from the stack into the FLAGS register.
POPFD	9D	Pop a double word from the stack into the EFLAGS register. (No prefix for encoding this in 64-bit mode.)
POPFQ	9D	Pop a quadword from the stack to the RFLAGS register.

Action

```
// See “Pseudocode Definition” on page 57.

POPF_START:

IF (REAL_MODE)
    POPF_REAL
ELSIF (PROTECTED_MODE)
    POPF_PROTECTED
ELSE // (VIRTUAL_MODE)
    POPF_VIRTUAL


POPF_REAL:

    POP.v temp_RFLAGS
    RFLAGS.v = temp_RFLAGS          // VIF,VIP,VM unchanged
                                    // RF cleared
    EXIT
```

```
   POPF_PROTECTED:

       POP.v temp_RFLAGS
       RFLAGS.v = temp_RFLAGS          // VIF,VIP,VM unchanged
                                       // IOPL changed only if (CPL==0)
                                      // IF changed only if (CPL<=old_RFLAGS.IOPL)
                                       // RF cleared
       EXIT


   POPF_VIRTUAL:

       IF (RFLAGS.IOPL==3)
       {
           POP.v temp_RFLAGS
           RFLAGS.v = temp_RFLAGS      // VIF,VIP,VM,IOPL unchanged
                                       // RF cleared
           EXIT
       }
       ELSIF ((CR4.VME==1) && (OPERAND_SIZE==16))
       {
           POP.w temp_RFLAGS
           IF (((temp_RFLAGS.IF==1) && (RFLAGS.VIP==1)) || (temp_RFLAGS.TF==1))
               EXCEPTION [#GP(0)]
                                           // notify the virtual-mode-manager to
deliver
                                       // the task's pending interrupts
           RFLAGS.w = temp_RFLAGS      // IF,IOPL unchanged
                                       // RFLAGS.VIF=temp_RFLAGS.IF
                                       // RF cleared
           EXIT
       }
       ELSE // ((RFLAGS.IOPL<3) && ((CR4.VME==0) || (OPERAND_SIZE!=16)))
           EXCEPTION [#GP(0)]
```

Related Instructions

PUSHF, PUSHFD, PUSHFQ

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
M		M	M		0	M	M	M	M	M	M	M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP		X		The I/O privilege level was less than 3 and one of the following conditions was true: • CR4.VME was 0. • The effective operand size was 32-bit. • Both the original EFLAGS.VIP and the new EFLAGS.IF bits were set. • The new EFLAGS.TF bit was set.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PREFETCH
PREFETCHW — Prefetch L1 Data-Cache Line

Loads the entire 64-byte aligned memory sequence *containing* the specified memory address into the L1 data cache. The position of the specified memory address within the 64-byte cache line is irrelevant. If a cache hit occurs, or if a memory fault is detected, no bus cycle is initiated and the instruction is treated as a NOP.

The PREFETCHW instruction loads the prefetched line and sets the cache-line state to Modified, in anticipation of subsequent data writes to the line. The PREFETCH instruction, by contrast, typically sets the cache-line state to Exclusive (depending on the hardware implementation).

The opcodes for the PREFETCH/PREFETCHW instructions include the ModRM byte; however, only the memory form of ModRM is valid. The register form of ModRM causes an invalid-opcode exception. Because there is no destination register, the three destination register field bits of the ModRM byte define the type of prefetch to be performed. The bit patterns 000b and 001b define the PREFETCH and PREFETCHW instructions, respectively. All other bit patterns are reserved for future use.

The *reserved* PREFETCH types do not result in an invalid-opcode exception if executed. Instead, for forward compatibility with future processors that may implement additional forms of the PREFETCH instruction, all reserved PREFETCH types are implemented as synonyms of the basic PREFETCH type (the PREFETCH instruction with type 000b).

The operation of these instructions is implementation-dependent. The processor implementation can ignore or change these instructions. The size of the cache line also depends on the implementation, with a minimum size of 32 bytes. For details on the use of this instruction, see the processor data sheets or other software-optimization documentation relating to particular hardware implementations.

When paging is enabled and PREFETCHW performs a prefetch from a writable page, it may set the PTE Dirty bit to 1.

Support for the PREFETCH and PREFETCHW instructions is indicated by CPUID Fn8000_0001_ECX[3DNowPrefetch] OR Fn8000_0001_EDX[LM] OR Fn8000_0001_EDX[3DNow] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Opcode	Description
PREFETCH *mem8*	0F 0D /0	Prefetch processor cache line into L1 data cache.
PREFETCHW *mem8*	0F 0D /1	Prefetch processor cache line into L1 data cache and mark it modified.

Related Instructions

PREFETCH*level*

rFLAGS Affected

None

Exceptions

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	PREFETCH and PREFETCHW instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[3DNowPrefetch] AND Fn8000_0001_EDX[LM] AND Fn8000_0001_EDX[3DNow] = 0.
	X	X	X	The operand was a register.

PREFETCH*level* Prefetch Data to Cache Level *level*

Loads a cache line from the specified memory address into the data-cache level specified by the locality reference bits 5:3 of the ModRM byte. Table 3-3 on page 285 lists the locality reference options for the instruction.

This instruction loads a cache line even if the *mem8* address is not aligned with the start of the line. If the cache line is already contained in a cache level that is lower than the specified locality reference, or if a memory fault is detected, a bus cycle is not initiated and the instruction is treated as a NOP.

The operation of this instruction is implementation-dependent. The processor implementation can ignore or change this instruction. The size of the cache line also depends on the implementation, with a minimum size of 32 bytes. AMD processors alias PREFETCH1 and PREFETCH2 to PREFETCH0. For details on the use of this instruction, see the software-optimization documentation relating to particular hardware implementations.

Mnemonic	Opcode	Description
PREFETCHNTA *mem8*	0F 18 /0	Move data closer to the processor using the NTA reference.
PREFETCHT0 *mem8*	0F 18 /1	Move data closer to the processor using the T0 reference.
PREFETCHT1 *mem8*	0F 18 /2	Move data closer to the processor using the T1 reference.
PREFETCHT2 *mem8*	0F 18 /3	Move data closer to the processor using the T2 reference.

Table 3-3. Locality References for the Prefetch Instructions

Locality Reference	Description
NTA	Non-Temporal Access—Move the specified data into the processor with minimum cache pollution. This is intended for data that will be used only once, rather than repeatedly. The specific technique for minimizing cache pollution is implementation-dependent and may include such techniques as allocating space in a software-invisible buffer, allocating a cache line in only a single way, etc. For details, see the software-optimization documentation for a particular hardware implementation.
T0	All Cache Levels—Move the specified data into all cache levels.
T1	Level 2 and Higher—Move the specified data into all cache levels except 0th level (L1) cache.
T2	Level 3 and Higher—Move the specified data into all cache levels except 0th level (L1) and 1st level (L2) caches.

Related Instructions

PREFETCH, PREFETCHW

rFLAGS Affected

None

Exceptions

None

PUSH Push onto Stack

Decrements the stack pointer and then copies the specified immediate value or the value in the specified register or memory location to the top of the stack (the memory location pointed to by SS:rSP).

The operand-size attribute determines the number of bytes pushed to the stack. The stack-size attribute determines whether SP, ESP, or RSP is the stack pointer. The address-size attribute is used only to locate the memory operand when pushing a memory operand to the stack.

If the instruction pushes the stack pointer (rSP), the resulting value on the stack is that of rSP before execution of the instruction.

There is a PUSH CS instruction but no corresponding POP CS. The RET (Far) instruction pops a value from the top of stack into the CS register as part of its operation.

In 64-bit mode, the operand size of all PUSH instructions defaults to 64 bits, and there is no prefix available to encode a 32-bit operand size. Using the PUSH CS, PUSH DS, PUSH ES, or PUSH SS instructions in 64-bit mode generates an invalid-opcode exception.

Pushing an odd number of 16-bit operands when the stack address-size attribute is 32 results in a misaligned stack pointer.

Mnemonic	Opcode	Description
PUSH reg/*mem16*	FF /6	Push the contents of a 16-bit register or memory operand onto the stack.
PUSH reg/*mem32*	FF /6	Push the contents of a 32-bit register or memory operand onto the stack. (No prefix for encoding this in 64-bit mode.)
PUSH reg/*mem64*	FF /6	Push the contents of a 64-bit register or memory operand onto the stack.
PUSH *reg16*	50 +*rw*	Push the contents of a 16-bit register onto the stack.
PUSH *reg32*	50 +*rd*	Push the contents of a 32-bit register onto the stack. (No prefix for encoding this in 64-bit mode.)
PUSH *reg64*	50 +*rq*	Push the contents of a 64-bit register onto the stack.
PUSH *imm8*	6A *ib*	Push an 8-bit immediate value (sign-extended to 16, 32, or 64 bits) onto the stack.
PUSH *imm16*	68 *iw*	Push a 16-bit immediate value onto the stack.
PUSH *imm32*	68 *id*	Push a 32-bit immediate value onto the stack. (No prefix for encoding this in 64-bit mode.)
PUSH *imm64*	68 *id*	Push a sign-extended 32-bit immediate value onto the stack.
PUSH CS	0E	Push the CS selector onto the stack. (Invalid in 64-bit mode.)

Mnemonic	Opcode	Description
PUSH SS	16	Push the SS selector onto the stack. (Invalid in 64-bit mode.)
PUSH DS	1E	Push the DS selector onto the stack. (Invalid in 64-bit mode.)
PUSH ES	06	Push the ES selector onto the stack. (Invalid in 64-bit mode.)
PUSH FS	0F A0	Push the FS selector onto the stack.
PUSH GS	0F A8	Push the GS selector onto the stack.

Related Instructions

POP

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	PUSH CS, PUSH DS, PUSH ES, or PUSH SS was executed in 64-bit mode.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PUSHA
PUSHAD — Push All GPRs onto Stack

Pushes the contents of the eAX, eCX, eDX, eBX, eSP (original value), eBP, eSI, and eDI general-purpose registers onto the stack in that order. This instruction decrements the stack pointer by 16 or 32 depending on operand size.

Using the PUSHA or PUSHAD instruction in 64-bit mode generates an invalid-opcode exception.

Mnemonic	Opcode	Description
PUSHA	60	Push the contents of the AX, CX, DX, BX, original SP, BP, SI, and DI registers onto the stack. (Invalid in 64-bit mode.)
PUSHAD	60	Push the contents of the EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI registers onto the stack. (Invalid in 64-bit mode.)

Related Instructions

POPA, POPAD

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	This instruction was executed in 64-bit mode.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PUSHF
PUSHFD
PUSHFQ

Push rFLAGS onto Stack

Decrements the rSP register and copies the rFLAGS register (except for the VM and RF flags) onto the stack. The instruction clears the VM and RF flags in the rFLAGS image before putting it on the stack.

The instruction pushes 2, 4, or 8 bytes, depending on the operand size.

In 64-bit mode, this instruction defaults to a 64-bit operand size and there is no prefix available to encode a 32-bit operand size.

In virtual-8086 mode, if system software has set the IOPL field to a value less than 3, a general-protection exception occurs if application software attempts to execute PUSHF*x* or POPF*x* while VME is not enabled or the operand size is not 16-bit.

Mnemonic	Opcode	Description
PUSHF	9C	Push the FLAGS word onto the stack.
PUSHFD	9C	Push the EFLAGS doubleword onto stack. (No prefix encoding this in 64-bit mode.)
PUSHFQ	9C	Push the RFLAGS quadword onto stack.

Action

```
// See “Pseudocode Definition” on page 57.

PUSHF_START:
IF (REAL_MODE)
    PUSHF_REAL
ELSIF (PROTECTED_MODE)
    PUSHF_PROTECTED
ELSE // (VIRTUAL_MODE)
    PUSHF_VIRTUAL

PUSHF_REAL:
    PUSH.v old_RFLAGS     // Pushed with RF and VM cleared.
 EXIT

PUSHF_PROTECTED:
    PUSH.v old_RFLAGS   // Pushed with RF cleared.
    EXIT

PUSHF_VIRTUAL:
    IF (RFLAGS.IOPL==3)
    {
        PUSH.v old_RFLAGS // Pushed with RF,VM cleared.
        EXIT
    }
```

```
ELSIF ((CR4.VME==1) && (OPERAND_SIZE==16))
{
    PUSH.v old_RFLAGS // Pushed with VIF in the IF position.
                      // Pushed with IOPL=3.
    EXIT
}
ELSE // ((RFLAGS.IOPL<3) && ((CR4.VME==0) || (OPERAND_SIZE!=16)))
    EXCEPTION [#GP(0)]
```

Related Instructions

POPF, POPFD, POPFQ

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP		X		The I/O privilege level was less than 3 and either VME was not enabled or the operand size was not 16-bit.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

RCL Rotate Through Carry Left

Rotates the bits of a register or memory location (first operand) to the left (more significant bit positions) and through the carry flag by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated through the carry flag are rotated back in at the right end (lsb) of the first operand location.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63.

For 1-bit rotates, the instruction sets the OF flag to the logical `xor` of the CF bit (after the rotate) and the most significant bit of the result. When the rotate count is greater than 1, the OF flag is undefined. When the rotate count is 0, no flags are affected.

Mnemonic	Opcode	Description
RCL *reg/mem8*,1	D0 /2	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location left 1 bit.
RCL *reg/mem8*, CL	D2 /2	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location left the number of bits specified in the CL register.
RCL *reg/mem8*, *imm8*	C0 /2 *ib*	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location left the number of bits specified by an 8-bit immediate value.
RCL *reg/mem16*, 1	D1 /2	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location left 1 bit.
RCL *reg/mem16*, CL	D3 /2	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location left the number of bits specified in the CL register.
RCL *reg/mem16*, *imm8*	C1 /2 *ib*	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location left the number of bits specified by an 8-bit immediate value.
RCL *reg/mem32*, 1	D1 /2	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location left 1 bit.
RCL *reg/mem32*, CL	D3 /2	Rotate 33 bits consisting of the carry flag and a 32-bit register or memory location left the number of bits specified in the CL register.
RCL *reg/mem32*, *imm8*	C1 /2 *ib*	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location left the number of bits specified by an 8-bit immediate value.
RCL *reg/mem64*, 1	D1 /2	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location left 1 bit.

Mnemonic	Opcode	Description
RCL *reg/mem64*, CL	D3 /2	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location left the number of bits specified in the CL register.
RCL *reg/mem64*, *imm8*	C1 /2 *ib*	Rotates the 65 bits consisting of the carry flag and a 64-bit register or memory location left the number of bits specified by an 8-bit immediate value.

Related Instructions

RCR, ROL, ROR

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M								M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

RCR Rotate Through Carry Right

Rotates the bits of a register or memory location (first operand) to the right (toward the less significant bit positions) and through the carry flag by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated through the carry flag are rotated back in at the left end (msb) of the first operand location.

The processor masks the upper three bits in the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63.

For 1-bit rotates, the instruction sets the OF flag to the logical `xor` of the two most significant bits of the result. When the rotate count is greater than 1, the OF flag is undefined. When the rotate count is 0, no flags are affected.

Mnemonic	Opcode	Description
RCR *reg/mem8*, 1	D0 /3	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location right 1 bit.
RCR *reg/mem8*,CL	D2 /3	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location right the number of bits specified in the CL register.
RCR *reg/mem8*,*imm8*	C0 /3 *ib*	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location right the number of bits specified by an 8-bit immediate value.
RCR *reg/mem16*,1	D1 /3	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location right 1 bit.
RCR *reg/mem16*,CL	D3 /3	Rotate the17 bits consisting of the carry flag and a 16-bit register or memory location right the number of bits specified in the CL register.
RCR *reg/mem16*, *imm8*	C1 /3 *ib*	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location right the number of bits specified by an 8-bit immediate value.
RCR *reg/mem32*,1	D1 /3	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location right 1 bit.
RCR *reg/mem32*,CL	D3 /3	Rotate 33 bits consisting of the carry flag and a 32-bit register or memory location right the number of bits specified in the CL register.
RCR *reg/mem32*, *imm8*	C1 /3 *ib*	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location right the number of bits specified by an 8-bit immediate value.
RCR *reg/mem64*,1	D1 /3	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location right 1 bit.

Mnemonic	Opcode	Description
RCR *reg/mem64*,CL	D3 /3	Rotate 65 bits consisting of the carry flag and a 64-bit register or memory location right the number of bits specified in the CL register.
RCR *reg/mem64*, *imm8*	C1 /3 *ib*	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location right the number of bits specified by an 8-bit immediate value.

Related Instructions

RCL, ROR, ROL

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M								M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

RDFSBASE Read FS.base
RDGSBASE Read GS.base

Copies the base field of the FS or GS segment descriptor to the specified register. When supported and enabled, these instructions can be executed at any processor privilege level. The RDFSBASE and RDGSBASE instructions are only defined in 64-bit mode.

System software must set the FSGSBASE bit (bit 16) of CR4 to enable the RDFSBASE and RDGSBASE instructions.

Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[FSGSBASE] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Opcode	Description
RDFSBASE *reg32*	F3 0F AE /0	Copy the lower 32 bits of FS.base to the specified general-purpose register.
RDFSBASE *reg64*	F3 0F AE /0	Copy the entire 64-bit contents of FS.base to the specified general-purpose register.
RDGSBASE *reg32*	F3 0F AE /1	Copy the lower 32 bits of GS.base to the specified general-purpose register.
RDGSBASE *reg64*	F3 0F AE /1	Copy the entire 64-bit contents of GS.base to the specified general-purpose register.

Related Instructions

WRFSBASE, WRGSBASE

rFLAGS Affected

None.

Exceptions

Exception	Legacy	Compat-ibility	64-bit	Cause of Exception
#UD	X	X		Instruction is not valid in compatibility or legacy modes.
			X	Instruction not supported as indicated by CPUID Fn0000_0007_EBX_x0[FSGSBASE] = 0 or, if supported, not enabled in CR4.

RDPID Read Processor ID

RDPID reads the value of TSC_AUX MSR used by the RDTSCP instruction into the specified destination register. Normal operand size prefixes do not apply and the update is either 32 bit or 64 bit based on the current mode.

The RDPID instruction can be used to access the TSC_AUX value at CPL > 0 in cases where the operating system has disabled unprivileged execution of the RDTSCP instruction.

The content of the TSC_AUX MSR, including how and even whether it actually indicates a processor ID, is a matter of operating system convention.

The RDPID instruction is supported if the feature flag CPUID Fn0000_0007_X0_ECX[22]=1.

Mnemonic	Opcode	Description
RDPID	F3 0F C7/7	Read TSC_AUX

Related Instructions

RDTSCP

rFLAGS Affected

rNone

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
																M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID Fn0000_0007_ECX[22] = 0.

RDPRU Read Processor Register

RDPRU instruction is used to give access to some processor registers that are typically only accessible when the privilege level is zero. ECX is used as the implicit register to specify which register to read. RDPRU places the specified register's value into EDX:EAX.

The RDPRU instruction normally can be executed at any privilege level. When CR4.TSD=1, RDPRU can only be used when the privilege level is zero. When the CPL>0 with CR4.TSD=1, the RDPRU instruction will generate a #UD fault.

The RDPRU instruction is supported if the feature flag CPUID Fn8000_0008 EBX[4]=1. The 16-bit field in CPUID Fn8000_0008-EDX[31:16] returns the largest ECX value that returns a valid register. Any unsupported ECX values return zero. Registers currently supported by ECX values are:

- ECX Value 0 = Register MPERF
- ECX Value 1 = Register APERF

Mnemonic	Opcode	Description
RDPRU	0F 01 FD	Copy register specified by ECX into EDX:EAX

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	0	0	0	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID Fn8000_0008_EBX[RDPRU] = 0 or CPL>0 and CR4.TSD=1.

RDRAND Read Random

Loads the destination register with a hardware-generated random value.

The size of the returned value in bits is determined by the size of the destination register.

Hardware modifies the CF flag to indicate whether the value returned in the destination register is valid. If CF = 1, the value is valid. If CF = 0, the value is invalid. Software must test the state of the CF flag prior to using the value returned in the destination register to determine if the value is valid. If the returned value is invalid, software must execute the instruction again. Software should implement a retry limit to ensure forward progress of code.

The execution of RDRAND clears the OF, SF, ZF, AF, and PF flags.

Support for the RDRAND instruction is optional. On processors that support the instruction, CPUID Fn0000_0001_ECX[RDRAND] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Opcode	Description
RDRAND *reg16*	0F C7 /6	Load the destination register with a 16-bit random number.
RDRAND *reg32*	0F C7 /6	Load the destination register with a 32-bit random number.
RDRAND *reg64*	0F C7 /6	Load the destination register with a 64-bit random number.

Related Instructions

RDSEED

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	0	0	0	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported as indicated by CPUID Fn0000_0001_ECX[RDRAND] = 0.

RDSEED Read Random Seed

Loads the destination register with a hardware-generated random "seed" value.

The size of the returned value in bits is determined by the size of the destination register.

Hardware modifies the CF flag to indicate whether the value returned in the destination register is valid. If CF = 1, the value is valid. If CF = 0, the value is invalid and will be returned as zero. Software must test the state of the CF flag prior to using the value returned in the destination register to determine if the value is valid. If the returned value is invalid, software must execute the instruction again. Software should implement a retry limit to ensure forward progress of code.

The execution of RDSEED clears the OF, SF, ZF, AF, and PF flags.

Mnemonic	Opcode	Description
RDSEED *reg16*	0F C7 /7	Read 16-bit random seed
RDSEED *reg32*	0F C7 /7	Read 32-bit random seed
RDSEED *reg64*	0F C7 /7	Read 64-bit random seed

Related Instructions

RDRAND

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	0	0	0	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank.Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported as indicated by CPUID Fn0000_0007_EBX_x0[RDSEED] = 0

RET (Near) Near Return from Called Procedure

Returns from a procedure previously entered by a CALL near instruction. This form of the RET instruction returns to a calling procedure within the current code segment.

This instruction pops the rIP from the stack, with the size of the pop determined by the operand size. The new rIP is then zero-extended to 64 bits. The RET instruction can accept an immediate value operand that it adds to the rSP after it pops the target rIP. This action skips over any parameters previously passed back to the subroutine that are no longer needed.

In 64-bit mode, the operand size defaults to 64 bits (eight bytes) without the need for a REX prefix. No prefix is available to encode a 32-bit operand size in 64-bit mode.

See RET (Far) for information on far returns—returns to procedures located outside of the current code segment. For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
RET	C3	Near return to the calling procedure.
RET *imm16*	C2 *iw*	Near return to the calling procedure then pop the specified number of bytes from the stack.

Action

```
RETN_START:

IF (OPCODE == retn imm16)
   temp_IMM = 16 bit immediate from the instruction, zero-extended to 64 bits
ELSE // (OPCODE == retn)
   temp_IMM = 0

IF (stack is not large enough for a v-sized pop)
         EXCEPTION[#SS(0)]

POP.v temp_RIP

IF ((64BIT_MODE) && (temp_RIP is non-canonical) ||
   (!64BIT_MODE) && (temp_RIP > CS.limit))
         EXCEPTION [#GP(0)]

IF (ShadowStacksEnabled at current CPL)
   {
   IF (v == 2)              // operand size = 16
      {
      temp_sstk_RIP = SSTK_READ_MEM.d [SSP]
      SSP = SSP + 4
```

```
      }
   ELSEIF (v == 4)          // operand size = 32
      {
      temp_sstk_RIP = SSTK_READ_MEM.d [SSP]
      SSP = SSP + 4
      }
   ELSE // (v == 8)         // operand size = 64
      {
      temp_sstk_RIP = SSTK_READ_MEM.q [SSP]
      SSP = SSP + 8
      }
   IF (temp_RIP != temp_sstk_RIP)
         EXCEPTION [#CP(RETN)]
   } end shadow stacks enabled

RSP.s = RSP + temp_IMM
RIP   = temp_RIP
EXIT   // end RETN
```

Related Instructions

CALL (Near), CALL (Far), RET (Far)

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
Control-protection, #CP			X	The return address on the program stack did not match the address on the shadow stack.

RET (Far) Far Return from Called Procedure

Returns from a procedure previously entered by a CALL Far instruction. This form of the RET instruction returns to a calling procedure in a different segment than the current code segment. It can return to the same CPL or to a less privileged CPL.

RET Far pops a target CS and rIP from the stack. If the new code segment is less privileged than the current code segment, the stack pointer is incremented by the number of bytes indicated by the immediate operand, if present; then a new SS and rSP are also popped from the stack.

The final value of rSP is incremented by the number of bytes indicated by the immediate operand, if present. This action skips over the parameters (previously passed to the subroutine) that are no longer needed.

All stack pops are determined by the operand size. If necessary, the target rIP is zero-extended to 64 bits before assuming program control.

If the CPL changes, the data segment selectors are set to NULL for any of the data segments (DS, ES, FS, GS) not accessible at the new CPL.

See RET (Near) for information on near returns—returns to procedures located inside the current code segment. For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Mnemonic	Opcode	Description
RETF	CB	Far return to the calling procedure.
RETF *imm16*	CA *iw*	Far return to the calling procedure, then pop the specified number of bytes from the stack.

Action

```
// For functions READ_DESCRIPTOR, ShadowStacksEnabled
// see "Pseudocode Definition" on page 57

RETF_START:

IF (PROTECTED_MODE)
   RETF_PROTECTED
ELSE // (REAL_MODE or VIRTUAL_MODE)
   RETF_REAL_OR_VIRTUAL


RETF_REAL_OR_VIRTUAL:

IF (OPCODE == retf imm16)
   temp_IMM = 16 bit immediate operand, zero-extended to 64 bits
ELSE // (OPCODE == retf)
   temp_IMM = 0
```

```
POP.v temp_RIP
POP.v temp_CS

IF (temp_RIP > CS.limit)
   EXCEPTION [#GP(0)]

CS.sel  = temp_CS
CS.base = temp_CS SHL 4

RSP.s = RSP + temp_IMM
RIP   = temp_RIP
EXIT   // end RETF real or virtual modes


RETF_PROTECTED:

IF (OPCODE == retf imm16)
   temp_IMM = 16 bit immediate operand, zero-extended to 64 bits
ELSE // (OPCODE == retf)
   temp_IMM = 0

POP.v temp_RIP
POP.v temp_CS
temp_CPL = temp_CS.rpl

IF (CPL == temp_CPL) // not changing privilege level
   RETF_PROTECTED_TO_SAME_PRIV
ELSE
   RETF_PROTECTED_TO_OUTER_PRIV


RETF_PROTECTED_TO_SAME_PRIV:
                           // CPL = temp_CS.rpl (RETF to same privilege level)
CS = READ_DESCRIPTOR (temp_CS, iret_chk)

IF ((64BIT_MODE) && (temp_RIP is non-canonical) ||
   (!64BIT_MODE) && (temp_RIP > CS.limit))
         EXCEPTION [#GP(0)]

RIP   = temp_RIP
RSP.s = RSP + temp_IMM

IF (ShadowStacksEnabled(current CPL))
   {
   IF (SSP[2:0] != 0)
        EXCEPTION [#CP(RETF/IRET)]      // SSP must be 8-byte aligned
   temp_sstk_CS      = SSTK_READ_MEM.q [SSP + 16] // read CS from sstk
   temp_sstk_LIP     = SSTK_READ_MEM.q [SSP + 8]  // read LIP
   temp_sstk_prevSSP = SSTK_READ_MEM.q [SSP]      // read previous SSP
   SSP = SSP + 24
```

```
   IF (temp_CS != temp_sstk_CS)
        EXCEPTION [#CP(RETF/IRET)]     // CS mismatch
   IF ((CS.base + RIP) != temp_sstk_LIP)
        EXCEPTION [#CP(RETF/IRET)]     // LIP mismatch
   IF (temp_sstk_prevSSP[1:0] != 0)
        EXCEPTION [#CP(RETF/IRET)]     // prevSSP must be 4-byte aligned
   IF ((COMPATIBILITY_MODE) && (tmp_sstk_prevSSP[63:32] != 0))
        EXCEPTION [#GP(0)]             // prevSSP must be <4GB in compat mode
   IF ((64BIT_MODE) && (temp_sstk_prevSSP is non-canonical))
        EXCEPTION [#GP(0)]
   SSP = temp_sstk_prevSSP
   } // end shadow stacks enabled at current CPL

EXIT  // end RETF to same privilege level

RETF_PROTECTED_TO_OUTER_PRIV:
                        // CPL != temp_CS.rpl (RETF changing privilege level)
POP.v temp_RSP
POP.v temp_SS

CS = READ_DESCRIPTOR (temp_CS, iret_chk)
temp_oldCPL = CPL

IF ((64BIT_MODE) && (temp_RIP is non-canonical) ||
   (!64BIT_MODE) && (temp_RIP > CS.limit))
         EXCEPTION [#GP(0)]

CPL = temp_CPL
SS = READ_DESCRIPTOR (temp_SS, ss_chk)

RIP   = temp_RIP
RSP.s = temp_RSP + temp_IMM

IF (ShadowStacksEnabled(old CPL))
   {
   IF (SSP[2:0] != 0)
         EXCEPTION [#CP(RETF/IRET)]  // SSP must be 8-byte aligned
   temp_sstk_CS  = SSTK_READ_MEM.q [SSP + 16] // read CS from sstk
   temp_sstk_LIP = SSTK_READ_MEM.q [SSP + 8]  // read LIP
   temp_SSP      = SSTK_READ_MEM.q [SSP]      // read previous SSP
   SSP = SSP +24
   IF (temp_CS != temp_sstk_CS)
         EXCEPTION [#CP(RETF/IRET)]    // CS mismatch
   IF ((CS.base + RIP) != temp_sstk_LIP)
         EXCEPTION [#CP(RETF/IRET)]    // LIP mismatch
   IF (temp_SSP[1:0] != 0)
         EXCEPTION [#CP(RETF/IRET)]    // prevSSP must be 4-byte aligned
   IF ((COMPATIBILITY_MODE) && (tmp_sstk_prevSSP[63:32] != 0))
         EXCEPTION [#GP(0)]            // prevSSP must be <4GB in compat mode
   }
temp_oldSSP = SSP
```

```
IF (ShadowStacksEnabled(new CPL))
   {
   IF ((ShadowStacksEnabled(CPL 3) && (old_CPL == 3))
       temp_SSP = PL3_SSP
   IF ((COMPATIBILITY_MODE) && (temp_SSP[63:32] != 0))
         EXCEPTION [#GP(0)]            // SSP must be <4GB in compat mode
   SSP = temp_SSP
   }

IF (ShadowStacksEnabled(old CPL))
   {  // check shadow stack token and clear busy
   bool invalid_token = FALSE
   < start atomic section >
   temp_Token= SSTK_READ_MEM.q [temp_oldSSP] // read supervisor sstk token
   IF ((temp_Token AND 0x01) != 1)
      invalid_Token = TRUE                    // token busy bit must be 1
   IF ((temp_Token AND ~0x01) != temp_oldSSP)
      invalid_Token = TRUE                   // address in token must = old SSP
   IF (!invalid_Token)
      temp_Token = temp_Token AND ~0x01      // if valid clear token busy bit
   SSTK_WRITE_MEM.q [temp_oldSSP] = temp_Token  // writeback token
   < end atomic section >
   } // end shadow stacks enabled

FOR (seg = ES, DS, FS, GS)
   IF ((seg.sel == NULL) || ((seg.attr.dpl < CPL) &&
       ((seg.attr.type == 'data') ||
        (seg.attr.type == 'non-conforming-code'))))
      seg = NULL // can't use lower DPL data segment at higher CPL
                 // also clears RPL of any null selectors
```

Related Instructions

CALL (Near), CALL (Far), RET (Near)

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Segment not present, #NP (selector)			X	The return code segment was marked not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Stack, #SS (selector)			X	The return stack segment was marked not present.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
General protection, #GP (selector)			X	The return code selector was a null selector.
			X	The return stack selector was a null selector and the return mode was non-64-bit mode or CPL was 3.
			X	The return code or stack descriptor exceeded the descriptor table limit.
			X	The return code or stack selector's TI bit was set but the LDT selector was a null selector.
			X	The segment descriptor for the return code was not a code segment.
			X	The RPL of the return code segment selector was less than the CPL.
			X	The return code segment was non-conforming and the segment selector's DPL was not equal to the RPL of the code segment's segment selector.
			X	The return code segment was conforming and the segment selector's DPL was greater than the RPL of the code segment's segment selector.
			X	The segment descriptor for the return stack was not a writable data segment.
			X	The stack segment descriptor DPL was not equal to the RPL of the return code segment selector.
			X	The stack segment selector RPL was not equal to the RPL of the return code segment selector.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned-memory reference was performed while alignment checking was enabled.
Control-protection, #CP			X	The return address on the program stack did not match the address on the shadow stack, or the previous SSP is not 4 byte aligned.

ROL Rotate Left

Rotates the bits of a register or memory location (first operand) to the left (toward the more significant bit positions) by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated out left are rotated back in at the right end (lsb) of the first operand location.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, it masks the upper two bits of the count, providing a count in the range of 0 to 63.

After completing the rotation, the instruction sets the CF flag to the last bit rotated out (the lsb of the result). For 1-bit rotates, the instruction sets the OF flag to the logical `xor` of the CF bit (after the rotate) and the most significant bit of the result. When the rotate count is greater than 1, the OF flag is undefined. When the rotate count is 0, no flags are affected.

Mnemonic	Opcode	Description
ROL *reg/mem8*, 1	D0 /0	Rotate an 8-bit register or memory operand left 1 bit.
ROL *reg/mem8*, CL	D2 /0	Rotate an 8-bit register or memory operand left the number of bits specified in the CL register.
ROL *reg/mem8*, *imm8*	C0 /0 *ib*	Rotate an 8-bit register or memory operand left the number of bits specified by an 8-bit immediate value.
ROL *reg/mem16*, 1	D1 /0	Rotate a 16-bit register or memory operand left 1 bit.
ROL *reg/mem16*, CL	D3 /0	Rotate a 16-bit register or memory operand left the number of bits specified in the CL register.
ROL *reg/mem16*, *imm8*	C1 /0 *ib*	Rotate a 16-bit register or memory operand left the number of bits specified by an 8-bit immediate value.
ROL *reg/mem32*, 1	D1 /0	Rotate a 32-bit register or memory operand left 1 bit.
ROL *reg/mem32*, CL	D3 /0	Rotate a 32-bit register or memory operand left the number of bits specified in the CL register.
ROL *reg/mem32*, *imm8*	C1 /0 *ib*	Rotate a 32-bit register or memory operand left the number of bits specified by an 8-bit immediate value.
ROL *reg/mem64*, 1	D1 /0	Rotate a 64-bit register or memory operand left 1 bit.
ROL *reg/mem64*, CL	D3 /0	Rotate a 64-bit register or memory operand left the number of bits specified in the CL register.
ROL *reg/mem64*, *imm8*	C1 /0 *ib*	Rotate a 64-bit register or memory operand left the number of bits specified by an 8-bit immediate value.

Related Instructions

RCL, RCR, ROR

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M								M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

ROR Rotate Right

Rotates the bits of a register or memory location (first operand) to the right (toward the less significant bit positions) by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated out right are rotated back in at the left end (the most significant bit) of the first operand location.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63.

After completing the rotation, the instruction sets the CF flag to the last bit rotated out (the most significant bit of the result). For 1-bit rotates, the instruction sets the OF flag to the logical `xor` of the two most significant bits of the result. When the rotate count is greater than 1, the OF flag is undefined. When the rotate count is 0, no flags are affected.

Mnemonic	Opcode	Description
ROR *reg/mem8*, 1	D0 /1	Rotate an 8-bit register or memory location right 1 bit.
ROR *reg/mem8*, CL	D2 /1	Rotate an 8-bit register or memory location right the number of bits specified in the CL register.
ROR *reg/mem8*, *imm8*	C0 /1 *ib*	Rotate an 8-bit register or memory location right the number of bits specified by an 8-bit immediate value.
ROR *reg/mem16*, 1	D1 /1	Rotate a 16-bit register or memory location right 1 bit.
ROR *reg/mem16*, CL	D3 /1	Rotate a 16-bit register or memory location right the number of bits specified in the CL register.
ROR *reg/mem16*, *imm8*	C1 /1 *ib*	Rotate a 16-bit register or memory location right the number of bits specified by an 8-bit immediate value.
ROR *reg/mem32*, 1	D1 /1	Rotate a 32-bit register or memory location right 1 bit.
ROR *reg/mem32*, CL	D3 /1	Rotate a 32-bit register or memory location right the number of bits specified in the CL register.
ROR *reg/mem32*, *imm8*	C1 /1 *ib*	Rotate a 32-bit register or memory location right the number of bits specified by an 8-bit immediate value.
ROR *reg/mem64*, 1	D1 /1	Rotate a 64-bit register or memory location right 1 bit.
ROR *reg/mem64*, CL	D3 /1	Rotate a 64-bit register or memory operand right the number of bits specified in the CL register.
ROR *reg/mem64*, *imm8*	C1 /1 *ib*	Rotate a 64-bit register or memory operand right the number of bits specified by an 8-bit immediate value.

Related Instructions

RCL, RCR, ROL

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M								M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

RORX Rotate Right Extended

Rotates the bits of the source operand right (toward the least-significant bit) by the number of bit positions specified in an immediate operand and writes the result to the destination. Does not affect the arithmetic flags.

This instruction has three operands:

RORX *dest*, *src, rot_cnt*

On each right-shift, the bit shifted out of the least-significant bit position is copied to the most-significant bit. This instruction performs a non-destructive operation; that is, the contents of the source operand are unaffected by the operation, unless the destination and source are the same general-purpose register.

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general-purpose register and the source (*src*) is either a general-purpose register or a memory operand. The rotate count *rot_cnt* is encoded in an immediate byte. When the operand size is 32, bits [7:5] of the immediate byte are ignored; when the operand size is 64, bits [7:6] of the immediate byte are ignored.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
RORX r*eg32*, *reg/mem32, imm8*	C4	$\overline{\text{RXB}}$.03	0.1111.0.11	F0 /r ib
RORX *reg64, reg/mem64, imm8*	C4	$\overline{\text{RXB}}$.03	1.1111.0.11	F0 /r ib

Related Instructions

SARX, SHLX, SHRX

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

SAHF Store AH into Flags

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). The instruction ignores bits 1, 3, and 5 of register AH; it sets those bits in the EFLAGS register to 1, 0, and 0, respectively.

The SAHF instruction is available in 64-bit mode if CPUID Fn8000_0001_ECX[LahfSahf] = 1. It is always available in the other operating modes (including compatibility mode)

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Opcode	Description
SAHF	9E	Loads the sign flag, the zero flag, the auxiliary flag, the parity flag, and the carry flag from the AH register into the lower 8 bits of the EFLAGS register.

Related Instructions

LAHF

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
												M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	The SAHF instruction is not supported in 64-bit mode, as indicated by CPUID Fn8000_0001_ECX[LahfSahf] = 0.

SAL
SHL — Shift Left

Shifts the bits of a register or memory location (first operand) to the left through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. For each bit shift, the SAL instruction clears the least-significant bit to 0. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63.

The effect of this instruction is multiplication by powers of two.

For 1-bit shifts, the instruction sets the OF flag to the logical `xor` of the CF bit (after the shift) and the most significant bit of the result. When the shift count is greater than 1, the OF flag is undefined.

If the shift count is 0, no flags are modified.

SHL is an alias to the SAL instruction.

Mnemonic	Opcode	Description
SAL *reg/mem8*, 1	D0 /4	Shift an 8-bit register or memory location left 1 bit.
SAL *reg/mem8*, CL	D2 /4	Shift an 8-bit register or memory location left the number of bits specified in the CL register.
SAL *reg/mem8*, *imm8*	C0 /4 *ib*	Shift an 8-bit register or memory location left the number of bits specified by an 8-bit immediate value.
SAL *reg/mem16*, 1	D1 /4	Shift a 16-bit register or memory location left 1 bit.
SAL *reg/mem16*, CL	D3 /4	Shift a 16-bit register or memory location left the number of bits specified in the CL register.
SAL *reg/mem16*, *imm8*	C1 /4 *ib*	Shift a 16-bit register or memory location left the number of bits specified by an 8-bit immediate value.
SAL *reg/mem32*, 1	D1 /4	Shift a 32-bit register or memory location left 1 bit.
SAL *reg/mem32*, CL	D3 /4	Shift a 32-bit register or memory location left the number of bits specified in the CL register.
SAL *reg/mem32*, *imm8*	C1 /4 *ib*	Shift a 32-bit register or memory location left the number of bits specified by an 8-bit immediate value.
SAL *reg/mem64*, 1	D1 /4	Shift a 64-bit register or memory location left 1 bit.
SAL *reg/mem64*, CL	D3 /4	Shift a 64-bit register or memory location left the number of bits specified in the CL register.
SAL *reg/mem64*, *imm8*	C1 /4 *ib*	Shift a 64-bit register or memory location left the number of bits specified by an 8-bit immediate value.

Mnemonic	Opcode	Description
SHL *reg/mem8*, 1	D0 /4	Shift an 8-bit register or memory location by 1 bit.
SHL *reg/mem8*, CL	D2 /4	Shift an 8-bit register or memory location left the number of bits specified in the CL register.
SHL *reg/mem8*, *imm8*	C0 /4 *ib*	Shift an 8-bit register or memory location left the number of bits specified by an 8-bit immediate value.
SHL *reg/mem16*, 1	D1 /4	Shift a 16-bit register or memory location left 1 bit.
SHL *reg/mem16*, CL	D3 /4	Shift a 16-bit register or memory location left the number of bits specified in the CL register.
SHL *reg/mem16*, *imm8*	C1 /4 *ib*	Shift a 16-bit register or memory location left the number of bits specified by an 8-bit immediate value.
SHL *reg/mem32*, 1	D1 /4	Shift a 32-bit register or memory location left 1 bit.
SHL *reg/mem32*, CL	D3 /4	Shift a 32-bit register or memory location left the number of bits specified in the CL register.
SHL *reg/mem32*, *imm8*	C1 /4 *ib*	Shift a 32-bit register or memory location left the number of bits specified by an 8-bit immediate value.
SHL *reg/mem64*, 1	D1 /4	Shift a 64-bit register or memory location left 1 bit.
SHL *reg/mem64*, CL	D3 /4	Shift a 64-bit register or memory location left the number of bits specified in the CL register.
SHL *reg/mem64*, *imm8*	C1 /4 *ib*	Shift a 64-bit register or memory location left the number of bits specified by an 8-bit immediate value.

Related Instructions

SAR, SHR, SHLD, SHRD

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	U	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS		X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

SAR Shift Arithmetic Right

Shifts the bits of a register or memory location (first operand) to the right through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The SAR instruction does not change the sign bit of the target operand. For each bit shift, it copies the sign bit to the next bit, preserving the sign of the result.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63.

For 1-bit shifts, the instruction clears the OF flag to 0. When the shift count is greater than 1, the OF flag is undefined.

If the shift count is 0, no flags are modified.

Although the SAR instruction effectively divides the operand by a power of 2, the behavior is different from the IDIV instruction. For example, shifting –11 (FFFFFFF5h) by two bits to the right (that is, divide –11 by 4), gives a result of FFFFFFFDh, or –3, whereas the IDIV instruction for dividing –11 by 4 gives a result of –2. This is because the IDIV instruction rounds off the quotient to zero, whereas the SAR instruction rounds off the remainder to zero for positive dividends and to negative infinity for negative dividends. So, for positive operands, SAR behaves like the corresponding IDIV instruction. For negative operands, it gives the same result if and only if all the shifted-out bits are zeroes; otherwise, the result is smaller by 1.

Mnemonic	Opcode	Description
SAR *reg/mem8*, 1	D0 /7	Shift a signed 8-bit register or memory operand right 1 bit.
SAR *reg/mem8*, CL	D2 /7	Shift a signed 8-bit register or memory operand right the number of bits specified in the CL register.
SAR *reg/mem8*, *imm8*	C0 /7 *ib*	Shift a signed 8-bit register or memory operand right the number of bits specified by an 8-bit immediate value.
SAR *reg/mem16*, 1	D1 /7	Shift a signed 16-bit register or memory operand right 1 bit.
SAR *reg/mem16*, CL	D3 /7	Shift a signed 16-bit register or memory operand right the number of bits specified in the CL register.
SAR *reg/mem16*, *imm8*	C1 /7 *ib*	Shift a signed 16-bit register or memory operand right the number of bits specified by an 8-bit immediate value.
SAR *reg/mem32*, 1	D1 /7	Shift a signed 32-bit register or memory location 1 bit.
SAR *reg/mem32*, CL	D3 /7	Shift a signed 32-bit register or memory location right the number of bits specified in the CL register.

Mnemonic	Opcode	Description
SAR *reg/mem32*, *imm8*	C1 /7 *ib*	Shift a signed 32-bit register or memory location right the number of bits specified by an 8-bit immediate value.
SAR *reg/mem64*, 1	D1 /7	Shift a signed 64-bit register or memory location right 1 bit.
SAR *reg/mem64*, CL	D3 /7	Shift a signed 64-bit register or memory location right the number of bits specified in the CL register.
SAR *reg/mem64*, *imm8*	C1 /7 *ib*	Shift a signed 64-bit register or memory location right the number of bits specified by an 8-bit immediate value.

Related Instructions

SAL, SHL, SHR, SHLD, SHRD

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	U	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

SARX Shift Right Arithmetic Extended

Shifts the bits of the first source operand right (toward the least-significant bit) arithmetically by the number of bit positions specified in the second source operand and writes the result to the destination. Does not affect the arithmetic flags.

This instruction has three operands:

SARX *dest*, *src, shft_cnt*

On each right-shift, the most-significant bit (the sign bit) is replicated. This instruction performs a non-destructive operation; that is, the contents of the source operand are unaffected by the operation, unless the destination and source are the same general-purpose register.

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general-purpose register and the first source (*src*) is either a general-purpose register or a memory operand. The second source operand *shft_cnt* is a general-purpose register. When the operand size is 32, bits [31:5] of *shft_cnt* are ignored; when the operand size is 64, bits [63:6] of *shft_cnt* are ignored.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
SARX r*eg32*, *reg/mem32,* r*eg32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.10	F7 /r
SARX *reg64, reg/mem64, reg64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.0.10	F7 /r

Related Instructions

RORX, SHLX, SHRX

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

SBB Subtract with Borrow

Subtracts an immediate value or the value in a register or a memory location (second operand) from a register or a memory location (first operand), and stores the result in the first operand location. If the carry flag (CF) is 1, the instruction subtracts 1 from the result. Otherwise, it operates like SUB.

The SBB instruction sign-extends immediate value operands to the length of the first operand size.

This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a borrow in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result.

This instruction is useful for multibyte (multiword) numbers because it takes into account the borrow from a previous SUB instruction.

The forms of the SBB instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
SBB AL, *imm8*	1C *ib*	Subtract an immediate 8-bit value from the AL register with borrow.
SBB AX, *imm16*	1D *iw*	Subtract an immediate 16-bit value from the AX register with borrow.
SBB EAX, *imm32*	1D *id*	Subtract an immediate 32-bit value from the EAX register with borrow.
SBB RAX, *imm32*	1D *id*	Subtract a sign-extended immediate 32-bit value from the RAX register with borrow.
SBB *reg/mem8*, *imm8*	80 /3 *ib*	Subtract an immediate 8-bit value from an 8-bit register or memory location with borrow.
SBB *reg/mem16*, *imm16*	81 /3 *iw*	Subtract an immediate 16-bit value from a 16-bit register or memory location with borrow.
SBB *reg/mem32*, *imm32*	81 /3 *id*	Subtract an immediate 32-bit value from a 32-bit register or memory location with borrow.
SBB *reg/mem64*, *imm32*	81 /3 *id*	Subtract a sign-extended immediate 32-bit value from a 64-bit register or memory location with borrow.
SBB *reg/mem16*, *imm8*	83 /3 *ib*	Subtract a sign-extended 8-bit immediate value from a 16-bit register or memory location with borrow.
SBB *reg/mem32*, *imm8*	83 /3 *ib*	Subtract a sign-extended 8-bit immediate value from a 32-bit register or memory location with borrow.
SBB *reg/mem64*, *imm8*	83 /3 *ib*	Subtract a sign-extended 8-bit immediate value from a 64-bit register or memory location with borrow.
SBB *reg/mem8*, *reg8*	18 */r*	Subtract the contents of an 8-bit register from an 8-bit register or memory location with borrow.
SBB *reg/mem16*, *reg16*	19 */r*	Subtract the contents of a 16-bit register from a 16-bit register or memory location with borrow.

Mnemonic	Opcode	Description
SBB *reg/mem32*, *reg32*	19 */r*	Subtract the contents of a 32-bit register from a 32-bit register or memory location with borrow.
SBB *reg/mem64*, *reg64*	19 */r*	Subtract the contents of a 64-bit register from a 64-bit register or memory location with borrow.
SBB *reg8*, *reg/mem8*	1A */r*	Subtract the contents of an 8-bit register or memory location from the contents of an 8-bit register with borrow.
SBB *reg16*, *reg/mem16*	1B */r*	Subtract the contents of a 16-bit register or memory location from the contents of a 16-bit register with borrow.
SBB *reg32*, *reg/mem32*	1B */r*	Subtract the contents of a 32-bit register or memory location from the contents of a 32-bit register with borrow.
SBB *reg64*, *reg/mem64*	1B */r*	Subtract the contents of a 64-bit register or memory location from the contents of a 64-bit register with borrow.

Related Instructions

SUB, ADD, ADC

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

SCAS
SCASB
SCASW
SCASD
SCASQ

Scan String

Compares the AL, AX, EAX, or RAX register with the byte, word, doubleword, or quadword pointed to by ES:rDI, sets the status flags in the rFLAGS register according to the results, and then increments or decrements the rDI register according to the state of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments the rDI register; otherwise, it decrements it. The instruction increments or decrements the rDI register by 1, 2, 4, or 8, depending on the size of the operands.

The forms of the SCASx instruction with an explicit operand address the operand at ES:rDI. The explicit operand serves only to specify the size of the values being compared.

The no-operands forms of the instruction use the ES:rDI registers to point to the value to be compared. The mnemonic determines the size of the operands and the specific register containing the other comparison value.

For block comparisons, the SCASx instructions support the REPE or REPZ prefixes (they are synonyms) and the REPNE or REPNZ prefixes (they are synonyms). For details about the REP prefixes, see “Repeat Prefixes” on page 12. A SCAS*x* instruction can also operate inside a loop controlled by the LOOP*cc* instruction.

Mnemonic	Opcode	Description
SCAS *mem8*	AE	Compare the contents of the AL register with the byte at ES:rDI, and then increment or decrement rDI.
SCAS *mem16*	AF	Compare the contents of the AX register with the word at ES:rDI, and then increment or decrement rDI.
SCAS *mem32*	AF	Compare the contents of the EAX register with the doubleword at ES:rDI, and then increment or decrement rDI.
SCAS *mem64*	AF	Compare the contents of the RAX register with the quadword at ES:rDI, and then increment or decrement rDI.
SCASB	AE	Compare the contents of the AL register with the byte at ES:rDI, and then increment or decrement rDI.
SCASW	AF	Compare the contents of the AX register with the word at ES:rDI, and then increment or decrement rDI.

Mnemonic	Opcode	Description
SCASD	AF	Compare the contents of the EAX register with the doubleword at ES:rDI, and then increment or decrement rDI.
SCASQ	AF	Compare the contents of the RAX register with the quadword at ES:rDI, and then increment or decrement rDI.

Related Instructions

CMP, CMPS*x*

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP			X	A null ES segment was used to reference memory.
	X	X	X	A memory address exceeded the ES segment limit or was non-canonical.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

SET*cc* Set Byte on Condition

Checks the status flags in the rFLAGS register and, if the flags meet the condition specified in the mnemonic (*cc*), sets the value in the specified 8-bit memory location or register to 1. If the flags do not meet the specified condition, SET*cc* clears the memory location or register to 0.

Mnemonics with the A (above) and B (below) tags are intended for use when performing unsigned integer comparisons; those with G (greater) and L (less) tags are intended for use with signed integer comparisons.

Software typically uses the SET*cc* instructions to set logical indicators. Like the CMOV*cc* instructions (page 152), the SET*cc* instructions can replace two instructions—a conditional jump and a move. Replacing conditional jumps with conditional sets can help avoid branch-prediction penalties that may result from conditional jumps.

If the logical value “true” (logical one) is represented in a high-level language as an integer with all bits set to 1, software can accomplish such representation by first executing the opposite SET*cc* instruction—for example, the opposite of SETZ is SETNZ—and then decrementing the result.

A ModR/M byte is used to identify the operand. The *reg* field in the ModR/M byte is unused.

Mnemonic	Opcode	Description
SETO *reg/mem8*	0F 90 /0	Set byte if overflow (OF = 1).
SETNO *reg/mem8*	0F 91 /0	Set byte if not overflow (OF = 0).
SETB *reg/mem8* SETC *reg/mem8* SETNAE *reg/mem8*	0F 92 /0	Set byte if below (CF = 1). Set byte if carry (CF = 1). Set byte if not above or equal (CF = 1).
SETNB *reg/mem8* SETNC *reg/mem8* SETAE *reg/mem8*	0F 93 /0	Set byte if not below (CF = 0). Set byte if not carry (CF = 0). Set byte if above or equal (CF = 0).
SETZ *reg/mem8* SETE *reg/mem8*	0F 94 /0	Set byte if zero (ZF = 1). Set byte if equal (ZF = 1).
SETNZ *reg/mem8* SETNE *reg/mem8*	0F 95 /0	Set byte if not zero (ZF = 0). Set byte if not equal (ZF = 0).
SETBE *reg/mem8* SETNA *reg/mem8*	0F 96 /0	Set byte if below or equal (CF = 1 or ZF = 1). Set byte if not above (CF = 1 or ZF = 1).
SETNBE *reg/mem8* SETA *reg/mem8*	0F 97 /0	Set byte if not below or equal (CF = 0 and ZF = 0). Set byte if above (CF = 0 and ZF = 0).
SETS *reg/mem8*	0F 98 /0	Set byte if sign (SF = 1).
SETNS *reg/mem8*	0F 99 /0	Set byte if not sign (SF = 0).
SETP *reg/mem8* SETPE *reg/mem8*	0F 9A /0	Set byte if parity (PF = 1). Set byte if parity even (PF = 1).
SETNP *reg/mem8* SETPO *reg/mem8*	0F 9B /0	Set byte if not parity (PF = 0). Set byte if parity odd (PF = 0).

Mnemonic	Opcode	Description
SETL *reg/mem8* SETNGE *reg/mem8*	0F 9C /0	Set byte if less (SF <> OF). Set byte if not greater or equal (SF <> OF).
SETNL *reg/mem8* SETGE *reg/mem8*	0F 9D /0	Set byte if not less (SF = OF). Set byte if greater or equal (SF = OF).
SETLE *reg/mem8* SETNG *reg/mem8*	0F 9E /0	Set byte if less or equal (ZF = 1 or SF <> OF). Set byte if not greater (ZF = 1 or SF <> OF).
SETNLE *reg/mem8* SETG *reg/mem8*	0F 9F /0	Set byte if not less or equal (ZF = 0 and SF = OF). Set byte if greater (ZF = 0 and SF = OF).

Related Instructions

None

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.

SFENCE Store Fence

Acts as a barrier to force strong memory ordering (serialization) between store instructions preceding the SFENCE and store instructions that follow the SFENCE. Stores to differing memory types, or within the WC memory type, may become visible out of program order; the SFENCE instruction ensures that the system completes all previous stores in such a way that they are globally visible before executing subsequent stores. This includes emptying the store buffer and all write-combining buffers.

The SFENCE instruction is weakly-ordered with respect to load instructions, data and instruction prefetches, and the LFENCE instruction. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around an SFENCE.

In addition to store instructions, SFENCE is strongly ordered with respect to other SFENCE instructions, MFENCE instructions, and serializing instructions. Further details on the use of MFENCE to order accesses among differing memory types may be found in *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, section 7.4 "Memory Types" on page 172.

The SFENCE instruction is an SSE1 instruction. Support for SSE1 instructions is indicated by CPUID Fn0000_0001_EDX[SSE] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, "Instruction Subsets and CPUID Feature Flags," on page 587.

Mnemonic	Opcode	Description
SFENCE	0F AE F8	Force strong ordering of (serialized) store operations.

Related Instructions

LFENCE, MFENCE, MCOMMIT

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid Opcode, #UD	X	X	X	The SSE instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[25]=0; and the AMD extensions to MMX are not supported, as indicated by CPUID Fn8000_0001_EDX[22]=0.

SHL Shift Left

This instruction is synonymous with the SAL instruction. For information, see “SAL SHL” on page 315.

SHLD Shift Left Double

Shifts the bits of a register or memory location (first operand) to the left by the number of bit positions in an unsigned immediate value or the CL register (third operand), and shifts in a bit pattern (second operand) from the right. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63. If the masked count is greater than the operand size, the result in the destination register is undefined.

If the shift count is 0, no flags are modified.

If the count is 1 and the sign of the operand being shifted changes, the instruction sets the OF flag to 1. If the count is greater than 1, OF is undefined.

Mnemonic	Opcode	Description
SHLD *reg/mem16*, *reg16*, *imm8*	0F A4 */r ib*	Shift bits of a 16-bit destination register or memory operand to the left the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.
SHLD *reg/mem16*, *reg16*, CL	0F A5 */r*	Shift bits of a 16-bit destination register or memory operand to the left the number of bits specified in the CL register, while shifting in bits from the second operand.
SHLD *reg/mem32*, *reg32*, *imm8*	0F A4 */r ib*	Shift bits of a 32-bit destination register or memory operand to the left the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.
SHLD *reg/mem32*, *reg32*, CL	0F A5 */r*	Shift bits of a 32-bit destination register or memory operand to the left the number of bits specified in the CL register, while shifting in bits from the second operand.
SHLD *reg/mem64*, *reg64*, *imm8*	0F A4 */r ib*	Shift bits of a 64-bit destination register or memory operand to the left the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.
SHLD *reg/mem64*, *reg64*, CL	0F A5 */r*	Shift bits of a 64-bit destination register or memory operand to the left the number of bits specified in the CL register, while shifting in bits from the second operand.

Related Instructions

SHRD, SAL, SAR, SHR, SHL

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	U	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

SHLX Shift Left Logical Extended

Shifts the bits of the first source operand left (toward the most-significant bit) by the number of bit positions specified in the second source operand and writes the result to the destination. Does not affect the arithmetic flags.

This instruction has three operands:

SHLX *dest*, *src, shft_cnt*

On each left-shift, a zero is shifted into the least-significant bit position. This instruction performs a non-destructive operation; that is, the contents of the source operand are unaffected by the operation, unless the destination and source are the same general-purpose register.

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general-purpose register and the first source (*src*) is either a general-purpose register or a memory operand. The second source operand *shft_cnt* is a general-purpose register. When the operand size is 32, bits [31:5] of *shft_cnt* are ignored; when the operand size is 64, bits [63:6] of *shft_cnt* are ignored.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
SHLX *reg32*, *reg/mem32*, *reg32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	F7 /r
SHLX *reg64, reg/mem64, reg64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.0.01	F7 /r

Related Instructions

RORX, SARX, SHRX

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

SHR Shift Right

Shifts the bits of a register or memory location (first operand) to the right through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

For each bit shift, the instruction clears the most-significant bit to 0.

The effect of this instruction is unsigned division by powers of two.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63.

For 1-bit shifts, the instruction sets the OF flag to the most-significant bit of the original value. If the count is greater than 1, the OF flag is undefined.

If the shift count is 0, no flags are modified.

Mnemonic	Opcode	Description
SHR *reg/mem8*, 1	D0 /5	Shift an 8-bit register or memory operand right 1 bit.
SHR *reg/mem8*, CL	D2 /5	Shift an 8-bit register or memory operand right the number of bits specified in the CL register.
SHR *reg/mem8*, *imm8*	C0 /5 *ib*	Shift an 8-bit register or memory operand right the number of bits specified by an 8-bit immediate value.
SHR *reg/mem16*, 1	D1 /5	Shift a 16-bit register or memory operand right 1 bit.
SHR *reg/mem16*, CL	D3 /5	Shift a 16-bit register or memory operand right the number of bits specified in the CL register.
SHR *reg/mem16*, *imm8*	C1 /5 *ib*	Shift a 16-bit register or memory operand right the number of bits specified by an 8-bit immediate value.
SHR *reg/mem32*, 1	D1 /5	Shift a 32-bit register or memory operand right 1 bit.
SHR *reg/mem32*, CL	D3 /5	Shift a 32-bit register or memory operand right the number of bits specified in the CL register.
SHR *reg/mem32*, *imm8*	C1 /5 *ib*	Shift a 32-bit register or memory operand right the number of bits specified by an 8-bit immediate value.
SHR *reg/mem64*, 1	D1 /5	Shift a 64-bit register or memory operand right 1 bit.
SHR *reg/mem64*, CL	D3 /5	Shift a 64-bit register or memory operand right the number of bits specified in the CL register.
SHR *reg/mem64*, *imm8*	C1 /5 *ib*	Shift a 64-bit register or memory operand right the number of bits specified by an 8-bit immediate value.

Related Instructions

SHL, SAL, SAR, SHLD, SHRD

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	U	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

SHRD Shift Right Double

Shifts the bits of a register or memory location (first operand) to the right by the number of bit positions in an unsigned immediate value or the CL register (third operand), and shifts in a bit pattern (second operand) from the left. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63. If the masked count is greater than the operand size, the result in the destination register is undefined.

If the shift count is 0, no flags are modified.

If the count is 1 and the sign of the value being shifted changes, the instruction sets the OF flag to 1. If the count is greater than 1, the OF flag is undefined.

Mnemonic	Opcode	Description
SHRD *reg/mem16*, *reg16*, *imm8*	0F AC */r ib*	Shift bits of a 16-bit destination register or memory operand to the right the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.
SHRD *reg/mem16*, *reg16*, CL	0F AD */r*	Shift bits of a 16-bit destination register or memory operand to the right the number of bits specified in the CL register, while shifting in bits from the second operand.
SHRD *reg/mem32*, *reg32*, *imm8*	0F AC */r ib*	Shift bits of a 32-bit destination register or memory operand to the right the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.
SHRD *reg/mem32*, *reg32*, CL	0F AD */r*	Shift bits of a 32-bit destination register or memory operand to the right the number of bits specified in the CL register, while shifting in bits from the second operand.
SHRD *reg/mem64*, *reg64*, *imm8*	0F AC */r ib*	Shift bits of a 64-bit destination register or memory operand to the right the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.
SHRD *reg/mem64*, *reg64*, CL	0F AD */r*	Shift bits of a 64-bit destination register or memory operand to the right the number of bits specified in the CL register, while shifting in bits from the second operand.

Related Instructions

SHLD, SHR, SHL, SAR, SAL

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	U	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

SHRX Shift Right Logical Extended

Shifts the bits of the first source operand right (toward the least-significant bit) by the number of bit positions specified in the second source operand and writes the result to the destination. Does not affect the arithmetic flags.

This instruction has three operands:

SHRX *dest*, *src, shft_cnt*

On each right-shift, a zero is shifted into the most-significant bit position. This instruction performs a non-destructive operation; that is, the contents of the source operand are unaffected by the operation, unless the destination and source are the same general-purpose register.

In 64-bit mode, the operand size is determined by the value of VEX.W. If VEX.W is 1, the operand size is 64 bits; if VEX.W is 0, the operand size is 32 bits. In 32-bit mode, VEX.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general-purpose register and the first source (*src*) is either a general-purpose register or a memory operand. The second source operand *shft_cnt* is a general-purpose register. When the operand size is 32, bits [31:5] of *shft_cnt* are ignored; when the operand size is 64, bits [63:6] of *shft_cnt* are ignored.

This instruction is a BMI2 instruction. Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
SHRX *reg32*, *reg/mem32, reg32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.11	F7 /r
SHRX *reg64, reg/mem64, reg64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.0.11	F7 /r

Related Instructions

RORX, SARX, SHLX

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		BMI2 instructions are only recognized in protected mode.
			X	BMI2 instructions are not supported, as indicated by CPUID Fn0000_0007_EBX_x0[BMI2] = 0.
			X	VEX.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

SLWPCB Store Lightweight Profiling Control Block Address

Flushes Lightweight Profiling (LWP) state to memory and returns the current effective address of the Lightweight Profiling Control Block (LWPCB) in the specified register. The LWPCB address returned is truncated to 32 bits if the operand size is 32.

If LWP is not currently enabled, SLWPCB sets the specified register to zero.

The flush operation stores the internal event counters for active events and the current ring buffer head pointer into the LWPCB. If there is an unwritten event record pending, it is written to the event ring buffer.

The LWP_CBADDR MSR holds the linear address of the current LWPCB. If the contents of LWP_CBADDR is not zero, the value returned in the specified register is an effective address that is calculated by subtracting the current DS.Base address from the linear address kept in LWP_CBADDR. Note that if DS has changed between the time LLWPCB was executed and the time SLWPCB is executed, this might result in an address that is not currently accessible by the application.

SLWPCB generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

It is possible to execute SLWPCB when the CPL != 3 or when SMM is active, but if the LWPCB pointer is not zero, system software must ensure that the LWPCB and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF fault. Using SLWPCB in these situations is not recommended.

See the discussion of lightweight profiling in Volume 2, Chapter 13 for more information on the use of the LLWPCB, SLWPCB, LWPINS, and LWPVAL instructions.

The SLWPCB instruction is implemented if LWP is supported on a processor. Support for LWP is indicated by CPUID Fn8000_0001_ECX[LWP] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
SLWPCB *reg32*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	12 /1
SLWPCB *reg64*	8F	$\overline{\text{RXB}}$.09	1.1111.0.00	12 /1

ModRM.reg augments the opcode and is assigned the value 001b. ModRM.r/m (augmented by XOP.R) specifies the register in which to put the LWPCB address. ModRM.mod must be 11b.

Related Instructions

LLWPCB, LWPINS, LWPVAL

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SLWPCB instruction is not supported, as indicated by CPUID Fn8000_0001_ECX[LWP] = 0.
	X	X		The system is not in protected mode.
			X	LWP is not available, or mod != 11b, or vvvv != 1111b.
Page fault, #PF			X	A page fault resulted from reading or writing the LWPCB.
			X	A page fault resulted from flushing an event to the ring buffer.

STC Set Carry Flag

Sets the carry flag (CF) in the rFLAGS register to one.

Mnemonic	Opcode	Description
STC	F9	Set the carry flag (CF) to one.

Related Instructions

CLC, CMC

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
																1
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

None

STD Set Direction Flag

Set the direction flag (DF) in the rFLAGS register to 1. If the DF flag is 0, each iteration of a string instruction increments the data pointer (index registers rSI or rDI). If the DF flag is 1, the string instruction decrements the pointer. Use the CLD instruction before a string instruction to make the data pointer increment.

Mnemonic	Opcode	Description
STD	FD	Set the direction flag (DF) to one.

Related Instructions

CLD, INSx, LODSx, MOVSx, OUTSx, SCASx, STOSx, CMPS*x*

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
									1							
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

None

STOS
STOSB
STOSW
STOSD
STOSQ

Store String

Copies a byte, word, doubleword, or quadword from the AL, AX, EAX, or RAX registers to the memory location pointed to by ES:rDI and increments or decrements the rDI register according to the state of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments the pointer; otherwise, it decrements the pointer. It increments or decrements the pointer by 1, 2, 4, or 8, depending on the size of the value being copied.

The forms of the STOS*x* instruction with an explicit operand use the operand only to specify the type (size) of the value being copied.

The no-operands forms specify the type (size) of the value being copied with the mnemonic.

The STOS*x* instructions support the REP prefixes. For details about the REP prefixes, see “Repeat Prefixes” on page 12. The STOS*x* instructions can also operate inside a LOOP*cc* instruction.

Mnemonic	Opcode	Description
STOS *mem8*	AA	Store the contents of the AL register to ES:rDI, and then increment or decrement rDI.
STOS *mem16*	AB	Store the contents of the AX register to ES:rDI, and then increment or decrement rDI.
STOS *mem32*	AB	Store the contents of the EAX register to ES:rDI, and then increment or decrement rDI.
STOS *mem64*	AB	Store the contents of the RAX register to ES:rDI, and then increment or decrement rDI.
STOSB	AA	Store the contents of the AL register to ES:rDI, and then increment or decrement rDI.
STOSW	AB	Store the contents of the AX register to ES:rDI, and then increment or decrement rDI.
STOSD	AB	Store the contents of the EAX register to ES:rDI, and then increment or decrement rDI.
STOSQ	AB	Store the contents of the RAX register to ES:rDI, and then increment or decrement rDI.

Related Instructions

LODSx, MOVSx

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	A memory address exceeded the ES segment limit or was non-canonical.
			X	The ES segment was a non-writable segment.
			X	A null ES segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

SUB Subtract

Subtracts an immediate value or the value in a register or memory location (second operand) from a register or a memory location (first operand) and stores the result in the first operand location. An immediate value is sign-extended to the length of the first operand.

This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a borrow in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result.

The forms of the SUB instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
SUB AL, *imm8*	2C *ib*	Subtract an immediate 8-bit value from the AL register and store the result in AL.
SUB AX, *imm16*	2D *iw*	Subtract an immediate 16-bit value from the AX register and store the result in AX.
SUB EAX, *imm32*	2D *id*	Subtract an immediate 32-bit value from the EAX register and store the result in EAX.
SUB RAX, *imm32*	2D *id*	Subtract a sign-extended immediate 32-bit value from the RAX register and store the result in RAX.
SUB *reg/mem8*, *imm8*	80 /5 *ib*	Subtract an immediate 8-bit value from an 8-bit destination register or memory location.
SUB *reg/mem16*, *imm16*	81 /5 *iw*	Subtract an immediate 16-bit value from a 16-bit destination register or memory location.
SUB *reg/mem32*, *imm32*	81 /5 *id*	Subtract an immediate 32-bit value from a 32-bit destination register or memory location.
SUB *reg/mem64*, *imm32*	81 /5 *id*	Subtract a sign-extended immediate 32-bit value from a 64-bit destination register or memory location.
SUB *reg/mem16*, *imm8*	83 /5 *ib*	Subtract a sign-extended immediate 8-bit value from a 16-bit register or memory location.
SUB *reg/mem32*, *imm8*	83 /5 *ib*	Subtract a sign-extended immediate 8-bit value from a 32-bit register or memory location.
SUB *reg/mem64*, *imm8*	83 /5 *ib*	Subtract a sign-extended immediate 8-bit value from a 64-bit register or memory location.
SUB *reg/mem8*, *reg8*	28 */r*	Subtract the contents of an 8-bit register from an 8-bit destination register or memory location.
SUB *reg/mem16*, *reg16*	29 */r*	Subtract the contents of a 16-bit register from a 16-bit destination register or memory location.
SUB *reg/mem32*, *reg32*	29 */r*	Subtract the contents of a 32-bit register from a 32-bit destination register or memory location.
SUB *reg/mem64*, *reg64*	29 */r*	Subtract the contents of a 64-bit register from a 64-bit destination register or memory location.

Mnemonic	Opcode	Description
SUB *reg8*, *reg/mem8*	2A */r*	Subtract the contents of an 8-bit register or memory operand from an 8-bit destination register.
SUB *reg16*, *reg/mem16*	2B */r*	Subtract the contents of a 16-bit register or memory operand from a 16-bit destination register.
SUB *reg32*, *reg/mem32*	2B */r*	Subtract the contents of a 32-bit register or memory operand from a 32-bit destination register.
SUB *reg64*, *reg/mem64*	2B */r*	Subtract the contents of a 64-bit register or memory operand from a 64-bit destination register.

Related Instructions

ADC, ADD, SBB

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

T1MSKC Inverse Mask From Trailing Ones

Finds the least significant zero bit in the source operand, clears all bits below that bit to 0, sets all other bits to 1 (including the found bit) and writes the result to the destination. If the least significant bit of the source operand is 0, the destination is written with all ones.

This instruction has two operands:

T1MSKC *dest*, *src*

In 64-bit mode, the operand size is determined by the value of XOP.W. If XOP.W is 1, the operand size is 64-bit; if XOP.W is 0, the operand size is 32-bit. In 32-bit mode, XOP.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is a general purpose register or a memory operand.

The T1MSKC instruction effectively performs a bit-wise logical `or` of the inverse of the source operand and the result of incrementing the source operand by 1 and stores the result to the destination register:

```
add tmp1, src, 1
not tmp2, src
or  dest, tmp1, tmp2
```

The value of the carry flag of rFLAGs is generated by the `add` pseudo-instruction and the remaining arithmetic flags are generated by the `or` pseudo-instruction.

The T1MSKC instruction is a TBM instruction. Support for this instruction is indicated by CPUID Fn8000_0001_ECX[TBM] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
T1MSKC *reg32*, *reg/mem32*	8F	$\overline{\text{RXB}}$.09	0.$\overline{\text{dest}}$.0.00	01 /7
T1MSKC *reg64, reg/mem64*	8F	$\overline{\text{RXB}}$.09	1.$\overline{\text{dest}}$.0.00	01 /7

Related Instructions

ANDN, BEXTR, BLCFILL, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, TZMSK, TZCNT

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL		OF	DF	IF	TF	SF	ZF	AF	PF	CF
									0				M	M	U	U	M
21	20	19	18	17	16	14	13	12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		TBM instructions are only recognized in protected mode.
			X	TBM instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[TBM] = 0.
			X	XOP.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

TEST Test Bits

Performs a bit-wise logical and on the value in a register or memory location (first operand) with an immediate value or the value in a register (second operand) and sets the flags in the rFLAGS register based on the result.

This instruction has two operands:

TEST *dest*, *src*

While the AND instruction changes the contents of the destination and the flag bits, the TEST instruction changes only the flag bits.

Mnemonic	Opcode	Description
TEST AL, *imm8*	A8 *ib*	and an immediate 8-bit value with the contents of the AL register and set rFLAGS to reflect the result.
TEST AX, *imm16*	A9 *iw*	and an immediate 16-bit value with the contents of the AX register and set rFLAGS to reflect the result.
TEST EAX, *imm32*	A9 *id*	and an immediate 32-bit value with the contents of the EAX register and set rFLAGS to reflect the result.
TEST RAX, *imm32*	A9 *id*	and a sign-extended immediate 32-bit value with the contents of the RAX register and set rFLAGS to reflect the result.
TEST *reg/mem8*, *imm8*	F6 /0 *ib*	and an immediate 8-bit value with the contents of an 8-bit register or memory operand and set rFLAGS to reflect the result.
TEST *reg/mem16*, *imm16*	F7 /0 *iw*	and an immediate 16-bit value with the contents of a 16-bit register or memory operand and set rFLAGS to reflect the result.
TEST *reg/mem32*, *imm32*	F7 /0 *id*	and an immediate 32-bit value with the contents of a 32-bit register or memory operand and set rFLAGS to reflect the result.
TEST *reg/mem64*, *imm32*	F7 /0 *id*	and a sign-extended immediate32-bit value with the contents of a 64-bit register or memory operand and set rFLAGS to reflect the result.
TEST *reg/mem8*, *reg8*	84 */r*	and the contents of an 8-bit register with the contents of an 8-bit register or memory operand and set rFLAGS to reflect the result.
TEST *reg/mem16*, *reg16*	85 */r*	and the contents of a 16-bit register with the contents of a 16-bit register or memory operand and set rFLAGS to reflect the result.
TEST *reg/mem32*, *reg32*	85 */r*	and the contents of a 32-bit register with the contents of a 32-bit register or memory operand and set rFLAGS to reflect the result.
TEST *reg/mem64*, *reg64*	85 */r*	and the contents of a 64-bit register with the contents of a 64-bit register or memory operand and set rFLAGS to reflect the result.

Related Instructions

AND, CMP

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	M	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

TZCNT Count Trailing Zeros

Counts the number of trailing zero bits in the 16-, 32-, or 64-bit general purpose register or memory source operand. Counting starts upward from the least significant bit and stops when the lowest bit having a value of 1 is encountered or when the most significant bit is encountered. The count is written to the destination register.

If the input operand is zero, CF is set to 1 and the size (in bits) of the input operand is written to the destination register. Otherwise, CF is cleared.

If the least significant bit is a one, the ZF flag is set to 1 and zero is written to the destination register. Otherwise, ZF is cleared.

TZCNT is a BMI instruction. Support for BMI instructions is indicated by CPUID Fn0000_0007_EBX_x0[BMI] = 1. If the TZCNT instruction is not available, the encoding is treated as the BSF instruction. Software *must* check the CPUID bit once per program or library initialization before using the TZCNT instruction or inconsistent behavior may result.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic		Opcode	Description
TZCNT	*reg16*, *reg/mem16*	F3 0F BC /r	Count the number of trailing zeros in reg/mem16.
TZCNT	*reg32*, *reg/mem32*	F3 0F BC /r	Count the number of trailing zeros in reg/mem32.
TZCNT	*reg64*, *reg/mem64*	F3 0F BC /r	Count the number of trailing zeros in reg/mem64.

Related Instructions

ANDN, BEXTR, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZMSK

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								U				U	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

TZMSK Mask From Trailing Zeros

Finds the least significant one bit in the source operand, sets all bits below that bit to 1, clears all other bits to 0 (including the found bit) and writes the result to the destination. If the least significant bit of the source operand is 1, the destination is written with all zeros.

This instruction has two operands:

TZMSK *dest*, *src*

In 64-bit mode, the operand size is determined by the value of XOP.W. If XOP.W is 1, the operand size is 64-bit; if XOP.W is 0, the operand size is 32-bit. In 32-bit mode, XOP.W is ignored. 16-bit operands are not supported.

The destination (*dest*) is a general purpose register.

The source operand (*src*) is a general purpose register or a memory operand.

The TZMSK instruction effectively performs a bit-wise logical `and` of the negation of the source operand and the result of subtracting 1 from the source operand, and stores the result to the destination register:

```
sub tmp1, src, 1
not tmp2, src
and dest, tmp1, tmp2
```

The value of the carry flag of rFLAGs is generated by the `sub` pseudo-instruction and the remaining arithmetic flags are generated by the `and` pseudo-instruction.

The TZMSK instruction is a TBM instruction. Support for this instruction is indicated by CPUID Fn8000_0001_ECX[TBM] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
TZMSK *reg32*, *reg/mem32*	8F	$\overline{\text{RXB}}$.09	0.$\overline{\text{dest}}$.0.00	01 /4
TZMSK *reg64, reg/mem64*	8F	$\overline{\text{RXB}}$.09	1.$\overline{\text{dest}}$.0.00	01 /4

Related Instructions

ANDN, BEXTR, BLCFILL, BLCI, BLCIC, BLCMSK, BLCS, BLSFILL, BLSI, BLSIC, BLSR, BLSMSK, BSF, BSR, LZCNT, POPCNT, T1MSKC, TZCNT

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	U	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		TBM instructions are only recognized in protected mode.
			X	TBM instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[TBM] = 0.
			X	XOP.L is 1.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

UD0, UD1, UD2 Undefined Operation

These opcodes generate an invalid opcode exception. Unlike other undefined opcodes that may be defined as legal instructions in the future, these opcodes are intended to stay undefined. On some AMD64 processor implementations, UD1 may report an invalid opcode exception regardless of whether fetching the ModRM byte could trigger a paging or segmentation exception.

Mnemonic	Opcode	Description
UD0	0F FF	Raise an invalid opcode exception
UD1	0F B9 /r	Raise an invalid opcode exception
UD2	0F 0B	Raise an invalid opcode exception.

Related Instructions

None

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	This instruction is not recognized.

WRFSBASE Write FS.base
WRGSBASE Write GS.base

Writes the base field of the FS or GS segment descriptor with the value contained in the register operand. When supported and enabled, these instructions can be executed at any processor privilege level. Instructions are only defined in 64-bit mode. The address written to the base field must be in canonical form or a #GP fault will occur.

System software must set the FSGSBASE bit (bit 16) of CR4 to enable the WRFSBASE and WRGSBASE instructions.

Support for this instruction is indicated by CPUID Fn0000_0007_EBX_x0[FSGSBASE] = 1.

For more information on using the CPUID instruction, see the instruction reference page for the CPUID instruction on page 165. For a description of all feature flags related to instruction subset support, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587.

Mnemonic	Opcode	Description
WRFSBASE *reg32*	F3 0F AE /2	Copy the contents of the specified 32-bit general-purpose register to the lower 32 bits of FS.base.
WRFSBASE *reg64*	F3 0F AE /2	Copy the contents of the specified 64-bit general-purpose register to FS.base.
WRGSBASE *reg32*	F3 0F AE /3	Copy the contents of the specified 32-bit general-purpose register to the lower 32 bits of GS.base.
WRGSBASE *reg64*	F3 0F AE /3	Copy the contents of the specified 64-bit general-purpose register to GS.base.

Related Instructions

RDFSBASE, RDGSBASE

rFLAGS Affected

None.

Exceptions

Exception	Legacy	Compatibility	64-bit	Cause of Exception
#UD	X	X		Instruction is not valid in compatibility or legacy modes.
			X	Instruction not supported as indicated by CPUID Fn0000_0007_EBX_x0[FSGSBASE] = 0 or, if supported, not enabled in CR4.
#GP			X	Attempt to write non-canonical address to segment base address.

XADD Exchange and Add

Exchanges the contents of a register (second operand) with the contents of a register or memory location (first operand), computes the sum of the two values, and stores the result in the first operand location.

The forms of the XADD instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

Mnemonic	Opcode	Description
XADD *reg/mem8*, *reg8*	0F C0 */r*	Exchange the contents of an 8-bit register with the contents of an 8-bit destination register or memory operand and load their sum into the destination.
XADD *reg/mem16*, *reg16*	0F C1 */r*	Exchange the contents of a 16-bit register with the contents of a 16-bit destination register or memory operand and load their sum into the destination.
XADD *reg/mem32*, *reg32*	0F C1 */r*	Exchange the contents of a 32-bit register with the contents of a 32-bit destination register or memory operand and load their sum into the destination.
XADD *reg/mem64*, *reg64*	0F C1 */r*	Exchange the contents of a 64-bit register with the contents of a 64-bit destination register or memory operand and load their sum into the destination.

Related Instructions

None

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

XCHG Exchange

Exchanges the contents of the two operands. The operands can be two general-purpose registers or a register and a memory location. If either operand references memory, the processor locks automatically, whether or not the LOCK prefix is used and independently of the value of IOPL. For details about the LOCK prefix, see “Lock Prefix” on page 11.

The x86 architecture commonly uses the XCHG EAX, EAX instruction (opcode 90h) as a one-byte NOP. In 64-bit mode, the processor treats opcode 90h as a true NOP only if it would exchange rAX with itself. Without this special handling, the instruction would zero-extend the upper 32 bits of RAX, and thus it would not be a true no-operation. Opcode 90h can still be used to exchange rAX and r8 if the appropriate REX prefix is used.

This special handling does not apply to the two-byte ModRM form of the XCHG instruction.

Mnemonic	Opcode	Description
XCHG AX, *reg16*	90 +*rw*	Exchange the contents of the AX register with the contents of a 16-bit register.
XCHG *reg16*, AX	90 +*rw*	Exchange the contents of a 16-bit register with the contents of the AX register.
XCHG EAX, *reg32*	90 +*rd*	Exchange the contents of the EAX register with the contents of a 32-bit register.
XCHG *reg32*, EAX	90 +*rd*	Exchange the contents of a 32-bit register with the contents of the EAX register.
XCHG RAX, *reg64*	90 +*rq*	Exchange the contents of the RAX register with the contents of a 64-bit register.
XCHG *reg64*, RAX	90 +*rq*	Exchange the contents of a 64-bit register with the contents of the RAX register.
XCHG *reg/mem8*, *reg8*	86 */r*	Exchange the contents of an 8-bit register with the contents of an 8-bit register or memory operand.
XCHG *reg8*, *reg/mem8*	86 */r*	Exchange the contents of an 8-bit register or memory operand with the contents of an 8-bit register.
XCHG *reg/mem16*, *reg16*	87 */r*	Exchange the contents of a 16-bit register with the contents of a 16-bit register or memory operand.
XCHG *reg16*, *reg/mem16*	87 */r*	Exchange the contents of a 16-bit register or memory operand with the contents of a 16-bit register.
XCHG *reg/mem32*, *reg32*	87 */r*	Exchange the contents of a 32-bit register with the contents of a 32-bit register or memory operand.
XCHG *reg32*, *reg/mem32*	87 */r*	Exchange the contents of a 32-bit register or memory operand with the contents of a 32-bit register.
XCHG *reg/mem64*, *reg64*	87 */r*	Exchange the contents of a 64-bit register with the contents of a 64-bit register or memory operand.
XCHG *reg64*, *reg/mem64*	87 */r*	Exchange the contents of a 64-bit register or memory operand with the contents of a 64-bit register.

Related Instructions

BSWAP, XADD

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The source or destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

XLAT
XLATB
Translate Table Index

Uses the unsigned integer in the AL register as an offset into a table and copies the contents of the table entry at that location to the AL register.

The instruction uses *seg*:[rBX] as the base address of the table. The value of *seg* defaults to the DS segment, but may be overridden by a segment prefix.

This instruction writes AL without changing RAX[63:8]. This instruction ignores operand size.

The single-operand form of the XLAT instruction uses the operand to document the segment and address size attribute, but it uses the base address specified by the rBX register.

This instruction is often used to translate data from one format (such as ASCII) to another (such as EBCDIC).

Mnemonic	Opcode	Description
XLAT *mem8*	D7	Set AL to the contents of DS:[rBX + unsigned AL].
XLATB	D7	Set AL to the contents of DS:[rBX + unsigned AL].

Related Instructions

None

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.

XOR Logical Exclusive OR

Performs a bit-wise logical xor operation on both operands and stores the result in the first operand location. The first operand can be a register or memory location. The second operand can be an immediate value, a register, or a memory location. XOR-ing a register with itself clears the register.

The forms of the XOR instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 11.

The instruction performs the following operation for each bit:

X	Y	X xor Y
0	0	0
0	1	1
1	0	1
1	1	0

Mnemonic	Opcode	Description
XOR AL, *imm8*	34 *ib*	xor the contents of AL with an immediate 8-bit operand and store the result in AL.
XOR AX, *imm16*	35 *iw*	xor the contents of AX with an immediate 16-bit operand and store the result in AX.
XOR EAX, *imm32*	35 *id*	xor the contents of EAX with an immediate 32-bit operand and store the result in EAX.
XOR RAX, *imm32*	35 *id*	xor the contents of RAX with a sign-extended immediate 32-bit operand and store the result in RAX.
XOR *reg/mem8*, *imm8*	80 /6 *ib*	xor the contents of an 8-bit destination register or memory operand with an 8-bit immediate value and store the result in the destination.
XOR *reg/mem16*, *imm16*	81 /6 *iw*	xor the contents of a 16-bit destination register or memory operand with a 16-bit immediate value and store the result in the destination.
XOR *reg/mem32*, *imm32*	81 /6 *id*	xor the contents of a 32-bit destination register or memory operand with a 32-bit immediate value and store the result in the destination.
XOR *reg/mem64*, *imm32*	81 /6 *id*	xor the contents of a 64-bit destination register or memory operand with a sign-extended 32-bit immediate value and store the result in the destination.
XOR *reg/mem16*, *imm8*	83 /6 *ib*	xor the contents of a 16-bit destination register or memory operand with a sign-extended 8-bit immediate value and store the result in the destination.

Mnemonic	Opcode	Description
XOR *reg/mem32*, *imm8*	83 /6 *ib*	xor the contents of a 32-bit destination register or memory operand with a sign-extended 8-bit immediate value and store the result in the destination.
XOR *reg/mem64*, *imm8*	83 /6 *ib*	xor the contents of a 64-bit destination register or memory operand with a sign-extended 8-bit immediate value and store the result in the destination.
XOR *reg/mem8*, *reg8*	30 */r*	xor the contents of an 8-bit destination register or memory operand with the contents of an 8-bit register and store the result in the destination.
XOR *reg/mem16*, *reg16*	31 */r*	xor the contents of a 16-bit destination register or memory operand with the contents of a 16-bit register and store the result in the destination.
XOR *reg/mem32*, *reg32*	31 */r*	xor the contents of a 32-bit destination register or memory operand with the contents of a 32-bit register and store the result in the destination.
XOR *reg/mem64*, *reg64*	31 */r*	xor the contents of a 64-bit destination register or memory operand with the contents of a 64-bit register and store the result in the destination.
XOR *reg8*, *reg/mem8*	32 */r*	xor the contents of an 8-bit destination register with the contents of an 8-bit register or memory operand and store the results in the destination.
XOR *reg16*, *reg/mem16*	33 */r*	xor the contents of a 16-bit destination register with the contents of a 16-bit register or memory operand and store the results in the destination.
XOR *reg32*, *reg/mem32*	33 */r*	xor the contents of a 32-bit destination register with the contents of a 32-bit register or memory operand and store the results in the destination.
XOR *reg64*, *reg/mem64*	33 */r*	xor the contents of a 64-bit destination register with the contents of a 64-bit register or memory operand and store the results in the destination.

Related Instructions

OR, AND, NOT, NEG

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	U	M	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

4 System Instruction Reference

This chapter describes the function, mnemonic syntax, opcodes, affected flags, and possible exceptions generated by the system instructions. System instructions are used to establish the processor operating mode, access processor resources, handle program and system errors, manage memory, and instantiate a virtual machine. Most of these instructions can only be executed by privileged software, such as the operating system or a Virtual Machine Monitor (VMM), also known as a hypervisor. Only system instructions can access certain processor resources, such as the control registers, model-specific registers, and debug registers.

Most system instructions are supported in all hardware implementations of the AMD64 architecture. The table below lists instructions that may not be supported on a given processor implementation. System software must execute the CPUID instruction using the function number listed to determine support prior to using these instructions.

Table 4-1. System Instruction Support Indicated by CPUID Feature Bits

Instruction	CPUID Feature Bit	Register[Bit]
CET_SS	0000_0007_0	ECX[7]
CLAC, STAC	0000_0007_0	EBX[20]
Long Mode and Long Mode instructions	8000_0001_EDX[LM]	EDX[29]
INVPCID	0000_0007_0	EBX[10]
INVLPGB, TLBSYNC	8000_0008_EBX[INVLPGB]	EBX[3]
MONITOR, MWAIT	0000_0001_ECX[MONITOR]	ECX[3]
RDPKRU, WRPKRU	0000_0007_0	ECX[4]
PSMASH, PVALIDATE, RMPADJUST, RMPUPDATE	8000_001F_EAX[SNP]	EAX[4]
RDMSR, WRMSR	0000_0001_EDX[MSR]	EDX[5]
RDTSCP	8000_0001_EDX[RDTSCP]	EDX[27]
SKINIT, STGI	8000_0001_ECX[SKINIT]	ECX[12]
SVM Architecture and instructions	8000_0001_ECX[SVM]	ECX[2]
SYSCALL, SYSRET	8000_0001_EDX[SysCallSysRet]	EDX[11]
SYSENTER, SYSEXIT	0000_0001_EDX[SysEnterSysExit]	EDX[11]
VMGEXIT	8000_001F[SEV-ES]	EAX[3]
WBNOINVD	8000_0008_EBX[WBNOINVD]	EBX[9]

There are also several other CPUID feature bits that indicate support for certain paging functions, virtual-mode extensions, machine-check exceptions, advanced programmable interrupt control (APIC), memory-type range registers (MTRRs), etc.

For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 165. For a comprehensive list of all instruction support feature flags, see Appendix D, “Instruction Subsets and CPUID Feature Flags,” on page 587. For a comprehensive list of all defined CPUID feature numbers and return values, see Appendix E, “Obtaining Processor Information Via the CPUID Instruction,” on page 593.

For further information about the system instructions and register resources, see:

- “System Instructions” in Volume 2.
- “Summary of Registers and Data Types” on page 38.
- “Notation” on page 53.
- “Instruction Prefixes” on page 5.

ARPL Adjust Requestor Privilege Level

Compares the requestor privilege level (RPL) fields of two segment selectors in the source and destination operands of the instruction. If the RPL field of the destination operand is less than the RPL field of the segment selector in the source register, then the zero flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the destination operand remains unchanged and the zero flag is cleared.

The destination operand can be either a 16-bit register or memory location; the source operand must be a 16-bit register.

The ARPL instruction is intended for use by operating-system procedures to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. The segment selector passed to the operating system is placed in the destination operand and the segment selector for the code segment of the application program is placed in the source operand. The RPL field in the source operand represents the privilege level of the application program. The ARPL instruction then insures that the RPL of the segment selector received by the operating system is no lower than the privilege level of the application program.

See “Adjusting Access Rights” in Volume 2, for more information on access rights.

In 64-bit mode, this opcode (63H) is used for the MOVSXD instruction.

Mnemonic	Opcode	Description
ARPL *reg/mem16*, *reg16*	63 */r*	Adjust the RPL of a destination segment selector to a level not less than the RPL of the segment selector specified in the 16-bit source register. (Invalid in 64-bit mode.)

Related Instructions

LAR, LSL, VERR, VERW

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
													M			
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected legacy and compatibility mode.
Stack, #SS			X	A memory address exceeded the stack segment limit.
General protection, #GP			X	A memory address exceeded a data segment limit.
			X	The destination operand was in a non-writable segment.
			X	A null segment selector was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

CLAC Clear Alignment Check Flag

Sets the Alignment Check flag in the rFLAGS register to zero. Support for the CLAC instruction is indicated by CPUID Fn07_EBX[20] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

Mnemonic	Opcode	Description
CLAC	0F 01 CA	Clear AC Flag

Related Instructions

STAC

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
			0													
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank.Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID
		X	X	Instruction is not supported in virtual mode
	X		X	Lock prefix (F0h) preceding opcode.
			X	CPL was not 0

CLGI Clear Global Interrupt Flag

Clears the global interrupt flag (GIF). While GIF is zero, all external interrupts are disabled.

This is a Secure Virtual Machine instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000_0001_ECX[SVM] = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 165.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume-2: System Instructions*, order# 24593.

Mnemonic	Opcode	Description
CLGI	0F 01 DD	Clears the global interrupt flag (GIF).

Related Instructions

STGI

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SVM instructions are not supported as indicated by CPUID Fn8000_0001_ECX[SVM] = 0.
			X	Secure Virtual Machine was not enabled (EFER.SVME=0).
	X	X		Instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not 0.

CLI Clear Interrupt Flag

Clears the interrupt flag (IF) in the rFLAGS register to zero, thereby masking external interrupts received on the INTR input. Interrupts received on the non-maskable interrupt (NMI) input are not affected by this instruction.

In real mode, this instruction clears IF to 0.

In protected mode and virtual-8086-mode, this instruction is IOPL-sensitive. If the CPL is less than or equal to the rFLAGS.IOPL field, the instruction clears IF to 0.

In protected mode, if IOPL < 3, CPL = 3, and protected mode virtual interrupts are enabled (CR4.PVI = 1), then the instruction instead clears rFLAGS.VIF to 0. If none of these conditions apply, the processor raises a general-purpose exception (#GP). For more information, see “Protected Mode Virtual Interrupts” in Volume 2.

In virtual-8086 mode, if IOPL < 3 and the virtual-8086-mode extensions are enabled (CR4.VME = 1), the CLI instruction clears the virtual interrupt flag (rFLAGS.VIF) to 0 instead.

See “Virtual-8086 Mode Extensions” in Volume 2 for more information about IOPL-sensitive instructions.

Mnemonic	Opcode	Description
CLI	FA	Clear the interrupt flag (IF) to zero.

Action

```
IF (CPL <= IOPL)
      RFLAGS.IF = 0

ELSEIF (((VIRTUAL_MODE) && (CR4.VME == 1))
        || ((PROTECTED_MODE) && (CR4.PVI == 1) && (CPL == 3)))
      RFLAGS.VIF = 0;

ELSE
      EXCEPTION[#GP(0)]
```

Related Instructions

STI

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		M								M						
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X		The CPL was greater than the IOPL and virtual mode extensions are not enabled (CR4.VME = 0).
			X	The CPL was greater than the IOPL and either the CPL was not 3 or protected mode virtual interrupts were not enabled (CR4.PVI = 0).

CLTS Clear Task-Switched Flag in CR0

Clears the task-switched (TS) flag in the CR0 register to 0. The processor sets the TS flag on each task switch. The CLTS instruction is intended to facilitate the synchronization of FPU context saves during multitasking operations.

This instruction can only be used if the current privilege level is 0.

See “System-Control Registers” in Volume 2 for more information on FPU synchronization and the TS flag.

Mnemonic	Opcode	Description
CLTS	0F 06	Clear the task-switched (TS) flag in CR0 to 0.

Related Instructions

LMSW, MOV CR*n*

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X	X	CPL was not 0.

CLRSSBSY Clear Shadow Stack Busy

Validates the busy (in use) shadow stack token pointed to by the memory operand and clears the tokens busy bit. If the token validation checks pass, CF is cleared to 0 and SSP is cleared to 0. If the token validation checks fail, CF is set to 1 and the token and SSP are not modified.

CLRSSBY is a privileged instruction and must be executed with CPL=0, otherwise a #GP exception is generated. If shadow stacks are not enabled at the supervisor level, a #UD exception is generated.

Mnemonic	Opcode	Description
CLRSSBSY *mem64*	F3 0F AE /6	Validate shadow stack token and clear busy bit.

Actions

```
// see "Pseudocode Definition" on page 57

IF (CR4.CET == 0)
      EXCEPTION [#UD]
IF (S_CET.SH_STK_EN == 0)
      EXCEPTION [#UD]
IF (CPL != 0)
      EXCEPTION [#GP(0)]

temp_linAdr = Linear_Address(mem64)

IF (temp_linAdr is not 8-byte aligned)
      EXCEPTION [#GP(0)]

bool INVALID_TOKEN = FALSE

< start atomic section >
temp_Token = SSTK_READ_MEM.q [temp_linAdr] // fetch token with locked read

IF ((temp_Token AND 0x01) != 1)
   INVALID_TOKEN = TRUE                    // token busy bit must be set

IF ((temp_Token AND ~0x01) != temp_linAdr)
  INVALID_TOKEN = TRUE          // address in token must equal
                                // linear address of mem64
IF (!INVALID_TOKEN)
   temp_Token = temp_Token AND ~0x01       // valid token, clear busy bit

SSTK_WRITE_MEM.q[temp_linAdr] = temp_Token // write back token and unlock
< end atomic section >

RFLAGS.ZF,PF,AF,OF,SF = 0

IF (INVALID_TOKEN)
   RFLAGS.CF = 1    // set CF if token not valid
ELSE
```

```
   {
    RFLAGS.CF = 0  // else clear CF
    SSP = 0        // and set SSP = 0
   }
EXIT
```

Related Instructions

SETSSBSY

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	0	0	0	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception (vector)	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		Instruction is only recognized in protected mode.
			X	CR4.CET = 0
			X	Shadow stacks not enabled at supervisor level
General protection, #GP			X	CPL ! = 0
			X	The linear address is not 8-byte aligned.
			X	A memory address exceeded a data segment limit.
			X	In long mode, the address of the memory operand was non-canonical.
			X	A null data segment was used to reference memory.
			X	A non-writable data segment was used.
			X	An execute-only code segment was used to reference memory.
Page fault, #PF			X	The linear address is not a supervisor shadow stack page in the OS page tables.
			X	A page fault resulted from the execution of the instruction.

HLT Halt

Causes the microprocessor to halt instruction execution and enter the HALT state. Entering the HALT state puts the processor in low-power mode. Execution resumes when an unmasked hardware interrupt (INTR), non-maskable interrupt (NMI), system management interrupt (SMI), RESET, or INIT occurs.

If an INTR, NMI, or SMI is used to resume execution after a HLT instruction, the saved instruction pointer points to the instruction following the HLT instruction.

Before executing a HLT instruction, hardware interrupts should be enabled. If rFLAGS.IF = 0, the system will remain in a HALT state until an NMI, SMI, RESET, or INIT occurs.

If an SMI brings the processor out of the HALT state, the SMI handler can decide whether to return to the HALT state or not. See *Volume 2: System Programming*, for information on SMIs.

Current privilege level must be 0 to execute this instruction.

Mnemonic	Opcode	Description
HLT	F4	Halt instruction execution.

Related Instructions

STI, CLI

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X	X	CPL was not 0.

INCSSP Increment Shadow Stack Pointer

Increments SSP by the operand size of the instruction multiplied by the unsigned 8-bit value in bits [7:0] of the register operand. The operand size is 8 bytes in 64-bit mode (when REX.W = 1) and is 4 bytes in all other cases.

Before incrementing SSP, the first and last elements of the shadow stack in the range specified by the register operand are read and discarded.

Mnemonic	Opcode	Description
INCSSPD *reg32*	F3 0F AE /05	Increment SSP by 4*(reg32[7:0]).
INCSSPQ *reg64*	F3 0F AE /05	Increment SSP by 8*(reg64[7:0]).

Action

```
IF ((CPL == 3) && (!SSTK_USER_ENABLED))
   EXCEPTION [#UD]
ELSEIF ((CPL < 3) && (!SSTK_SUPV_ENABLED))
   EXCEPTION [#UD]

IF (OPERAND_SIZE == 64)
   {
   temp_numItems = (reg64[7:0] == 0) ? 1 : reg64[7:0]
   temp = SSTK_READ_MEM.q [SSP]                        // touch TOS and last
   temp = SSTK_READ_MEM.q [SSP + temp_numItems*8 - 8] // element in range
   SSP = SSP + reg64[7:0]*8 // increment SSP
   }
ELSE
   {
   temp_numItems = (reg32[7:0] == 0) ? 1 : reg32[7:0]
   temp = SSTK_READ_MEM.d [SSP]                        // touch TOS and last
   temp = SSTK_READ_MEM.d [SSP + temp_numItems*4 - 4] // element in range
   SSP = SSP + reg32[7:0]*4 // increment SSP
   }
EXIT
```

Related Instructions

RDSSP, RSTORSSP

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		Instruction is only recognized in protected mode.
			X	CR4.CET = 0
			X	Shadow stacks are not enabled at the current privilege level.
Page fault, #PF			X	A page fault occurred when touching the first or last element of the shadow stack in the range specified.
			X	The first or last element in the range specified is not in a shadow stack page.
General protection, #GP			X	In long mode, the address of the memory operand was non-canonical.
			X	A memory address exceeded a data segment limit.
			X	A null data segment was used to reference memory.

INT 3 Interrupt to Debug Vector

Calls the debug exception handler. This instruction maps to a 1-byte opcode (CC) that raises a #BP exception. The INT 3 instruction is normally used by debug software to set instruction breakpoints by replacing the first byte of the instruction opcode bytes with the INT 3 opcode.

This one-byte INT 3 instruction behaves differently from the two-byte INT 3 instruction (opcode CD 03) (see “INT” in Chapter 3 “General Purpose Instructions” for further information) in two ways:

The #BP exception is handled without any IOPL checking in virtual x86 mode. (IOPL mismatches will not trigger an exception.)

- In VME mode, the #BP exception is not redirected via the interrupt redirection table. (Instead, it is handled by a protected mode handler.)

Mnemonic	Opcode	Description
INT 3	CC	Trap to debugger at Interrupt 3.

For complete descriptions of the steps performed by INT instructions, see the following:

- *Legacy-Mode Interrupts:* “Legacy Protected-Mode Interrupt Control Transfers” in Volume 2.
- *Long-Mode Interrupts:* “Long-Mode Interrupt Control Transfers” in Volume 2.

Action

```
// Refer to INT instruction’s Action section for the details on INT_N_REAL,
// INT_N_PROTECTED, and INT_N_VIRTUAL_TO_PROTECTED.
INT3_START:

If (REAL_MODE)
    INT_N_REAL                  //N = 3

ELSEIF (PROTECTED_MODE)
    INT_N_PROTECTED             //N = 3

ELSE // VIRTUAL_MODE
    INT_N_VIRTUAL_TO_PROTECTED  //N = 3
```

Related Instructions

INT, INTO, IRET

rFLAGS Affected

If a task switch occurs, all flags are modified; otherwise, setting are as follows:

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
			M	0	0	M				M	0					
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Breakpoint, #BP	X	X	X	INT 3 instruction was executed.
Invalid TSS, #TS (selector)		X	X	As part of a stack switch, the target stack segment selector or rSP in the TSS that was beyond the TSS limit.
		X	X	As part of a stack switch, the target stack segment selector in the TSS was beyond the limit of the GDT or LDT descriptor table.
		X	X	As part of a stack switch, the target stack segment selector in the TSS was a null selector.
		X	X	As part of a stack switch, the target stack segment selector's TI bit was set, but the LDT selector was a null selector.
		X	X	As part of a stack switch, the target stack segment selector in the TSS contained a RPL that was not equal to its DPL.
		X	X	As part of a stack switch, the target stack segment selector in the TSS contained a DPL that was not equal to the CPL of the code segment selector.
		X	X	As part of a stack switch, the target stack segment selector in the TSS was not a writable segment.
Segment not present, #NP (selector)		X	X	The accessed code segment, interrupt gate, trap gate, task gate, or TSS was not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Stack, #SS (selector)		X	X	After a stack switch, a memory address exceeded the stack segment limit or was non-canonical and a stack switch occurred.
		X	X	As part of a stack switch, the SS register was loaded with a non-null segment selector and the segment was marked not present.
General protection, #GP	X	X	X	A memory address exceeded the data segment limit or was non-canonical.
	X	X	X	The target offset exceeded the code segment limit or was non-canonical.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP (selector)	X	X	X	The interrupt vector was beyond the limit of IDT.
		X	X	The descriptor in the IDT was not an interrupt, trap, or task gate in legacy mode or not a 64-bit interrupt or trap gate in long mode.
		X	X	The DPL of the interrupt, trap, or task gate descriptor was less than the CPL.
		X	X	The segment selector specified by the interrupt or trap gate had its TI bit set, but the LDT selector was a null selector.
		X	X	The segment descriptor specified by the interrupt or trap gate exceeded the descriptor table limit or was a null selector.
		X	X	The segment descriptor specified by the interrupt or trap gate was not a code segment in legacy mode, or not a 64-bit code segment in long mode.
			X	The DPL of the segment specified by the interrupt or trap gate was greater than the CPL.
		X		The DPL of the segment specified by the interrupt or trap gate pointed was not 0 or it was a conforming segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

INVD Invalidate Caches

Invalidates all levels of cache associated with this processor. This may or may not include lower level caches associated with another processor that shares any level of this processor's cache hierarchy.

No data is written back to main memory from invalidating the caches.

CPUID Fn8000_001D_EDX[WBINVD]_x*N* indicates the behavior of the processor at various levels of the cache hierarchy. If the feature bit is 0, the instruction causes the invalidation of all lower level caches of other processors sharing the designated level of cache. If the feature bit is 1, the instruction does not necessarily cause the invalidation of all lower level caches of other processors sharing the designated level of cache. See Appendix E, “Obtaining Processor Information Via the CPUID Instruction,” on page 593 for more information on using the CPUID function.

This is a privileged instruction. The current privilege level (CPL) of a procedure invalidating the processor’s internal caches must be 0.

To insure that data is written back to memory prior to invalidating caches, use the WBINVD instruction.

This instruction does not invalidate TLB caches.

INVD is a serializing instruction.

Mnemonic	Opcode	Description
INVD	0F 08	Invalidate internal caches and trigger external cache invalidations.

Related Instructions

WBINVD, WBNOINVD, CLWB, CLFLUSH

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X	X	CPL was not 0.

INVLPG Invalidate TLB Entry

Invalidates the TLB entry that would be used for the 1-byte memory operand.

This instruction invalidates the TLB entry, regardless of the G (Global) bit setting in the associated PDE or PTE entry and regardless of the page size (4 Kbytes, 2 Mbytes, 4 Mbytes, or 1 Gbyte). It may invalidate any number of additional TLB entries, in addition to the targeted entry. INVLPG only invalidates TLB entries tagged with the current PCID and also global pages regardless of PCIDs. If PCIDs are disabled (CR4.PCID=0) then the current PCID is zero.

INVLPG is a serializing instruction and a privileged instruction. The current privilege level must be 0 to execute this instruction.

See “Page Translation and Protection” in Volume 2 for more information on page translation.

Mnemonic	Opcode	Description
INVLPG *mem8*	0F 01 /7	Invalidate the TLB entry for the page containing a specified memory location.

Related Instructions

INVLPGA, INVLPGB, INVPCID, MOV CR*n* (CR3 and CR4)

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X	X	CPL was not 0.

INVLPGA Invalidate TLB Entry in a Specified ASID

Invalidates the TLB mapping for a given virtual page and a given ASID. The virtual (linear) address is specified in the implicit register operand rAX. The portion of rAX used to form the address is determined by the effective address size (current execution mode and optional address size prefix). The ASID is taken from ECX.

The INVLPGA instruction may invalidate any number of additional TLB entries, in addition to the targeted entry.

The INVLPGA instruction is a serializing instruction and a privileged instruction. The current privilege level must be 0 to execute this instruction.

This is a Secure Virtual Machine (SVM) instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000_0001_ECX[SVM] = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 165.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume-2: System Instructions*, order# 24593.

Mnemonic	Opcode	Description
INVLPGA rAX, ECX	0F 01 DF	Invalidates the TLB mapping for the virtual page specified in rAX and the ASID specified in ECX.

Related Instructions

INVLPG, INVLPGB, INVPCID

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SVM instructions are not supported as indicated by CPUID Fn8000_0001_ECX[SVM] = 0.
			X	Secure Virtual Machine was not enabled (EFER.SVME=0).
	X	X		Instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not 0.

INVLPGB Invalidate TLB Entry(s) with Broadcast

Invalidates the TLB entry or entries specified by the descriptor in the rAX:EDX register pair. Invalidations are done both in the local TLB and broadcast to all processors to perform the same invalidations. The virtual (linear) address is specified in the implicit register operand rAX. The portion of rAX used to form the address is determined by the effective address size.

The TLB control field is specified in rAX[5:0]. It determines which components of the address (VA, PCID, ASID) are valid for comparison in the TLB and whether to include global entries in the invalidation process. If rAX[4] is set, only the final translation is invalidated and not the cached upper level TLB entries that lead to the final page. This ability may not be possible with all processors in which case the bit is ignored. If rAX[5] is set, all nested translations that could be used for guest translation selected in rAX[4:0] are flushed. rAX[5] can only be set if CPUID Fn8000_0008_EBX[21 =1. ECX provides a count of the number of pages to include in invalidation with the specified address and the page size at which to increment the specified address.

The descriptor in rAX has the following format:

rAX	Attributes
0	Valid VA
1	Valid PCID
2	Valid ASID
3	Include Global
4	Final Translation Only
5	Include Nested Translations
11:6	Reserved, MBZ
63:12 or 31:12	VA

rAX[3:0] provides for various types of invalidations. A few examples are listed in the following table, but all values are legal.

rAX [3:0]	Action
0xF	Invalidate all TLB entries that match {ASID, PCID, VA} including Global
0xC	Invalidate all TLB entries that match {ASID} including Global
0xD	Invalidate all TLB entries that match {ASID, VA} including Global
0x4	Invalidate all TLB entries that match {ASID} excluding Global

rAX [3:0]	Action
0xE	Invalidate all TLB entries that match {ASID, PCID} including Global
0x6	Invalidate all TLB entries that match {ASID, PCID} excluding Global

The descriptor in EDX has the following format:

EDX	Attributes
15:0	ASID
27:16	PCID
31:28	Reserved, MBZ

ECX[15:0] contains a count of the number of sequential pages to invalidate in addition to the original virtual address, starting from the virtual address specified in rAX. A count of 0 invalidates a single page. ECX[31]=0 indicates to increment the virtual address at the 4K boundary. ECX[31]=1 indicates to increment the virtual address at the 2M boundary. The maximum count supported is reported in CPUID function 8000_0008h, EDX[15:0].

This instruction invalidates the TLB entry or entries, regardless of the page size (4 Kbytes, 2 Mbytes, 4 Mbytes, or 1 Gbyte). It may invalidate any number of additional TLB entries in addition to the targeted entry or entries to accomplish the specified function. INVLPGB follows the same rules for cached upper TLB entries as INVLPG which is controlled by EFER.TCE. However, since this is a broadcast, the invalidation is controlled by the EFER.TCE value on the processor executing the INVLPGB instruction. (See Section 3, “Translation Cache Extension” in AMD64 Architecture Programmer’s Manual Volume 2 for more information on EFER.TCE.)

Under the following circumstances, execution of INVLPGB will result in a General Protection fault (#GP):

- If SVM is disabled, requesting the ASID field with any value but zero, even if the ASID is not necessary for the flush.
- If PCID is disabled, requesting the PCID field with any value but zero, even if the PCID is not necessary for the flush.
- If the request exceeds the number of valid ASIDs for the processor, even if the ASID is not valid.
- Attempts to request a count larger than the maximum count supported, even if the VA is not valid
- Attempts to execute an INVLPGB while in 4M paging mode.

Guest Usage of INVLPGB. Guest usage of INVLPGB is supported only when the instruction has been explicitly enabled by the hypervisor in the VMCB (see APM Volume 2 Appendix B, Table B-1: VMCB Layout, Control Area). Support for INVLPGB/TLBSYNC hypervisor enable in VMCB is indicated by CPUID Fn8000_000A_EDX[24] = 1.

A guest that executes a legal INVLPGB that is not intercepted will have the requested ASID field replaced by the current ASID and the valid ASID bit set before doing the broadcast invalidation. Because of its broadcast nature, the ASID field must be global and all processors must allocate the same ASID to the same Guest for proper operation. Hypervisors that do not support a global ASID must intercept the Guest usage of INVLPGB, if enabled, for proper behavior.

Two forms of INVLPGB intercepts, conditional and unconditional, are available to the hypervisor. The unconditional intercept traps all guest usage of INVLPGB. The conditional intercept traps only illegally-specified INVLPGB instructions. An illegally specified INVLPGB is one that would, if not intercepted, cause a #GP for any reason other than not being executed at CPL 0.

INVLPGB is a privileged instruction but not a serializing instruction. It must be executed at CPL 0, but will broadcast the invalidate to the rest of the processors which may be running at any privilege level.

INVLPGB is weakly ordered as it broadcasts the invalidation types throughout the system to all processors, so that a batch of invalidations can be done in a parallel fashion. For software to guarantee that all processors have seen and done the TLB invalidations, a TLBSYNC must be executed on the initiating processor.

Mnemonic	Opcode	Description
INVLPGB	0F 01 FE	Invalidates TLB entry(s) with Broadcast.

Related Instructions

TLBSYNC, INVLPG, INVLPGA, INVPCID

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
	X	X	X	This instruction is not supported as indicated by CPUID Fn8000_0008_EBX[INVLPGB] = 0.
			X	The hypervisor has not enabled Guest usage of this instruction.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP			X	CPL was not 0.
			X	EAX[11:6] is not zero or EAX[5] not zero if not supported.
				EDX[31:28] is not zero.
			X	CR4.PCID =0 and EDX[PCID] is not zero.
			X	EFER.SVME =0 and EDX[ASID] is not zero.
			X	EDX[ASID] > number of supported ASIDs.
			X	ECX[15:0] > maximum page count supported.
			X	4M paging is active.

INVPCID Invalidate TLB Entry(s) in a Specified PCID

Invalidates the TLB entry or entries on the logical processor for a given PCID in the local TLB based on the operation type specified in the register operand and the PCID and virtual (linear) address specified by the descriptor in the memory operand. (See “Process Context Identifier” in Chapter 5 of the AMD64 Architecture Programmer’s Manual Volume 2 for more information on PCIDs.)

The register operand is always 64 bits in 64-bit mode and 32 bits outside 64-bit mode regardless of value of CS.D.

The operation type is specified in the register operand bits [1:0]. The operation type determines which components of the address (VA, PCID) are valid for comparison in the TLB and whether to include global valid bits in the invalidation process.

The operation types are:

reg32/64 [1:0]	Action
0	Invalidate TLB entries that match {PCID, VA} excluding Global
1	Invalidate all TLB entries that match {PCID} excluding Global
2	Invalidate all TLB entries including Global
3	Invalidate all TLB entries excluding Global

The descriptor in the memory operand is formatted as follows:

127:64	63:12	11:0
VA	Reserved, MBZ	PCID

This instruction invalidates the TLB entry or entries, regardless of the page size (4 Kbytes, 2 Mbytes, 4 Mbytes, or 1 Gbyte). It may invalidate any number of additional TLB entries, in addition to the targeted entry or entries to accomplish the specified function. INVPCID follows the same rules for cached upper TLB entries as INVLPG which is controlled by EFER.TCE. (See Section 3, “Translation Cache Extension” in AMD64 Architecture Programmer’s Manual Volume 2 for more information on EFER.TCE.)

If PCID is disabled (CR4.PCID = 0), all TLB entries are being cached with PCID = 0. When CR4.PCID = 0, executing INVPCID with type 0 and 1 is only allowed if the PCID specified in the descriptor is zero. Furthermore, when CR4.PCID = 0, executing INVPCID with type 2 or 3 invalidate mappings only for PCID = 0.

INVPCID is a serializing instruction and a privileged instruction. The current privilege level must be 0 to execute this instruction.

Mnemonic	Opcode	Description
INVPCID reg32, mem128	66 0F 38 82 /r	Invalidates the TLB entry(s) by PCID in r32 and descriptor in mem28.
INVPCID reg64, mem128	66 0F 38 82 /r	Invalidates the TLB entry(s) by PCID in r64 and descriptor in mem28.

Related Instructions

INVLPG, INVLPGA, INVLPGB, TLBSYNC

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
			X	This instruction not supported as indicated by CPUID Fn0000_0007_EBX_x0[INVPCID] = 0.
			X	If mod=11 (register is specified instead of memory for desc).
			X	If the LOCK prefix is used.
General protection, #GP			X	CPL was not 0.
			X	An invalid type (>3) was specified in register operand.
			X	Bits 63:12 of descriptor in memory operand are not all zero.
			X	Invalidation type 0 was specified and the virtual address in bits 127:64 of descriptor is not canonical.
			X	Invalidation type 0 or 1 and bits 11:0 of descriptor are not zero when CR4.PCIDE = 0.
			X	An execute-only code segment was used to reference memory.
			X	A memory address exceeded a data segment limit.
			X	In long mode, the address of the memory operand was non-canonical.
			X	A null data segment was used to reference memory.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
Page Fault, #PF			X	A page fault resulted from the execution of the instruction.

IRET
IRETD
IRETQ
Return from Interrupt

Returns program control from an exception or interrupt handler to a program or procedure previously interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions also perform a return from a nested task. All flags, CS, and rIP are restored to the values they had before the interrupt so that execution may continue at the next instruction following the interrupt or exception. In 64-bit mode or if the CPL changes, SS and RSP are also restored.

IRET, IRETD, and IRETQ are synonyms mapping to the same opcode. They are intended to provide semantically distinct forms for various opcode sizes. The IRET instruction is used for 16-bit operand size; IRETD is used for 32-bit operand sizes; IRETQ is used for 64-bit operands. The latter form is only meaningful in 64-bit mode.

IRET, IRETD, or IRETQ must be used to terminate the exception or interrupt handler associated with the exception, external interrupt, or software-generated interrupt.

IRET*x* is a serializing instruction.

For detailed descriptions of the steps performed by IRET*x* instructions, see the following:

- *Legacy-Mode Interrupts:* “Legacy Protected-Mode Interrupt Control Transfers” in Volume 2.
- *Long-Mode Interrupts:* “Long-Mode Interrupt Control Transfers” in Volume 2.

Mnemonic	Opcode	Description
IRET	CF	Return from interrupt (16-bit operand size).
IRETD	CF	Return from interrupt (32-bit operand size).
IRETQ	CF	Return from interrupt (64-bit operand size).

Action

```
// For functions READ_DESCRIPTOR, ShadowStacksEnabled
// see "Pseudocode Definition" on page 57

IRET_START:

IF (REAL_MODE)
   IRET_REAL

ELSIF (PROTECTED_MODE)
   IRET_PROTECTED

ELSE // (VIRTUAL_MODE)
   IRET_VIRTUAL
```

```
IRET_REAL:

POP.v temp_RIP
POP.v temp_CS
POP.v temp_RFLAGS

IF (temp_RIP > CS.limit)
   EXCEPTION [#GP(0)]

CS.sel = temp_CS
CS.base = temp_CS SHL 4
RFLAGS.v = temp_RFLAGS // VIF,VIP,VM unchanged
RIP = temp_RIP
EXIT


IRET_PROTECTED:

IF (RFLAGS.NT == 1)
   IF (LEGACY_MODE)        // IRET does a task-switch to a previous task
      TASK_SWITCH          // using the 'back link' field in the TSS
   ELSE // (LONG_MODE)
      EXCEPTION [#GP(0)]   // task switches aren't supported in long mode

POP.v temp_RIP
POP.v temp_CS
POP.v temp_RFLAGS

IF ((temp_RFLAGS.VM==1) && (CPL==0) && (LEGACY_MODE))
   IRET_FROM_PROTECTED_TO_VIRTUAL

IF (temp_CS.rpl = CPL)
      changing_CPL = FALSE
ELSEIF (temp_CS.rpl > CPL)
      changing_CPL = TRUE
ELSE // (temp_CS.rpl < CPL)
      EXCEPTION [#GP(temp_CS)] // IRET to greater priv not allowed

IF ((64BIT_MODE) || (changing_CPL))
   POP.v temp_RSP // in 64-bit mode or changing CPL,  IRET always pops SS:RSP
   POP.v temp_SS

CS = READ_DESCRIPTOR (temp_CS, iret_chk)

IF ((64BIT_MODE) && (temp_RIP is non-canonical) ||
   (!64BIT_MODE) && (temp_RIP > CS.limit))
         EXCEPTION [#GP(0)]

IF (changing_CPL)
   IRET_PROTECTED_TO_OUTER_PRIV
ELSE
```

```
   IRET_PROTECTED_TO_SAME_PRIV


IRET_PROTECTED_TO_OUTER_PRIV:

CPL = CS.rpl

// SS:RSP were popped, so load them into the registers
SS = READ_DESCRIPTOR (temp_SS, ss_chk)
RSP.s = temp_RSP

// pop shadow stack and compare with program stack
IF (ShadowStacksEnabled(old CPL))
   {
   IF (SSP[2:0] != 0)
         EXCEPTION [#CP(RETF/IRET)]  // SSP must be 8-byte aligned
   IF (temp_newCPL != 3)
      {
      temp_sstk_CS  = SSTK_READ_MEM.q [SSP + 16] // read CS from sstk
      temp_sstk_LIP = SSTK_READ_MEM.q [SSP + 8]  // read LIP
      temp_SSP      = SSTK_READ_MEM.q [SSP]      // read previous SSP
      SSP = SSP +24
      IF (temp_CS != temp_sstk_CS)
            EXCEPTION [#CP(RETF/IRET)]      // CS mismatch
      IF ((CS.base + RIP) != temp_sstk_LIP)
            EXCEPTION [#CP(RETF/IRET)]      // LIP mismatch
      IF (temp_SSP[1:0] != 0)
            EXCEPTION [#CP(RETF/IRET)]      // prevSSP must be 4-byte aligned
      }
   }

temp_oldSSP = SSP

IF (ShadowStacksEnabled(new CPL))
   IF (new CPL == 3)
      temp_SSP = PL3_SSP
   IF ((COMPATIBILITY_MODE) && (temp_SSP[63:32] != 0))
         EXCEPTION [#GP(0)] // SSP must be <4GB in compat mode
   SSP = temp_SSP

IF (ShadowStacksEnabled(old CPL)) // check shadow stack token, clear busy
   {
   bool invalid_token = FALSE
   < start atomic section >
   temp_Token= SSTK_READ_MEM.q [temp_oldSSP] // read supervisor sstk token
   IF ((temp_Token AND 0x01) != 1)
      invalid_Token = TRUE                   // token busy bit must be 1
   IF ((temp_Token AND ~0x01) != temp_oldSSP)
      invalid_Token = TRUE                   // address in token must=oldSSP
   IF (!invalid_Token)
      temp_Token = temp_Token AND ~0x01      // clear token busy, if valid
```

```
   SSTK_WRITE_MEM.q [temp_oldSSP] = temp_Token  // writeback token
   < end atomic section >
   } // end shadow stacks enabled at old CPL

FOR (seg = ES, DS, FS, GS)
   IF ((seg.sel == NULL) || ((seg.attr.dpl < CPL) &&
       ((seg.attr.type == 'data') ||
        (seg.attr.type == 'non-conforming-code'))))
      seg = NULL // can't use lower DPL data segment at higher CPL
                 // also clears RPL of any null selectors

RFLAGS.v = temp_RFLAGS  // VIF,VIP,IOPL only changed if old_CPL == 0
                        // IF only changed if old_CPL <= old_RFLAGS.IOPL
                        // VM unchanged
                        // RF cleared
RIP = temp_RIP
EXIT   // end IRET_PROTECTED_TO_OUTER_PRIV


IRET_PROTECTED_TO_SAME_PRIV:

IF (started in 64-bit mode)
   { // in Long Mode SS:RSP were popped, so load them into the registers
   SS = READ_DESCRIPTOR (temp_SS, ss_chk)
   RSP.s = temp_RSP
   }

IF (ShadowStacksEnabled(current CPL))  // pop the shadow stack
   {                                   // and compare with program stack
   IF (SSP[2:0] != 0)
         EXCEPTION [#CP(RETF/IRET)]     // SSP must be 8-byte aligned
   temp_sstk_CS  = SSTK_READ_MEM.q [SSP + 16] // read CS from sstk
   temp_sstk_LIP = SSTK_READ_MEM.q [SSP + 8]  // read LIP
   temp_SSP      = SSTK_READ_MEM.q [SSP]      // read previous SSP
   SSP = SSP +24
   IF (temp_CS != temp_sstk_CS)
        EXCEPTION [#CP(RETF/IRET)]      // CS mismatch
   IF ((CS.base + RIP) != temp_sstk_LIP)
         EXCEPTION [#CP(RETF/IRET)]     // LIP mismatch
   IF (temp_SSP[1:0] != 0)
         EXCEPTION [#CP(RETF/IRET)]     // prevSSP must be 4-byte aligned
   IF ((COMPATIBILITY_MODE) && (tmp_sstk_prevSSP[63:32] != 0))
         EXCEPTION [#GP(0)] // prevSSP must be <4GB in compat mode
   } // end shadow stack enabled at current CPL

// check shadow stack token, clear busy
IF ((ShadowStacksEnabled(currentCPL)) && (LONG_MODE))
   {
   bool invalid_token = FALSE
   < start atomic section >
   temp_Token= SSTK_READ_MEM.q [temp_oldSSP] // read supervisor sstk token
```

```
   IF ((temp_Token AND 0x01) != 1)
      invalid_Token = TRUE                    // token busy bit must be 1
   IF ((temp_Token AND ~x01) != temp_oldSSP)
      invalid_Token = TRUE                    // address in token must=oldSSP
   IF temp_SSP = SSP
      to_same_sstk = TRUE                     // switch was to same sstk
   IF ((!invalid_Token) AND (!to_same_sstk))
      temp_Token = temp_Token AND ~0x01       // clear token busy, if valid
   SSTK_WRITE_MEM.q [temp_oldSSP] = temp_Token  // writeback token
   < end atomic section >
   } // end shadow stacks enabled at CPL and in Long Mode

RFLAGS.v = temp_RFLAGS  // VIF,VIP,IOPL only changed if old_CPL == 0
                        // IF only changed if old_CPL <= old_RFLAGS.IOPL
                        // VM unchanged
                        // RF cleared
RIP = temp_RIP
EXIT    // end IRET_PROTECTED_TO_SAME_PRIV


IRET_VIRTUAL:

IF ((RFLAGS.IOPL < 3) && (CR4.VME == 0))
   EXCEPTION [#GP(0)]

POP.v temp_RIP
POP.v temp_CS
POP.v temp_RFLAGS

IF (temp_RIP > CS.limit)
   EXCEPTION [#GP(0)]

IF (RFLAGS.IOPL == 3)
   {
   RFLAGS.v = temp_RFLAGS  // VIF,VIP,VM,IOPL unchanged, RF cleared
   CS.sel = temp_CS
   CS.base = temp_CS SHL 4

   RIP = temp_RIP
   EXIT
   }

// (IOPL < 3) && (CR4.VME == 1)
ELSEIF ((OPERAND_SIZE == 16) &&
        ((temp_RFLAGS.IF == 0) || (RFLAGS.VIP == 0)) &&
        (temp_RFLAGS.TF == 0))
   {
   RFLAGS.w = temp_RFLAGS  // RFLAGS.VIF = temp_RFLAGS.IF
                           // IF unchanged, RF cleared
   CS.sel = temp_CS
   CS.base = temp_CS SHL 4
```

```
   RIP = temp_RIP
   EXIT
   }

ELSE
   // ((RFLAGS.IOPL < 3) && (CR4.VME == 1) && ((OPERAND_SIZE == 32) ||
   // ((temp_RFLAGS.IF == 1) && (RFLAGS.VIP == 1)) ||
   // (temp_RFLAGS.TF == 1)))
   EXCEPTION [#GP(0)]


IRET_FROM_PROTECTED_TO_VIRTUAL:

// temp_RIP already popped
// temp_CS already popped
// temp_RFLAGS already popped, temp_RFLAGS.VM = 1
// and CPL = 0

POP.d temp_RSP
POP.d temp_SS
POP.d temp_ES
POP.d temp_DS
POP.d temp_FS
POP.d temp_GS

// force the segments to have virtual-mode values
FOR (seg = CS, SS, ES, DS, FS, GS)
   {
   seg.sel       = temp_seg
   seg.base      = temp_seg SHL 4
   seg.limit     = 0x0000FFFF
   IF (seg == CS)
      CS.attr  = 16-bit dpl3 code
   ELSEIF (seg == SS)
      SS.attr  = 16-bit dpl3 stack
   ELSE
      seg.attr = 16-bit dpl3 data
   }

RSP.d    = temp_RSP
RFLAGS.d = temp_RFLAGS
CPL      = 3

temp_oldSSP = SSP

IF (ShadowStacksEnabled(old CPL))    // old CPL is 0 at this point
   {                                 // check shadow stack token, clear busy
   bool invalid_token = FALSE
   < start atomic section >
   temp_Token= SSTK_READ_MEM.q [temp_oldSSP] // read supervisor sstk token
   IF ((temp_Token AND 0x01) != 1)
```

```
      invalid_Token = TRUE                    // token busy bit must be 1
   IF ((temp_Token AND ~0x01) != temp_oldSSP)
      invalid_Token = TRUE                    // address in token must = oldSSP
   IF (!invalid_Token)
      temp_Token = temp_Token AND ~0x01       // clear token busy, if valid
   SSTK_WRITE_MEM.q [temp_oldSSP] = temp_Token  // writeback token
   < end atomic section >
   }    // end shadow stacks enabled at old CPL

RIP = temp_RIP AND 0x0000FFFF
EXIT // end IRET FROM PROTECTED TO VIRTUAL
```

Related Instructions

INT, INTO, INT3

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Segment not present, #NP (selector)			X	The return code segment was marked not present.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
Stack, #SS (selector)			X	The SS register was loaded with a non-null segment selector and the segment was marked not present.
General protection, #GP	X	X	X	The target offset exceeded the code segment limit or was non-canonical.
		X		IOPL was less than 3 and one of the following conditions was true: • CR4.VME was 0. • The effective operand size was 32-bit. • Both the original EFLAGS.VIP and the new EFLAGS.IF were set. • The new EFLAGS.TF was set.
			X	IRET*x* was executed in long mode while EFLAGS.NT=1.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP (selector)			X	The return code selector was a null selector.
			X	The return stack selector was a null selector and the return mode was non-64-bit mode or CPL was 3.
			X	The return code or stack descriptor exceeded the descriptor table limit.
			X	The return code or stack selector's TI bit was set but the LDT selector was a null selector.
			X	The segment descriptor for the return code was not a code segment.
			X	The RPL of the return code segment selector was less than the CPL.
			X	The return code segment was non-conforming and the segment selector's DPL was not equal to the RPL of the code segment's segment selector.
			X	The return code segment was conforming and the segment selector's DPL was greater than the RPL of the code segment's segment selector.
			X	The segment descriptor for the return stack was not a writable data segment.
			X	The stack segment descriptor DPL was not equal to the RPL of the return code segment selector.
			X	The stack segment selector RPL was not equal to the RPL of the return code segment selector.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
Control-protection, #CP			X	The return address on the program stack did not match the address on the shadow stack, or the previous SSP is not 4 byte aligned, or the previous SSP was not <4GB when returning to 32-bit mode or compatibility mode.

LAR Load Access Rights Byte

Loads the access rights from the segment descriptor specified by a 16-bit source register or memory operand into a specified 16-bit, 32-bit, or 64-bit general-purpose register and sets the zero (ZF) flag in the rFLAGS register if successful. LAR clears the zero flag if the descriptor is invalid for any reason.

The LAR instruction checks that:

- the segment selector is not a null selector.
- the descriptor is within the GDT or LDT limit.
- the descriptor DPL is greater than or equal to both the CPL and RPL, or the segment is a conforming code segment.
- the descriptor type is valid for the LAR instruction. Valid descriptor types are shown in the following table. LDT and TSS descriptors in 64-bit mode, and call-gate descriptors in long mode, are only valid if bits 12:8 of doubleword +12 are zero.

See Volume 2, Section 6.4 for more information on checking access rights using LAR.

Valid Descriptor Type		Description
Legacy Mode	**Long Mode**	
All	All	All code and data descriptors
1	—	Available 16-bit TSS
2	2	LDT
3	—	Busy 16-bit TSS
4	—	16-bit call gate
5	—	Task gate
9	9	Available 32-bit or 64-bit TSS
B	B	Busy 32-bit or 64-bit TSS
C	C	32-bit or 64-bit call gate

If the segment descriptor passes these checks, the attributes are loaded into the destination general-purpose register. If it does not, then the zero flag is cleared and the destination register is not modified.

When the operand size is 16 bits, access rights include the DPL and Type fields located in bytes 4 and 5 of the descriptor table entry. Before loading the access rights into the destination operand, the low order word is masked with FF00H.

When the operand size is 32 or 64 bits, access rights include the DPL and type as well as the descriptor type (S field), segment present (P flag), available to system (AVL flag), default operation size (D/B

flag), and granularity flags located in bytes 4–7 of the descriptor. Before being loaded into the destination operand, the doubleword is masked with 00FF_FF00H.

In 64-bit mode, for both 32-bit and 64-bit operand sizes, 32-bit register results are zero-extended to 64 bits.

This instruction can only be executed in protected mode.

Mnemonic	Opcode	Description
LAR *reg16*, *reg/mem16*	0F 02 */r*	Reads the GDT/LDT descriptor referenced by the 16-bit source operand, masks the attributes with FF00h and saves the result in the 16-bit destination register.
LAR *reg32*, *reg/mem16*	0F 02 */r*	Reads the GDT/LDT descriptor referenced by the 16-bit source operand, masks the attributes with 00FFFF00h and saves the result in the 32-bit destination register.
LAR *reg64*, *reg/mem16*	0F 02 */r*	Reads the GDT/LDT descriptor referenced by the 16-bit source operand, masks the attributes with 00FFFF00h and saves the result in the 64-bit destination register.

Related Instructions

ARPL, LSL, VERR, VERW

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
													M			
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or zero is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded the data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

LGDT Load Global Descriptor Table Register

Loads the pseudo-descriptor specified by the source operand into the global descriptor table register (GDTR). The pseudo-descriptor is a memory location containing the GDTR base and limit. In legacy and compatibility mode, the pseudo-descriptor is 6 bytes; in 64-bit mode, it is 10 bytes.

If the operand size is 16 bits, the high-order byte of the 6-byte pseudo-descriptor is not used. The lower two bytes specify the 16-bit limit and the third, fourth, and fifth bytes specify the 24-bit base address. The high-order byte of the GDTR is filled with zeros.

If the operand size is 32 bits, the lower two bytes specify the 16-bit limit and the upper four bytes specify a 32-bit base address.

In 64-bit mode, the lower two bytes specify the 16-bit limit and the upper eight bytes specify a 64-bit base address. In 64-bit mode, operand-size prefixes are ignored and the operand size is forced to 64-bits; therefore, the pseudo-descriptor is always 10 bytes.

This instruction is only used in operating system software and must be executed at CPL 0. It is typically executed once in real mode to initialize the processor before switching to protected mode.

LGDT is a serializing instruction.

Mnemonic	Opcode	Description
LGDT *mem16:32*	0F 01 /2	Loads *mem16:32* into the global descriptor table register.
LGDT *mem16:64*	0F 01 /2	Loads *mem16:64* into the global descriptor table register.

Related Instructions

LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The operand was a register.
Stack, #SS	X		X	A memory address exceeded the stack segment limit or was non-canonical.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X		X	A memory address exceeded the data segment limit or was non-canonical.
		X	X	CPL was not 0.
			X	The new GDT base address was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.

LIDT Load Interrupt Descriptor Table Register

Loads the pseudo-descriptor specified by the source operand into the interrupt descriptor table register (IDTR). The pseudo-descriptor is a memory location containing the IDTR base and limit. In legacy and compatibility mode, the pseudo-descriptor is six bytes; in 64-bit mode, it is 10 bytes.

If the operand size is 16 bits, the high-order byte of the 6-byte pseudo-descriptor is not used. The lower two bytes specify the 16-bit limit and the third, fourth, and fifth bytes specify the 24-bit base address. The high-order byte of the IDTR is filled with zeros.

If the operand size is 32 bits, the lower two bytes specify the 16-bit limit and the upper four bytes specify a 32-bit base address.

In 64-bit mode, the lower two bytes specify the 16-bit limit, and the upper eight bytes specify a 64-bit base address. In 64-bit mode, operand-size prefixes are ignored and the operand size is forced to 64-bits; therefore, the pseudo-descriptor is always 10 bytes.

This instruction is only used in operating system software and must be executed at CPL 0. It is normally executed once in real mode to initialize the processor before switching to protected mode.

LIDT is a serializing instruction.

Mnemonic	Opcode	Description
LIDT *mem16:32*	0F 01 /3	Loads *mem16:32* into the interrupt descriptor table register.
LIDT *mem16:64*	0F 01 /3	Loads *mem16:64* into the interrupt descriptor table register.

Related Instructions

LGDT, LLDT, LTR, SGDT, SIDT, SLDT, STR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The operand was a register.
Stack, #SS	X		X	A memory address exceeded the stack segment limit or was non-canonical.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X		X	A memory address exceeded the data segment limit or was non-canonical.
		X	X	CPL was not 0.
			X	The new IDT base address was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.

LLDT Load Local Descriptor Table Register

Loads the specified segment selector into the visible portion of the local descriptor table (LDT). The processor uses the selector to locate the descriptor for the LDT in the global descriptor table. It then loads this descriptor into the hidden portion of the LDTR.

If the source operand is a null selector, the LDTR is marked invalid and all references to descriptors in the LDT will generate a general protection exception (#GP), except for the LAR, VERR, VERW or LSL instructions.

In legacy and compatibility modes, the LDT descriptor is 8 bytes long and contains a 32-bit base address.

In 64-bit mode, the LDT descriptor is 16-bytes long and contains a 64-bit base address. The LDT descriptor type (02h) is redefined in 64-bit mode for use as the 16-byte LDT descriptor.

This instruction must be executed in protected mode. It is only provided for use by operating system software at CPL 0.

LLDT is a serializing instruction.

Mnemonic	Opcode	Description
LLDT *reg/mem16*	0F 00 /2	Load the 16-bit segment selector into the local descriptor table register and load the LDT descriptor from the GDT.

Related Instructions

LGDT, LIDT, LTR, SGDT, SIDT, SLDT, STR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Segment not present, #NP (selector)			X	The LDT descriptor was marked not present.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	CPL was not 0.
			X	A null data segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP (selector)			X	The source selector did not point into the GDT.
			X	The descriptor was beyond the GDT limit.
			X	The descriptor was not an LDT descriptor.
			X	The descriptor's extended attribute bits were not zero in 64-bit mode.
			X	The new LDT base address was non-canonical.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.

LMSW Load Machine Status Word

Loads the lower four bits of the 16-bit register or memory operand into bits 3:0 of the machine status word in register CR0. Only the protection enabled (PE), monitor coprocessor (MP), emulation (EM), and task switched (TS) bits of CR0 are modified. Additionally, LMSW can set CR0.PE, but cannot clear it.

The LMSW instruction can be used only when the current privilege level is 0. It is only provided for compatibility with early processors.

Use the MOV CR0 instruction to load all 32 or 64 bits of CR0.

Mnemonic	Opcode	Description
LMSW *reg/mem16*	0F 01 /6	Load the lower 4 bits of the source into the lower 4 bits of CR0.

Related Instructions

MOV CR*n*, SMSW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X		X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X		X	A memory address exceeded a data segment limit or was non-canonical.
		X	X	CPL was not 0.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.

LSL Load Segment Limit

Loads the segment limit from the segment descriptor specified by a 16-bit source register or memory operand into a specified 16-bit, 32-bit, or 64-bit general-purpose register and sets the zero (ZF) flag in the rFLAGS register if successful. LSL clears the zero flag if the descriptor is invalid for any reason.

In 64-bit mode, for both 32-bit and 64-bit operand sizes, 32-bit register results are zero-extended to 64 bits.

The LSL instruction checks that:

- the segment selector is not a null selector.
- the descriptor is within the GDT or LDT limit.
- the descriptor DPL is greater than or equal to both the CPL and RPL, or the segment is a conforming code segment.
- the descriptor type is valid for the LAR instruction. Valid descriptor types are shown in the following table. LDT and TSS descriptors in 64-bit mode are only valid if bits 12:8 of doubleword +12 are zero, as described in “System Descriptors” in Volume 2.

Valid Descriptor Type		Description
Legacy Mode	**Long Mode**	
—	—	All code and data descriptors
1	—	Available 16-bit TSS
2	2	LDT
3	—	Busy 16-bit TSS
9	9	Available 32-bit or 64-bit TSS
B	B	Busy 32-bit or 64-bit TSS

If the segment selector passes these checks and the segment limit is loaded into the destination general-purpose register, the instruction sets the zero flag of the rFLAGS register to 1. If the selector does not pass the checks, then LSL clears the zero flag to 0 and does not modify the destination.

The instruction calculates the segment limit to 32 bits, taking the 20-bit limit and the granularity bit into account. When the operand size is 16 bits, it truncates the upper 16 bits of the 32-bit adjusted segment limit and loads the lower 16-bits into the target register.

Mnemonic	Opcode	Description
LSL *reg16*, *reg/mem16*	0F 03 */r*	Loads a 16-bit general-purpose register with the segment limit for a selector specified in a 16-bit memory or register operand.

LSL *reg32*, *reg/mem16*	0F 03 */r*	Loads a 32-bit general-purpose register with the segment limit for a selector specified in a 16-bit memory or register operand.
LSL *reg64*, *reg/mem16*	0F 03 */r*	Loads a 64-bit general-purpose register with the segment limit for a selector specified in a 16-bit memory or register operand.

Related Instructions

ARPL, LAR, VERR, VERW

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
													M			
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

LTR Load Task Register

Loads the specified segment selector into the visible portion of the task register (TR). The processor uses the selector to locate the descriptor for the TSS in the global descriptor table. It then loads this descriptor into the hidden portion of TR. The TSS descriptor in the GDT is marked busy, but no task switch is made.

If the source operand is null, a general protection exception (#GP) is generated.

In legacy and compatibility modes, the TSS descriptor is 8 bytes long and contains a 32-bit base address.

In 64-bit mode, the instruction references a 64-bit descriptor to load a 64-bit base address. The TSS type (09H) is redefined in 64-bit mode for use as the 16-byte TSS descriptor.

This instruction must be executed in protected mode when the current privilege level is 0. It is only provided for use by operating system software.

The operand size attribute has no effect on this instruction.

LTR is a serializing instruction.

Mnemonic	Opcode	Description
LTR *reg/mem16*	0F 00 /3	Load the 16-bit segment selector into the task register and load the TSS descriptor from the GDT.

Related Instructions

LGDT, LIDT, LLDT, STR, SGDT, SIDT, SLDT

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Segment not present, #NP (selector)			X	The TSS descriptor was marked not present.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	CPL was not 0.
			X	A null data segment was used to reference memory.
			X	The new TSS selector was a null selector.
General protection, #GP (selector)			X	The source selector did not point into the GDT.
			X	The descriptor was beyond the GDT limit.
			X	The descriptor was not an available TSS descriptor.
			X	The descriptor's extended attribute bits were not zero in 64-bit mode.
			X	The new TSS base address was non-canonical.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.

MONITOR Setup Monitor Address

Establishes a linear address range of memory for hardware to monitor and puts the processor in the monitor event pending state. When in the monitor event pending state, the monitoring hardware detects stores to the specified linear address range and causes the processor to exit the monitor event pending state. The MWAIT instruction uses the state of the monitor hardware.

The address range should be a write-back memory type. Executing MONITOR on an address range for a non-write-back memory type is not guaranteed to cause the processor to enter the monitor event pending state. The size of the linear address range that is established by the MONITOR instruction can be determined by CPUID function 0000_0005h.

The [rAX] register provides the effective address. The DS segment is the default segment used to create the linear address. Segment overrides may be used with the MONITOR instruction.

The ECX register specifies optional extensions for the MONITOR instruction. There are currently no extensions defined and setting any bits in ECX will result in a #GP exception. The ECX register operand is implicitly 32-bits.

The EDX register specifies optional hints for the MONITOR instruction. There are currently no hints defined and EDX is ignored by the processor. The EDX register operand is implicitly 32-bits.

The MONITOR instruction can be executed at CPL 0 and is allowed at CPL > 0 only if MSR C001_0015h[MonMwaitUserEn] = 1. When MSR C001_0015h[MonMwaitUserEn] = 0, MONITOR generates #UD at CPL > 0. (See the *BIOS and Kernel Developer's Guide* applicable to your product for specific details on MSR C001_0015h.)

MONITOR performs the same segmentation and paging checks as a 1-byte read.

Support for the MONITOR instruction is indicated by CPUID Fn0000_0001_ECX[MONITOR] = 1. Software must check the CPUID bit once per program or library initialization before using the MONITOR instruction, or inconsistent behavior may result. Software designed to run at CPL greater than 0 must also check for availability by testing whether executing MONITOR causes a #UD exception.

The following pseudo-code shows typical usage of a MONITOR/MWAIT pair:

```
EAX = Linear_Address_to_Monitor;
ECX = 0;  // Extensions
EDX = 0;  // Hints

while (!matching_store_done){
      MONITOR EAX, ECX, EDX
      IF (!matching_store_done) {
           MWAIT EAX, ECX
  }
}
```

Mnemonic	Opcode	Description
MONITOR	0F 01 C8	Establishes a linear address range to be monitored by hardware and activates the monitor hardware.

Related Instructions

MWAIT, MONITORX, MWAITX

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The MONITOR/MWAIT instructions are not supported, as indicated by CPUID Fn0000_0001_ECX[MONITOR] = 0.
		X	X	CPL was not 0 and MSR C001_0015[MonMwaitUserEn] = 0.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
	X	X	X	ECX was non-zero.
			X	A null data segment was used to reference memory.
Page Fault, #PF		X	X	A page fault resulted from the execution of the instruction.

MOV CR*n* Move to/from Control Registers

Moves the contents of a 32-bit or 64-bit general-purpose register to a control register or vice versa.

In 64-bit mode, the operand size is fixed at 64 bits without the need for a REX prefix. In non-64-bit mode, the operand size is fixed at 32 bits and the upper 32 bits of the destination are forced to 0.

CR0 maintains the state of various control bits. CR2 and CR3 are used for page translation. CR4 holds various feature enable bits. CR8 is used to prioritize external interrupts. CR1, CR5, CR6, CR7, and CR9 through CR15 are all reserved and raise an undefined opcode exception (#UD) if referenced.

CR8 can be read and written in 64-bit mode, using a REX prefix. CR8 can be read and written in all modes using a LOCK prefix instead of a REX prefix to specify the additional opcode bit. To verify whether the LOCK prefix can be used in this way, check for support of this feature. CPUID Fn8000_0001_ECX[AltMovCr8] = 1, indicates that this feature is supported.

For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

CR8 can also be read and modified using the task priority register described in “System-Control Registers” in Volume 2.

This instruction is always treated as a register-to-register (MOD = 11) instruction, regardless of the encoding of the MOD field in the MODR/M byte.

MOV CR*n* is a privileged instruction and must always be executed at CPL = 0.

MOV CR*n* is a serializing instruction.

Mnemonic	Opcode	Description
MOV CR*n*, *reg32*	0F 22 /r	Move the contents of a 32-bit register to CR*n*
MOV CR*n*, *reg64*	0F 22 /r	Move the contents of a 64-bit register to CR*n*
MOV *reg32*, CR*n*	0F 20 /r	Move the contents of CR*n* to a 32-bit register.
MOV *reg64*, CR*n*	0F 20 /r	Move the contents of CR*n* to a 64-bit register.
MOV CR8, *reg32*	F0 0F 22/r	Move the contents of a 32-bit register to CR8.
MOV CR8, *reg64*	F0 0F 22/r	Move the contents of a 64-bit register to CR8.
MOV reg32, CR8	F0 0F 20/r	Move the contents of CR8 into a 32-bit register.
MOV reg64, CR8	F0 0F 20/r	Move the contents of CR8 into a 64-bit register.

Related Instructions

CLTS, LMSW, SMSW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid Instruction, #UD	X	X	X	An illegal control register was referenced (CR1, CR5–CR7, CR9–CR15).
	X	X	X	The use of the LOCK prefix to read CR8 is not supported, as indicated by CPUID Fn8000_0001_ECX[AltMovCr8] = 0.
General protection, #GP		X	X	CPL was not 0.
	X		X	An attempt was made to set CR0.PG = 1 and CR0.PE = 0.
	X		X	An attempt was made to set CR0.CD = 0 and CR0.NW = 1.
	X		X	Reserved bits were set in the page-directory pointers table (used in the legacy extended physical addressing mode) and the instruction modified CR0, CR3, or CR4.
	X		X	An attempt was made to write 1 to any reserved bit in CR0, CR3, CR4 or CR8.
	X		X	An attempt was made to set CR0.PG while long mode was enabled (EFER.LME = 1), but paging address extensions were disabled (CR4.PAE = 0).
			X	An attempt was made to clear CR4.PAE while long mode was active (EFER.LMA = 1).
			X	An attempt was made to set CR4.PCIDE=1 when long mode was disabled (EFER.LMA=0).
			X	An attempt was made to set CR4.PCIDE=1 when CR3[11:0] <>0.
			X	An attempt was made to set CR0.PG=0 when CR4.PCIDE=1.

MOV DR*n* Move to/from Debug Registers

Moves the contents of a debug register into a 32-bit or 64-bit general-purpose register or vice versa.

In 64-bit mode, the operand size is fixed at 64 bits without the need for a REX prefix. In non-64-bit mode, the operand size is fixed at 32-bits and the upper 32 bits of the destination are forced to 0.

DR0 through DR3 are linear breakpoint address registers. DR6 is the debug status register and DR7 is the debug control register. DR4 and DR5 are aliased to DR6 and DR7 if CR4.DE = 0, and are reserved if CR4.DE = 1.

DR8 through DR15 are reserved and generate an undefined opcode exception if referenced.

These instructions are privileged and must be executed at CPL 0.

The `MOV DRn,reg32` and `MOV DRn,reg64` instructions are serializing instructions.

The MOV(DR) instruction is always treated as a register-to-register (MOD = 11) instruction, regardless of the encoding of the MOD field in the MODR/M byte.

See “Debug and Performance Resources” in Volume 2 for details.

Mnemonic	Opcode	Description
MOV *reg32*, DR*n*	0F 21 /r	Move the contents of DR*n* to a 32-bit register.
MOV *reg64*, DR*n*	0F 21 /r	Move the contents of DR*n* to a 64-bit register.
MOV DR*n*, *reg32*	0F 23 /r	Move the contents of a 32-bit register to DR*n*.
MOV DR*n*, *reg64*	0F 23 /r	Move the contents of a 64-bit register to DR*n*.

Related Instructions

None

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Debug, #DB	X		X	A debug register was referenced while the general detect (GD) bit in DR7 was set.
Invalid opcode, #UD	X		X	DR4 or DR5 was referenced while the debug extensions (DE) bit in CR4 was set.
			X	An illegal debug register (DR8–DR15) was referenced.
General protection, #GP		X	X	CPL was not 0.
			X	A 1 was written to any of the upper 32 bits of DR6 or DR7 in 64-bit mode.

MWAIT Monitor Wait

Used in conjunction with the MONITOR instruction to cause a processor to wait until a store occurs to a specific linear address range from another processor. The previously executed MONITOR instruction causes the processor to enter the monitor event pending state. The MWAIT instruction may enter an implementation dependent power state until the monitor event pending state is exited. The MWAIT instruction has the same effect on architectural state as the NOP instruction.

Events that cause an exit from the monitor event pending state include:

- A store from another processor matches the address range established by the MONITOR instruction.
- Any unmasked interrupt, including INTR, NMI, SMI, INIT.
- RESET.
- Any far control transfer that occurs between the MONITOR and the MWAIT.

EAX specifies optional hints for the MWAIT instruction. Optimized C-state request is communicated through EAX[7:4]. The processor C-state is EAX[7:4]+1, so to request C0 is to place the value F in EAX[7:4] and to request C1 is to place the value 0 in EAX[7:4]. All other components of EAX should be zero when making the C1 request. Setting a reserved bit in EAX is ignored by the processor. This is implicitly a 32-bit operand.

ECX specifies optional extensions for the MWAIT instruction. The only extension currently defined is ECX bit 0, which allows interrupts to wake MWAIT, even when eFLAGS.IF = 0. Support for this extension is indicated by a feature flage returned by the CPUID instruction. Setting any unsupported bit in ECX results in a #GP exception. This is implicitly a 32-bit operand.

CPUID Function 0000_0005h indicates support for extended features of MONITOR/MWAIT:

- CPUID Fn0000_0005_ECX[EMX] = 1 indicates support for enumeration of MONITOR/MWAIT extensions.
- CPUID Fn0000_0005_ECX[IBE] = 1 indicates that MWAIT can set ECX[0] to allow interrupts to cause an exit from the monitor event pending state even when eFLAGS.IF = 0.

The MWAIT instruction can be executed at CPL 0 and is allowed at CPL > 0 only if MSR C001_0015h[MonMwaitUserEn] =1. When MSR C001_0015h[MonMwaitUserEn] is 0, MWAIT generates #UD at CPL > 0. (See the *BIOS and Kernel Developer's Guide* applicable to your product for specific details on MSR C001_0015h.)

Support for the MWAIT instruction is indicated by CPUID Fn0000_0001_ECX[MONITOR] = 1. Software MUST check the CPUID bit once per program or library initialization before using the MWAIT instruction, or inconsistent behavior may result. Software designed to run at CPL greater than 0 must also check for availability by testing whether executing MWAIT causes a #UD exception.

The use of the MWAIT instruction is contingent upon the satisfaction of the following coding requirements:

- MONITOR must precede the MWAIT and occur in the same loop.
- MWAIT must be conditionally executed only if the awaited store has not already occurred. (This prevents a race condition between the MONITOR instruction arming the monitoring hardware and the store intended to trigger the monitoring hardware.)

The following pseudo-code shows typical usage of a MONITOR/MWAIT pair:

```
EAX = Linear_Address_to_Monitor;
ECX = 0;  // Extensions
EDX = 0;  // Hints

WHILE (!matching_store_done ){
     MONITOR EAX, ECX, EDX
     IF ( !matching_store_done ) {
          MWAIT EAX, ECX
     }
}
```

Mnemonic	Opcode	Description
MWAIT	0F 01 C9	Causes the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.

Related Instructions

MONITOR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The MONITOR/MWAIT instructions are not supported, as indicated by CPUID Fn0000_0001_ECX[MONITOR] = 0.
		X	X	CPL was not 0 and MSRC001_0015[MonMwaitUserEn] = 0.
General protection, #GP	X	X	X	Unsupported extension bits were set in ECX

PSMASH Page Smash

Expands a 2MB-page RMP entry into a corresponding set of contiguous 4KB-page RMP entries. The 2MB page's system physical address is specified in the RAX register.

The new entries inherit the attributes of the original entry. Upon completion, a return code is stored in EAX. rFLAGS bits OF, ZF, AF, PF and SF are set based on this return code.

The PSMASH instruction invalidates all TLB entries in the system that translate to the 2MB page being expanded.

This instruction is intended for hypervisor use. Attempted execution at an ASID other than 0 will result in a FAIL_PERMISSION return code.

This is a privileged instruction. Attempted execution at a privilege level other than CPL0 will result in a #GP(0) exception. In addition, this instruction is only valid in 64-bit mode with SNP enabled; in all other modes a #UD exception will be generated.

Support for this instruction is indicated by CPUID Fn8000_001F_EAX[SNP]=1.

Mnemonic	Opcode	Description
PSMASH	F3 0F 01 FF	Creates 512 4KB RMP entries from a 2MB RMP entry

Action

```
SYSTEM_PA = RAX & ~0x1FFFFF

IF (!64BIT_MODE)                // Instruction only valid in 64-bit mode
   EXCEPTION [#UD]

IF (!SYSCFG.SNP_EN)             // Instruction only valid when SNP is enabled
   EXCEPTION [#UD]

IF (CPL != 0)                   // Instruction only allowed at CPL 0
   EXCEPTION [#GP(0)]

IF (CURRENT_ASID != 0)          // Instruction only allowed at ASID 0
   EAX = FAIL_PERMISSION
   EXIT

RMP_ENTRY_PA = RMP_BASE + 0x4000 + (SYSTEM_PA / 0x1000) * 16

IF (RMP_ENTRY_PA > RMP_END)     // System address must have an RMP entry
   EAX = FAIL_INPUT
   EXIT

temp_RMP = READ_MEM_PA.o [RMP_ENTRY_PA]

IF (temp_RMP.IMMUTABLE || !temp_RMP.ASSIGNED || (temp_RMP.PAGE_SIZE != 2MB))
```

```
  EAX = FAIL_BADADDR
  EXIT

temp_RMP.PAGE_ SIZE = 4KB
WRITE_MEM_PA.o [RMP_ENTRY_PA] = temp_RMP

FOR (I = 1; I < 512, I++)
{
   temp_RMP.GUEST_PA = temp_RMP.GUEST_PA + 0x1000;
   WRITE_MEM_PA.o [RMP_ENTRY_PA + I * 16] = temp_RMP;
}

EAX = SUCCESS
EXIT
```

Return Codes

Value	Name	Description
0	SUCCESS	Successful completion
1	FAIL_INPUT	Illegal input parameters
2	FAIL_PERMISSION	Current ASID not 0
3	FAIL_INUSE	Another processor is modifying the same RMP entry
4	FAIL_BADADDR	The page did not meet smashing criteria

Related Instructions

RMPUPDATE, PVALIDATE, RMPADJUST

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SNP instructions are not supported as indicated by CPUID Fn8000_001F_EAX[SNP] = 0
	X	X	X	This instruction is only recognized in 64-bit mode
			X	SYSCFG[SNP_EN] was not set to 1
General Protection, #GP			X	CPL was not 0

PVALIDATE Page Validate

Validates or rescinds validation of a guest page's RMP entry. The guest virtual address is specified in the register operand rAX. The portion of RAX used to form the address is determined by the effective address size (current execution mode and optional address size prefix). The page size is specified in ECX[0]. The new RMP Validated bit is specified in EDX[0].

The PVALIDATE instruction is used by an SNP-active guest to modify the validation status of a page. The PVALIDATE instruction will attempt to access the provided page and will take a #VMEXIT(NPF) if a nested translation error occurs or the translated address is outside the range of memory covered by the RMP. Assuming no error is detected, the PVALIDATE instruction will store EDX[0] to the Validated bit in the page's RMP entry.

Upon completion, a return code is stored in EAX. rFLAGS bits OF, ZF, AF, PF and SF are set based on this return code. If the instruction completed successfully, the rFLAGS bit CF indicates if the contents of the RMP entry were changed or not.

While this instruction is intended for use in SNP-active guest system software, it is recognized in any operating mode at CPL0. If the PVALIDATE instruction is executed by an SNP-active guest and changes the Validated bit in the RMP entry, upon completion it sets rFLAGS.CF to 0. If the PVALIDATE instruction is executed in a non-SNP-active environment or does not change the Validated bit in the RMP entry, it sets rFLAGS.CF to 1 and otherwise behaves as a NOP instruction.

This is a privileged instruction. Attempted execution at a privilege level other than CPL0 will result in a #GP(0) exception.

PVALIDATE performs the same segmentation and paging checks as a 1-byte read. PVALIDATE does not invalidate TLB caches.

Support for this instruction is indicated by CPUID Fn8000_001F_EAX[SNP]=1.

Mnemonic	Opcode	Description
PVALIDATE	F2 0F 01 FF	Performs guest page validation

Action

```
GUEST_VA = rAX & ~0xFFF
PAGE_SIZE = ECX[0]
VALIDATE_PAGE = EDX[0]

IF (CPL != 0)                // This instruction is only allowed at CPL 0
   EXCEPTION [#GP(0)]

IF (!SNP_ACTIVE)
   rFLAGS.CF = 1             // Set CF to indicate that the RMP was not changed
   EAX = SUCCESS
   EXIT
```

```
IF (CURRENT_VMPL != 0)
   EXCEPTION [#GP(0)]       // This instruction is only allowed at VMPL 0

IF ((PAGE_SIZE == 2MB)  && (GUEST_VA[20:12] != 0))
   EAX = FAIL_INPUT         // Page size is 2MB and page is not 2MB aligned
   EXIT

(SYSTEM_PA, GUEST_PA) = TRANSLATE(GUEST_VA)
RMP_ENTRY_PA = RMP_BASE + 0x4000 + (SYSTEM_PA / 0x1000) * 16

IF (RMP_ENTRY_PA > RMP_END)
   #VMEXIT(NPF)             //Translated system address must have an RMP entry

temp_RMP = READ_MEM_PA.o [RMP_ENTRY_PA]

IF (temp_RMP.IMMUTABLE || !temp_RMP.ASSIGNED ||
   (temp_RMP.GUEST_PA != GUEST_PA) || (temp_RMP.ASID != ASID) ||
   (temp_RMP.PAGE_SIZE != nPT page size) ||
   ((temp_RMP.PAGE_SIZE == 2MB) && (PAGE_SIZE == 4KB)))
   #VMEXIT(NPF)

IF ((RMP_DATA.PAGE_SIZE == 4KB) && (PAGE_SIZE == 2MB))
   EAX = FAIL_SIZEMISMATCH  // 2MB validation backed by 4KB pages
   EXIT

IF (temp_RMP.VALIDATED == VALIDATE_PAGE)
   rFLAGS.CF = 1
ELSE
   rFLAGS.CF = 0

temp_RMP.VALIDATED = VALIDATE_PAGE
WRITE_MEM_PA.o [RMP_ENTRY_PA] = temp_RMP
EAX = SUCCESS
EXIT
```

Return Codes

Value	Name	Description
0	SUCCESS	Successful completion (regardless of whether Validated bit changed state)
1	FAIL_INPUT	Illegal input parameters
6	FAIL_SIZEMISMATCH	Page size mismatch between guest (2M) and RMP entry (4K)

Related Instructions

RMPUPDATE, PSMASH, RMPADJUST

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SNP instructions are not supported as indicated by CPUID Fn8000_001F_EAX[SNP] = 0
General Protection, #GP		X	X	CPL was not 0
	X	X	X	Current VMPL was not zero
Page Fault, #PF		X	X	A page fault resulted from the execution of the instruction
			X	The effective C-bit was a 0 during the guest page table walk

RDMSR Read Model-Specific Register

Loads the contents of a 64-bit model-specific register (MSR) specified in the ECX register into registers EDX:EAX. The EDX register receives the high-order 32 bits and the EAX register receives the low order bits. The RDMSR instruction ignores operand size; ECX always holds the MSR number, and EDX:EAX holds the data. If a model-specific register has fewer than 64 bits, the unimplemented bit positions loaded into the destination registers are undefined.

This instruction must be executed at a privilege level of 0 or a general protection exception (#GP) will be raised. This exception is also generated if a reserved or unimplemented model-specific register is specified in ECX.

Support for the RDMSR instruction is indicated by CPUID Fn0000_0001_EDX[MSR] = 1 OR CPUID Fn8000_0001_EDX[MSR] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

For more information about model-specific registers, see the documentation for various hardware implementations and “Model-Specific Registers (MSRs)” in *Volume 2: System Programming*.

Mnemonic	Opcode	Description
RDMSR	0F 32	Copy MSR specified by ECX into EDX:EAX.

Related Instructions

WRMSR, RDTSC, RDPMC

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The RDMSR instruction is not supported, as indicated by CPUID Fn0000_0001_EDX[MSR] = 0 or CPUID Fn8000_0001_EDX[MSR] = 0.
General protection, #GP		X	X	CPL was not 0.
	X		X	The value in ECX specifies a reserved or unimplemented MSR address.

RDPKRU Read Protection Key Rights

Loads the contents of the 32-bit Protection Key Rights (PKRU) register into RAX[31:0] and clears the upper 32 bits of RAX. RDX is also cleared to 0. The RDPKRU instruction ignores operand size.

This instruction must be executed with ECX=0, otherwise a general protection fault (#GP) is generated. The upper 32 bits of RCX are ignored. Memory protection keys must be enabled (CR4.PKE=1), otherwise executing this instruction generates an invalid opcode fault (#UD).

Software can check that the operating system has enabled memory protection keys (CR4.PKE=1) by testing CPUID Function 0000_0007h_ECX[OSPKE]. (See Section 5, “Protection Key Rights for User Pages” in AMD64 Architecture Programmer’s Manual Volume 2 for more information on memory protection keys.)

RDPKRU can be executed at any privilege level.

Mnemonic	Opcode	Description
RDPKRU	0F 01 EE	Read the PKRU MSR into EAX and clear RDX

Related Instructions

WRPKRU

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	CR4.PKE=0
General protection, #GP			X	ECX was not zero

RDPMC Read Performance-Monitoring Counter

Reads the contents of a 64-bit performance counter and returns it in the registers EDX:EAX. The ECX register is used to specify the index of the performance counter to be read. The EDX register receives the high-order 32 bits and the EAX register receives the low order 32 bits of the counter. The RDPMC instruction ignores operand size; the index and the return values are all 32 bits.

The base architecture supports four core performance counters: PerfCtr0–3. An extension to the architecture increases the number of core performance counters to 6 (PerfCtr0–5). Other extensions add four northbridge performance counters NB_PerfCtr0–3 and four L2 cache performance counters L2I_PerfCtr0–3.

To select the core performance counter to be read, specify the counter index, rather than the performance counter MSR address. To access the northbridge performance counters, specify the index of the counter plus 6. To access the L2 cache performance counters, specify the index of the counter plus 10.

Programs running at any privilege level can read performance monitor counters if the PCE flag in CR4 is set to 1; otherwise this instruction must be executed at a privilege level of 0.

This instruction is not serializing. Therefore, there is no guarantee that all instructions have completed at the time the performance counter is read.

For more information about performance-counter registers, see the documentation for various hardware implementations and “Performance Counters” in Volume 2.

Support for performance counters beyond PerfCtr0–3 is indicated as follows:

- CPUID Fn8000_0001_ECX[PerfCtrExtCore] = 1 indicates support for core performance counters PerfCtr4-5
- CPUID Fn8000_0001_ECX[PerfCtrExtNB] = 1 indicates support for the 4 architecturally defined Northbridge performance counters .
- CPUID Fn8000_0022_EBX[NumDfPmc] > 4 indicates support for additional Northbridge performance counters.
- CPUID Fn8000_0001_ECX[PerfCtrExtLLC] = 1 indicates support for the six L3 performance counters.

For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

Instruction Encoding

Mnemonic	Opcode	Description
RDPMC	0F 33	Copy the performance monitor counter specified by ECX into EDX:EAX.

Related Instructions

RDMSR, WRMSR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General Protection, #GP	X	X	X	The value in ECX specified an unimplemented performance counter number.
		X	X	CPL was not 0 and CR4.PCE = 0.

RDSSP Read Shawdow Stack Pointer

Reads the current Shadow Stack Pointer (SSP) to the specified GPR. The operand size is 64 bits in 64-bit mode when REX.W=1 and is 32 bits in all other cases. RDSSP is treated as a NOP if CR4.CET = 0, or if shadow stacks are not enabled at the current privilege level.

Mnemonic	Opcode	Description
RDSSPD *reg32*	F3 0F 1E /1	Read SSP[31:0] to reg32
RDSSPQ *reg64*	F3 0F 1E /1	Read SSP[63:0] to reg64

Action

```
IF (((CPL==3) && SSTK_USER_ENABLED) || ((CPL!=3) && SSTK_SUPV_ENABLED))

   IF (OPERAND_SIZE == 64)
      reg64 = SSP
   ELSE
      reg32 = SSP[31:0]
EXIT
```

Related Instructions

RDSSP, RSTORSSP

rFLAGS Affected

None

Exceptions

None.

RDTSC Read Time-Stamp Counter

Loads the value of the processor's 64-bit time-stamp counter into registers EDX:EAX.

The time-stamp counter (TSC) is contained in a 64-bit model-specific register (MSR). The processor sets the counter to 0 upon reset and increments the counter every clock cycle. INIT does not modify the TSC.

The high-order 32 bits are loaded into EDX, and the low-order 32 bits are loaded into the EAX register. This instruction ignores operand size.

When the time-stamp disable flag (TSD) in CR4 is set to 1, the RDTSC instruction can only be used at privilege level 0. If the TSD flag is 0, this instruction can be used at any privilege level.

This instruction is not serializing. Therefore, there is no guarantee that all instructions have completed at the time the time-stamp counter is read.

The behavior of the RDTSC instruction is implementation dependent. The TSC counts at a constant rate, but may be affected by power management events (such as frequency changes), depending on the processor implementation. If CPUID Fn8000_0007_EDX[TscInvariant] = 1, then the TSC rate is ensured to be invariant across all P-States, C-States, and stop-grant transitions (such as STPCLK Throttling); therefore, the TSC is suitable for use as a source of time. Consult the *BIOS and Kernel Developer's Guide* applicable to your product for information concerning the effect of power management on the TSC.

Support for the RDTSC instruction is indicated by CPUID Fn0000_0001_EDX[TSC] = 1 OR CPUID Fn8000_0001_EDX[TSC] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

Instruction Encoding

Mnemonic	Opcode	Description
RDTSC	0F 31	Copy the time-stamp counter into EDX:EAX.

Related Instructions

RDTSCP, RDMSR, WRMSR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The RDTSC instruction is not supported, as indicated by CPUID Fn0000_0001_EDX[TSC] = 0 OR CPUID Fn8000_0001_EDX[TSC] = 0.
General protection, #GP		X	X	CPL was not 0 and CR4.TSD = 1.

RDTSCP Read Time-Stamp Counter and Processor ID

Loads the value of the processor's 64-bit time-stamp counter into registers EDX:EAX, and loads the value of TSC_AUX into ECX. This instruction ignores operand size.

The time-stamp counter is contained in a 64-bit model-specific register (MSR). The processor sets the counter to 0 upon reset and increments the counter every clock cycle. INIT does not modify the TSC.

The high-order 32 bits are loaded into EDX, and the low-order 32 bits are loaded into the EAX register.

The TSC_AUX value is contained in the low-order 32 bits of the TSC_AUX register (MSR address C000_0103h). This MSR is initialized by privileged software to any meaningful value, such as a processor ID, that software wants to associate with the returned TSC value.

When the time-stamp disable flag (TSD) in CR4 is set to 1, the RDTSCP instruction can only be used at privilege level 0. If the TSD flag is 0, this instruction can be used at any privilege level.

Unlike the RDTSC instruction, RDTSCP forces all older instructions to retire before reading the time-stamp counter.

The behavior of the RDTSCP instruction is implementation dependent. The TSC counts at a constant rate, but may be affected by power management events (such as frequency changes), depending on the processor implementation. If CPUID Fn8000_0007_EDX[TscInvariant] = 1, then the TSC rate is ensured to be invariant across all P-States, C-States, and stop-grant transitions (such as STPCLK Throttling); therefore, the TSC is suitable for use as a source of time. Consult the *BIOS and Kernel Developer's Guide* applicable to your product for information concerning the effect of power management on the TSC.

Support for the RDTSCP instruction is indicated by CPUID Fn8000_0001_EDX[RDTSCP] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

Instruction Encoding

Mnemonic	Opcode	Description
RDTSCP	0F 01 F9	Copy the time-stamp counter into EDX:EAX and the TSC_AUX register into ECX.

Related Instructions

RDTSC

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The RDTSCP instruction is not supported, as indicated by CPUID Fn8000_0001_EDX[RDTSCP] = 0.
General protection, #GP		X	X	CPL was not 0 and CR4.TSD = 1.

RMPADJUST Adjust RMP Permissions

Modifies RMP permissions for a guest page. The guest virtual address is specified in the RAX register. The page size is specified in RCX[0]. The target VMPL and its permissions are specified in the RDX register as follows:

RDX bits	Field	Description
[63:17]	RESERVED	
[16]	VMSA	Indicates if the page may be used as a VM Save Area page. This bit is ignored whenever the current VMPL is not 0
[15:8]	TARGET_PERM_MASK	Desired permission mask settings
[7:0]	TARGET_VMPL	Target VMPL

The RMPADJUST instruction is used by an SNP-active guest to modify RMP permissions of a lesser-privileged VMPL. The RMPADJUST instruction will attempt to access the specified page and will take a #VMEXIT(NPF) if a nested translation error occurs or the translated address is outside the range of memory covered by the RMP. Assuming no such error is detected, the target VMPL is numerically higher than the current VMPL, and the specified permissions for the target VMPL are not greater than the permissions of the current VMPL, the RMPADJUST instruction will modify the target permission mask in the RMP entry.

Upon completion, a return code is stored in EAX. rFLAGS bits OF, ZF, AF, PF and SF are set based on this return code.

RMPADJUST performs the same segmentation and paging checks as a 1-byte read. RMPADJUST does not invalidate TLB caches.

This is a privileged instruction. Attempted execution at a privilege level other than CPL0 will result in a #GP(0) exception. In addition, this instruction is only valid in 64-bit mode in an SNP-active guest; in all other modes a #UD exception will be generated.

Support for this instruction is indicated by CPUID Fn8000_001F_EAX[SNP]=1.

Mnemonic	Opcode	Description
RMPADJUST	F3 0F 01 FE	Modifies RMP permissions

Action

```
GUEST_VA = RAX & ~0xFFF
PAGE_SIZE = RCX[0]
TARGET_VMPL = RDX[7:0]
TARGET_PERM_MASK = RDX[15:8]
VMSA = RDX[16]

IF (!64BIT_MODE)                    // Instruction only valid in 64-bit mode
   EXCEPTION [#UD]

IF (!SNP_ACTIVE)
```

```
   EXCEPTION [#UD]

IF (CPL != 0)                     // Instruction only allowed at CPL 0
   EXCEPTION [#GP(0)]

IF ((PAGE_SIZE == 2MB) && (GUEST_VA[20:12] != 0))
   EAX = FAIL_INPUT               // Page size is 2MB and not 2MB aligned
   EXIT

IF (TARGET_VMPL <= CURRENT_VMPL)  // Only permissions for numerically-
   EAX = FAIL_PERMISSION          // higher VMPL can be modified
   EXIT

(SYSTEM_PA, GUEST_PA) = TRANSLATE(GUEST_VA)
RMP_ENTRY_PA = RMP_BASE + 0x4000 + (SYSTEM_PA / 0x1000) * 16

IF (RMP_ENTRY_PA > RMP_END)       // Translated system address
   #VMEXIT(NPF)                   // must have an RMP entry

temp_RMP = READ_MEM_PA.o [RMP_ENTRY_PA]

IF (temp_RMP.IMMUTABLE || !temp_RMP.ASSIGNED ||
   (temp_RMP.GUEST_PA != GUEST_PA) || (temp_RMP.ASID != ASID) ||
   (temp_RMP.PAGE_SIZE != nPT page size) ||
   ((temp_RMP.PAGE_SIZE == 2MB) && (PAGE_SIZE == 4KB)))
   #VMEXIT(NPF)

IF (!temp_RMP.VALIDATED)
   #VC(PAGE_NOT_VALIDATED)

IF ((RMP_DATA.PAGE_SIZE == 4KB) && (PAGE_SIZE == 2MB))
   EAX = FAIL_SIZEMISMATCH
   EXIT

IF (TARGET_PERM_MASK & ~temp_RMP.PERMISSIONS[CURRENT_VMPL])
   EAX = FAIL_PERMISSION
   EXIT

IF (CURRENT_VMPL == 0)
   temp_RMP.VMSA = VMSA

temp_RMP.PERMISSIONS[TARGET_VMPL] = TARGET_PERM_MASK

WRITE_MEM_PA.o [RMP_ENTRY_PA] = temp_RMP
EAX = SUCCESS
EXIT
```

Return Codes

Value	Name	Description
0	SUCCESS	Successful completion
1	FAIL_INPUT	Illegal input parameters
2	FAIL_PERMISSION	Insufficient permissions
6	FAIL_SIZEMISMATCH	Page size mismatch between guest and RMP

Related Instructions

PVALIDATE, RMPUPDATE, PSMASH

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SNP instructions are not supported as indicated by CPUID Fn8000_001F_EAX[SNP] = 0
	X	X	X	This instruction is only recognized in 64-bit mode
			X	Guest is not SNP-Active
General Protection, #GP			X	CPL was not 0
Page Fault, #PF			X	A page fault resulted from the execution of the instruction
			X	The effective C-bit was a 0 during the guest page table walk
VMM Communication, #VC			X	RMP.VALIDATED was not set to 1

RMPUPDATE Write RMP Entry

Writes a new RMP entry. The system physical address of a page whose RMP entry is modified is specified in the RAX register. The RCX register provides the effective address of a 16-byte data structure which contains the new RMP state. The DS segment is the default segment used to create the linear address, but may be overridden by a segment prefix. The layout of the data structure with the new RMP state is as follows:

Byte Offset	Length (bytes)	Name	Description
00h	8	GUEST_PA	Guest physical address
08h	1	ASSIGNED	Assigned flag (bit 0)
09h	1	PAGE_SIZE	Page size (0 = 4KB, 1 = 2MB) (bit 0)
0Ah	1	IMMUTABLE	Immutable flag (bit 0)
0Bh	1	-	Reserved (SBZ)
0Ch	4	ASID	ASID of intended page owner

The RMPUPDATE instruction checks that new RMP state is legal before it updates the RMP table.

Upon completion, a return code is stored in EAX. rFLAGS bits OF, ZF, AF, PF and SF are set based on this return code.

The RMPUPDATE instruction invalidates all TLB entries in the system that translate to the page being modified.

This instruction is intended for hypervisor use. Attempted execution at an ASID other than 0 will result in a FAIL_PERMISSION return code.

This is a privileged instruction. Attempted execution at a privilege level other than CPL0 will result in a #GP(0) exception. In addition, this instruction is only valid in 64-bit mode with SNP enabled; in all other modes a #UD exception will be generated.

Support for this instruction is indicated by the feature flag CPUID Fn8000_001F_EAX[SNP]=1.

Mnemonic	Opcode	Description
RMPUPDATE	F2 0F 01 FE	Writes a new RMP entry

Action

```
SYSTEM_PA = RAX & ~0xFFF
NEW_RMP_PTR = RCX

IF (!64BIT_MODE)                    // Instruction only valid in 64-bit mode
   EXCEPTION [#UD]

IF (!SYSCFG.SNP_EN)                 // Instruction only valid when SNP enabled
   EXCEPTION [#UD]
```

```
IF (CPL != 0)                       // Instruction only allowed at CPL 0
   EXCEPTION [#GP(0)]

IF (CURRENT_ASID != 0)              // Instruction only allowed at ASID 0
   EAX = FAIL_PERMISSION
   EXIT

NEW_RMP = READ_MEM.o [NEW_RMP_PTR]

IF ((NEW_RMP.PAGE_SIZE == 2MB) && (SYSTEM_PA[20:12] != 0))
   EAX = FAIL_INPUT
   EXIT

IF (!NEW_RMP.ASSIGNED && (NEW_RMP.IMMUTABLE || (NEW_RMP.ASID != 0))
   EAX = FAIL_INPUT
   EXIT

RMP_ENTRY_PA = RMP_BASE + 0x4000 + (SYSTEM_PA / 0x1000) * 16

IF (RMP_ENTRY_PA > RMP_END)         // System address must have an RMP entry
   EAX = FAIL_INPUT
   EXIT

OLD_RMP = READ_MEM_PA.o [RMP_ENTRY_PA]

IF (OLD_RMP.IMMUTABLE)
   EAX = FAIL_PERMISSION
   EXIT

IF (NEW_RMP.PAGE_SIZE == 4KB)
   IF ((SYSTEM_PA[20:12] == 0) && (OLD_RMP.PAGE_SIZE == 2MB))
      EAX = FAIL_OVERLAP
      EXIT
ELSE
   IF (Any 4KB RMP entry with (RMP.ASSIGNED == 1) exists in 2MB region)
      EAX = FAIL_OVERLAP
      EXIT
   ELSE
      FOR (I = 1; I < 512, I++)
      {
         temp_RMP = 0
         temp_RMP.ASSIGNED = NEW_RMP.ASSIGNED
         WRITE_MEM.o [RMP_ENTRY_PA + I * 16] = temp_RMP;
      }

IF (!NEW_RMP.ASSIGNED)
   temp_RMP = 0
ELSE
   temp_RMP.ASID = NEW_RMP.ASID
   temp_RMP.GUEST_PA = NEW_RMP.GUEST_PA
   temp_RMP.PAGE_SIZE = NEW_RMP.PAGE_SIZE
```

```
      temp_RMP.ASSIGNED = NEW_RMP.ASSIGNED
      temp_RMP.IMMUTABLE = NEW_RMP.IMMUTABLE

      temp_RMP.VALIDATED = OLD_RMP.VALIDATED
      temp_RMP.PERMISSIONS = OLD_RMP.PERMISSIONS
      temp_RMP.VMSA = OLD_RMP.VMSA

      IF (NEW_RMP.ASID == 0)
         temp_RMP.GUEST_PA = 0

      IF ((OLD_RMP.ASID ^ NEW_RMP.ASID) ||
         (OLD_RMP.GUEST_PA ^ NEW_RMP.GUEST_PA) ||
         (OLD_RMP.PAGE_SIZE ^ NEW_RMP.PAGE_SIZE) ||
         (OLD_RMP.ASSIGNED ^ NEW_RMP.ASSIGNED))
         N = CPUID Fn8000001F_EBX[15:12]
         temp_RMP.VALIDATED = 0
         temp_RMP.VMSA = 0
         temp_RMP.PERMISSIONS[0] = 0xF
         temp_RMP.PERMISSIONS[1:(N-1)] = 0

   WRITE_MEM_PA.o [RMP_ENTRY_PA] = temp_RMP
   EAX = SUCCESS
   EXIT
```

Return Codes

Value	Name	Description
0	SUCCESS	Successful completion
1	FAIL_INPUT	Illegal input parameters
2	FAIL_PERMISSION	Current ASID not 0 or RMP entry is Immutable
3	FAIL_INUSE	Another processor is modifying the same RMP entry
4	FAIL_OVERLAP	4KB page and 2MB page RMP overlap detected

Related Instructions

PVALIDATE, PSMASH, RMPADJUST

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SNP instructions are not supported as indicated by CPUID Fn8000_001F_EAX[SNP] = 0
	X	X	X	This instruction is only recognized in 64-bit mode
			X	SYSCFG[SNP_EN] was not set to 1
General Protection, #GP			X	CPL was not 0
			X	A null data segment was used to reference memory

RSM Resume from System Management Mode

Resumes an operating system or application procedure previously interrupted by a system management interrupt (SMI). The processor state is restored from the information saved when the SMI was taken. The processor goes into a shutdown state if it detects invalid state information in the system management mode (SMM) save area during RSM.

RSM will shut down if any of the following conditions are found in the save map (SSM):

- An illegal combination of flags in CR0 (CR0.PG = 1 and CR0.PE = 0, or CR0.NW = 1 and CR0.CD = 0).
- A reserved bit in CR3, CR4, or the extended feature enable register (EFER) is set to 1.
- A reserved bit in the range 63:32 of CR0, DR6, or DR7 is set to 1.
- The following bit combination occurs: EFER.LME = 1, CR0.PG = 1, CR4.PAE = 0.
- The following bit combination occurs: EFER.LME = 1, CR0.PG = 1, CR4.PAE = 1, CS.D = 1, CS.L = 1.
- SMM revision field has been modified.
- The following bit combination occurs: CR4.PCIDE=1 and EFER.LMA=0.

RSM cannot modify EFER.SVME. Attempts to do so are ignored.

When EFER.SVME is 1, RSM reloads the four PDPEs (through the incoming CR3) when returning to a mode that has legacy PAE mode paging enabled.

When EFER.SVME is 1, the RSM instruction is permitted to return to paged real mode (i.e., CR0.PE=0 and CR0.PG=1).

The AMD64 architecture uses a new 64-bit SMM state-save memory image. This 64-bit save-state map is used in all modes, regardless of mode. See “System-Management Mode” in Volume 2 for details.

Mnemonic	Opcode	Description
RSM	0F AA	Resume operation of an interrupted program.

Related Instructions

None

rFLAGS Affected

All flags are restored from the state-save map (SSM).

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The processor was not in System Management Mode (SMM).

RSTORSSP Restore Saved Shadow Stack Pointer

Restores SSP using the shadow stack restore token pointed to by the memory operand. If the token validation checks pass, SSP is set to the linear address of the memory operand and the restore token is replaced with a previous SSP token.

If a return to the previous shadow stack is required, the SAVEPREVSSP instruction can be used to save the previous SSP token to the previous stack. Otherwise, the INCSSP instruction can be used to pop the unneeded previous SSP token from the shadow stack.

If the restored SSP is 4-byte aligned and not 8-byte aligned, CF is set to 1 indicating an alignment hole. The INCSSP instruction can be used to increment SSP past the alignment hole.

Mnemonic	Opcode	Description
RSTORSSP *mem64*	F3 0F 01 /5	Restore SSP and create previous SSP token.

Action

```
// see "Pseudocode Definition" on page 57

IF ((CPL == 3) && (!SSTK_USER_ENABLED))
      EXCEPTION [#UD]

IF ((CPL < 3) && (!SSTK_SUPV_ENABLED))
      EXCEPTION [#UD]

temp_linAdr = Linear_Address(mem64)
IF (temp_linAdr is not 8-byte aligned)
      EXCEPTION [#GP(0)]

bool INVALID_TOKEN = FALSE

< start atomic section >

temp_rstorToken = SSTK_READ_MEM.q [mem64] // fetch token, with locked read

IF ((temp_rstorToken AND 0x02) != 0)
   INVALID_TOKEN = TRUE             // token bit 1 must be clear

IF (64BIT_MODE != (temp_rstorToken AND 0x01))
   INVALID_TOKEN = TRUE             // token bit 0 must match current mode

IF (!64-bit mode) && (temp_rstorToken[63:32] != 0))
   INVALID_TOKEN = TRUE             // previous SSP must be <4Gb in
                                    // legacy and compat modes

temp_prevSSP = (temp_rstorToken AND ~0x01) – 8
temp_prevSSP =  temp_prevSSP AND ~0x07
```

```
IF (temp_prevSSP != temp_linAdr)
   INVALID_TOKEN = TRUE       // prev SSP from token must match lin addr

temp_prevSSPtoken = SSP OR 64BIT_MODE OR 0x02  //create the previousSSP token
SSTK_WRITE_MEM.q [mem64] = INVALID_TOKEN ? temp_rstorToken : temp_prevSSPtoken
                                    // write token and unlock
< end atomic section >

IF (INVALID_TOKEN)
      EXCEPTION [#CP(RSTORSSP)]
ELSE
   {
   SSP = temp_linAdr                // SSP = linear address of memory operand
   RFLAGS.ZF,PF,AF,OF,SF = 0
   RFLAGS.CF = (temp_rstorToken AND 0x04) ? 1 : 0; // set CF if SSP in token
                                                   // was 4-byte aligned
   }

EXIT
```

Related Instructions

SAVEPREVSSP

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	0	0	0	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	CR4.CET = 0
			X	Shadow stacks not enabled at current privilege level.
General protection, #GP			X	The linear address was not 8-byte aligned.
			X	A memory address exceeded a data segment limit.
			X	In long mode, the address of the memory operand was non-canonical.
			X	A null data segment was used to reference memory.
			X	A non-writeable data segment was used.
			X	An execute-only code segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Control Protection, #CP			X	The mode bit (bit 0) in the token did not match the current mode.
			X	The type bit (bit 1) in the token was not 0.
			X	The SSP address in the token did not match the linear address of the memory operand.
Page fault, #PF			X	The linear address was not a shadow stack page.
			X	A page fault resulted from the execution of the instruction.

SAVEPREVSSP Save Previous Shadow Stack Pointer

Saves a restore shadow stack token to previous shadow stack. The previous SSP pointer is taken from the previous SSP token found at the top of the current shadow stack. The previous SSP token is then popped from the current shadow stack.

Mnemonic	Opcode	Description
SAVEPREVSSP	F3 0F 01 EA	Push restore shadow stack token to the previous shadow stack

Action

```
// see "Pseudocode Definition" on page 57

IF ((CPL == 3) && (!SSTK_USER_ENABLED))
      EXCEPTION [#UD]

IF ((CPL < 3) && (!SSTK_SUPV_ENABLED))
      EXCEPTION [#UD]

IF (SSP is not 8-byte aligned)
    EXCEPTION [#GP(0)]

temp_prevSSPtoken = SSTK_READ_MEM.q [SSP] // pop prev SSP token
                                          // from current stack
temp_SSP = SSP
temp_SSP = temp_SSP + 8

IF (RFLAGS.CF)         // CF indicates a 4-byte alignment hole exists
   IF (64BIT_MODE)
     EXCEPTION [#GP(0)] // alignment hole allowed only in legacy/compat mode
   ELSE
      {
      hole = SSTK_READ_MEM.d [temp_SSP] // pop the 4-byte alignment hole
      temp_SSP = temp_SSP + 4
      IF (hole != 0)
         EXCEPTION [#GP(0)]       // the alignment hole must be all 0's
      }
IF ((temp_prevSSPtoken AND 0x02) != 1)
    EXCEPTION [#GP(0)]            // prev SSP token must have bit 1 set

IF (64BIT_MODE != (temp_prevSSPtoken AND 0x01))
    EXCEPTION [#GP(0)]            // token bit 0 must match current mode

IF (!64-bit mode) && (temp_prevSSPtoken[63:32] != 0))
    EXCEPTION [#GP(0)]            // previous SSP must be <4Gb in
                                  // legacy and compat modes
```

```
temp_oldSSP = temp_prevSSPtoken AND ~0x03
temp_rstorSSPtoken = temp_oldSSP OR (64BIT_MODE) //create the restore
                                                 SSP token
SSTK_WRITE_MEM.d [temp_oldSSP - 4] = 0x0 // zero out hole (in case aligning
                                         // oldSSP creates a hole)
temp_oldSSP = temp_oldSSP AND ~0x07      // align oldSSP to next 8b boundary
SSTK_WRITE_MEM.q [temp_oldSSP-8]= temp_rstorSSPtoken // write restore token to
                                                     // old stack
SSP = temp_SSP                           // no faults, update SSP
```

Related Instructions

RSTORSSP

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		Instruction is only recognized in protected mode.
			X	CR4.CET = 0
			X	Shadow stacks not enabled at current privilege level.
General protection, #GP			X	The SSP was not 8-byte aligned.
			X	The type bit (bit 1) in the token was not 1.
			X	CF was set in 64-bit mode.
			X	The previous SSP was >4Gb when not in 64-bit mode.
			X	A non-zero alignment hole was found in legacy or compatibility mode.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
			X	A shadow stack reference was made to a non-shadow stack page.

SETSSBSY Set Shadow Stack Busy

Validates a non-busy (not in-use) shadow stack token pointed to by the PL0_SSP MSR and sets the token's busy bit. If the validation checks pass, SSP is set to the address in PL0_SSP.

SETSSBY is a privileged instruction and must be executed with CPL=0, otherwise a #GP exception is generated. If shadow stacks are not enabled at the supervisor level, a #UD exception is generated.

Mnemonic	Opcode	Description
SETSSBSY	F3 0F 01 E8	Validate token and set shadow stack busy bit

Action

```
// see "Pseudocode Definition" on page 57

IF (CR4.CET == 0)
      EXCEPTION [#UD]
IF (S_CET.SH_STK_EN == 0)
      EXCEPTION [#UD]
IF (CPL != 0)
      EXCEPTION [#GP(0)]

temp_newSSP = PL0_SSP

IF (temp_newSSP is not 8-byte aligned)
      EXCEPTION [#GP(0)]

bool FAULT = FALSE

< start atomic section >

temp_Token = SSTK_READ_MEM.q [temp_newSSP] // fetch token with locked read

IF ((!64-bit mode) && (temp_token[63:32] != 0))
   FAULT=TRUE                          // address in token must be < 4GB
                                       // in legacy/compatibility mode
IF ((temp_Token AND 0x01) != 0)
   FAULT = TRUE                        // token busy bit must be 0
IF ((temp_Token AND ~x01) != temp_newSSP)
   FAULT = TRUE                        // address in token must match new SSP
IF (!FAULT)
   temp_Token = temp_Token OR 0x01     // if no faults, set token busy bit

SSTK_WRITE_MEM.q [temp_newSSP] = temp_Token // write token and unlock

< end atomic section >

IF (FAULT)
   EXCEPTION [#CP(SETSSBSY)]
```

```
ELSE
   SSP = temp_newSSP    // if no faults, SSP = PL0_SSP

EXIT
```

Related Instructions

CLRSSBSY

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid Opcode, #UD	X	X	X	Instruction is only recognized in protected mode.
			X	CR4.CET = 0.
			X	Shadow stacks not enabled at supervisor level.
General Protection, #GP			X	CPL ! = 0
			X	PL0_SSP MSR is not 8-byte aligned.
Control, #CP			X	The shadow stack token is busy.
			X	The shadow stack token reserved bits are not 0.
			X	PL0_SSP MSR >4Gb when not in 64-bit mode.
			X	The new SSP in the token != PL0_SSP.
Page Fault, #PF			X	PL0_SSP MSR is not a supervisor shadow stack page.
			X	A page fault resulted from the execution of the instruction.

SGDT Store Global Descriptor Table Register

Stores the global descriptor table register (GDTR) into the destination operand. In legacy and compatibility mode, the destination operand is 6 bytes; in 64-bit mode, it is 10 bytes. In all modes, operand-size prefixes are ignored.

In non-64-bit mode, the lower two bytes of the operand specify the 16-bit limit and the upper 4 bytes specify the 32-bit base address.

In 64-bit mode, the lower two bytes of the operand specify the 16-bit limit and the upper 8 bytes specify the 64-bit base address.

This instruction is intended for use in operating system software, but it can be used at any privilege level.

Mnemonic	Opcode	Description
SGDT *mem16:32*	0F 01 /0	Store global descriptor table register to memory.
SGDT *mem16:64*	0F 01 /0	Store global descriptor table register to memory.

Related Instructions

SIDT, SLDT, STR, LGDT, LIDT, LLDT, LTR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The operand was a register.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

SIDT Store Interrupt Descriptor Table Register

Stores the interrupt descriptor table register (IDTR) in the destination operand. In legacy and compatibility mode, the destination operand is 6 bytes; in 64-bit mode it is 10 bytes. In all modes, operand-size prefixes are ignored.

In non-64-bit mode, the lower two bytes of the operand specify the 16-bit limit and the upper 4 bytes specify the 32-bit base address.

In 64-bit mode, the lower two bytes of the operand specify the 16-bit limit and the upper 8 bytes specify the 64-bit base address.

This instruction is intended for use in operating system software, but it can be used at any privilege level.

Mnemonic	Opcode	Description
SIDT *mem16:32*	0F 01 /1	Store interrupt descriptor table register to memory.
SIDT *mem16:64*	0F 01 /1	Store interrupt descriptor table register to memory.

Related Instructions

SGDT, SLDT, STR, LGDT, LIDT, LLDT, LTR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The operand was a register.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

SKINIT Secure Init and Jump with Attestation

Securely reinitializes the cpu, allowing for the startup of trusted software (such as a VMM). The code to be executed after reinitialization can be verified based on a secure hash comparison. SKINIT takes the physical base address of the SLB as its only input operand, in EAX. The SLB must be structured as described in “Secure Loader Block” on page 499 of the *AMD64 Architecture Programmer’s Manual Volume 2: System Programming*, order# 24593, and is assumed to contain the code for a Secure Loader (SL).

This is a Secure Virtual Machine (SVM) instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000_0001_ECX[SVM] = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 165.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume 2: System Instructions*, order# 24593.

Mnemonic	Opcode	Description
SKINIT EAX	0F 01 DE	Secure initialization and jump, with attestation.

Action

```
IF ((EFER.SVME == 0) && !(CPUID 8000_0001.ECX[SKINIT]) || (!PROTECTED_MODE))

     EXCEPTION [#UD]        // This instruction can only be executed
                            //  in protected mode with SVM enabled.

IF (CPL != 0)               // This instruction is only allowed at CPL 0.
    EXCEPTION [#GP]

Initialize processor state as for an INIT signal
CR0.PE = 1

CS.sel = 0x0008
CS.attr = 32-bit code, read/execute
CS.base = 0
CS.limit = 0xFFFFFFFF

SS.sel = 0x0010
SS.attr = 32-bit stack, read/write, expand up
SS.base = 0
SS.limit = 0xFFFFFFFF

EAX = EAX & 0xFFFF0000 // Form SLB base address.
EDX = family/model/stepping
ESP = EAX + 0x00010000  // Initial SL stack.
Clear GPRs other than EAX, EDX, ESP

EFER = 0
VM_CR.DPD = 1
```

```
VM_CR.R_INIT = 1
VM_CR.DIS_A20M = 1


Enable SL_DEV, to protect 64Kbyte of physical memory starting at
the physical address in EAX

GIF = 0

Read the SL length from offset 0x0002 in the SLB
Copy the SL image to the TPM for attestation

Read the SL entrypoint offset from offset 0x0000 in the SLB
Jump to the SL entrypoint, at EIP = EAX+entrypoint offset
```

Related Instructions

None.

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	Secure Virtual Machine was not enabled (EFER.SVME=0) and both of the following conditions were true: • SVM-Lock is not available, as indicated by CPUID Fn8000_000A_EDX[SVML] = 0. • DEV is not available, as indicated by CPUID Fn8000_0001_ECX[SKINIT] = 0.
	X	X		Instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not 0.

SLDT Store Local Descriptor Table Register

Stores the local descriptor table (LDT) selector to a register or memory destination operand.

If the destination is a register, the selector is zero-extended into a 16-, 32-, or 64-bit general purpose register, depending on operand size.

If the destination operand is a memory location, the segment selector is written to memory as a 16-bit value, regardless of operand size.

This SLDT instruction can only be used in protected mode, but it can be executed at any privilege level.

Mnemonic	Opcode	Description
SLDT *reg16*	0F 00 /0	Store the segment selector from the local descriptor table register to a 16-bit register.
SLDT *reg32*	0F 00 /0	Store the segment selector from the local descriptor table register to a 32-bit register.
SLDT *reg64*	0F 00 /0	Store the segment selector from the local descriptor table register to a 64-bit register.
SLDT *mem16*	0F 00 /0	Store the segment selector from the local descriptor table register to a 16-bit memory location.

Related Instructions

SIDT, SGDT, STR, LIDT, LGDT, LLDT, LTR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

SMSW Store Machine Status Word

Stores the lower bits of the machine status word (CR0). The target can be a 16-, 32-, or 64-bit register or a 16-bit memory operand.

This instruction is provided for compatibility with early processors.

This instruction can be used at any privilege level (CPL).

Mnemonic	Opcode	Description
SMSW *reg16*	0F 01 /4	Store the low 16 bits of CR0 to a 16-bit register.
SMSW *reg32*	0F 01 /4	Store the low 32 bits of CR0 to a 32-bit register.
SMSW *reg64*	0F 01 /4	Store the entire 64-bit CR0 to a 64-bit register.
SMSW *mem16*	0F 01 /4	Store the low 16 bits of CR0 to memory.

Related Instructions

LMSW, MOV CR*n*

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

STAC Set Alignment Check Flag

Sets the Alignment Check flag in the rFLAGS register to one. Support for the STAC instruction is indicated by CPUID Fn07_EBX[20] =1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

Mnemonic	Opcode	Description
STAC	0F 01 CB	Sets the AC flag

Related Instructions

CLAC

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
			1													
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank.Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID
		X		Instruction is not supported in virtual mode
			X	Lock prefix (F0h) preceding opcode.
			X	CPL was not 0

STI Set Interrupt Flag

Sets the interrupt flag (IF) in the rFLAGS register to 1, thereby allowing external interrupts received on the INTR input. Interrupts received on the non-maskable interrupt (NMI) input are not affected by this instruction.

In real mode, this instruction sets IF to 1.

In protected mode and virtual-8086-mode, this instruction is IOPL-sensitive. If the CPL is less than or equal to the rFLAGS.IOPL field, the instruction sets IF to 1.

In protected mode, if IOPL < 3, CPL = 3, and protected mode virtual interrupts are enabled (CR4.PVI = 1), then the instruction instead sets rFLAGS.VIF to 1. If none of these conditions apply, the processor raises a general protection exception (#GP). For more information, see “Protected Mode Virtual Interrupts” in Volume 2.

In virtual-8086 mode, if IOPL < 3 and the virtual-8086-mode extensions are enabled (CR4.VME = 1), the STI instruction instead sets the virtual interrupt flag (rFLAGS.VIF) to 1.

If STI sets the IF flag and IF was initially clear, then interrupts are not enabled until after the instruction following STI. Thus, if IF is 0, this code will not allow an INTR to happen:

```
STI
CLI
```

In the following sequence, INTR will be allowed to happen only after the NOP.

```
STI
NOP
CLI
```

If STI sets the VIF flag and VIP is already set, a #GP fault will be generated.

See “Virtual-8086 Mode Extensions” in Volume 2 for more information about IOPL-sensitive instructions.

Mnemonic	Opcode	Description
STI	FB	Set interrupt flag (IF) to 1.

Action

```
IF (CPL <= IOPL)
      RFLAGS.IF = 1

ELSIF (((VIRTUAL_MODE) && (CR4.VME == 1))
        || ((PROTECTED_MODE) && (CR4.PVI == 1) && (CPL == 3)))
       {
         IF (RFLAGS.VIP == 1)
             EXCEPTION[#GP(0)]
         RFLAGS.VIF = 1
       }
ELSE
       EXCEPTION[#GP(0)]
```

Related Instructions

CLI

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
		M								M						
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. M (modified) is either set to one or cleared to zero. Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X		The CPL was greater than the IOPL and virtual-mode extensions were not enabled (CR4.VME = 0).
			X	The CPL was greater than the IOPL and either the CPL was not 3 or protected-mode virtual interrupts were not enabled (CR4.PVI = 0).
		X	X	This instruction would set RFLAGS.VIF to 1 and RFLAGS.VIP was already 1.

STGI Set Global Interrupt Flag

Sets the global interrupt flag (GIF) to 1. While GIF is zero, all external interrupts are disabled.

This is a Secure Virtual Machine (SVM) instruction.

Attempted execution of this instruction causes a #UD exception if SVM is not enabled and neither SVM Lock nor the device exclusion vector (DEV) are supported. Support for SVM Lock is indicated by CPUID Fn8000_000A_EDX[SVML] = 1. Support for DEV is part of the SKINIT architecture and is indicated by CPUID Fn8000_0001_ECX[SKINIT] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

For information on enabling SVM, see “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume-2: System Instructions*, order# 24593.

Mnemonic	Opcode	Description
STGI	0F 01 DC	Sets the global interrupt flag (GIF).

Related Instructions

CLGI

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD			X	Secure Virtual Machine was not enabled (EFER.SVME=0) and both of the following conditions were true: • SVM Lock is not available, as indicated by CPUID Fn8000_000A_EDX[SVML] = 0. • DEV is not available, as indicated by CPUID Fn8000_0001_ECX[SKINIT] = 0.
	X	X		Instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not 0.

STR Store Task Register

Stores the task register (TR) selector to a register or memory destination operand.

If the destination is a register, the selector is zero-extended into a 16-, 32-, or 64-bit general purpose register, depending on the operand size.

If the destination is a memory location, the segment selector is written to memory as a 16-bit value, regardless of operand size.

The STR instruction can only be used in protected mode, but it can be used at any privilege level.

Mnemonic	Opcode	Description
STR *reg16*	0F 00 /1	Store the segment selector from the task register to a 16-bit general-purpose register.
STR *reg32*	0F 00 /1	Store the segment selector from the task register to a 32-bit general-purpose register.
STR *reg64*	0F 00 /1	Store the segment selector from the task register to a 64-bit general-purpose register.
STR *mem16*	0F 00 /1	Store the segment selector from the task register to a 16-bit memory location.

Related Instructions

LGDT, LIDT, LLDT, LTR, SIDT, SGDT, SLDT

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

SWAPGS Swap GS Register with KernelGSbase MSR

Provides a fast method for system software to load a pointer to system data structures. SWAPGS can be used upon entering system-software routines as a result of a SYSCALL instruction, an interrupt or an exception. Prior to returning to application software, SWAPGS can be used to restore the application data pointer that was replaced by the system data-structure pointer.

This instruction can only be executed in 64-bit mode. Executing SWAPGS in any other mode generates an undefined opcode exception.

The SWAPGS instruction only exchanges the base-address value located in the KernelGSbase model-specific register (MSR address C000_0102h) with the base-address value located in the hidden-portion of the GS selector register (GS.base). This allows the system-kernel software to access kernel data structures by using the GS segment-override prefix during memory references.

The address stored in the KernelGSbase MSR must be in canonical form. The WRMSR instruction used to load the KernelGSbase MSR causes a general-protection exception if the address loaded is not in canonical form. The SWAPGS instruction itself does not perform a canonical check.

This instruction is only valid in 64-bit mode at CPL 0. A general protection exception (#GP) is generated if this instruction is executed at any other privilege level.

For additional information about this instruction, refer to “System Instructions” in Volume 2.

Examples

At a kernel entry point, the OS uses SwapGS to obtain a pointer to kernel data structures and simultaneously save the user's GS base. Upon exit, it uses SwapGS to restore the user's GS base:

```
SystemCallEntryPoint:
SwapGS                          ; get kernel pointer, save user GSbase
mov gs:[SavedUserRSP], rsp     ; save user's stack pointer
mov rsp, gs:[KernelStackPtr]   ; set up kernel stack
push rax                        ; now save user GPRs on kernel stack
               .                ; perform system service
               .
SwapGS                          ; restore user GS, save kernel pointer
```

Mnemonic	Opcode	Description
SWAPGS	0F 01 F8	Exchange GS base with KernelGSBase MSR. (Invalid in legacy and compatibility modes.)

Related Instructions

None

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	This instruction was executed in legacy or compatibility mode.
General protection, #GP			X	CPL was not 0.

SYSCALL Fast System Call

Transfers control to a fixed entry point in an operating system. It is designed for use by system and application software implementing a flat-segment memory model.

The SYSCALL and SYSRET instructions are low-latency system call and return control-transfer instructions, which assume that the operating system implements a flat-segment memory model. By eliminating unneeded checks, and by loading pre-determined values into the CS and SS segment registers (both visible and hidden portions), calls to and returns from the operating system are greatly simplified. These instructions can be used in protected mode and are particularly well-suited for use in 64-bit mode, which requires implementation of a paged, flat-segment memory model.

This instruction has been optimized by reducing the number of checks and memory references that are normally made so that a call or return takes considerably fewer clock cycles than the CALL FAR /RET FAR instruction method.

It is assumed that the base, limit, and attributes of the Code Segment will remain flat for all processes and for the operating system, and that only the current privilege level for the selector of the calling process should be changed from a current privilege level of 3 to a new privilege level of 0. It is also assumed (but not checked) that the RPL of the SYSCALL and SYSRET target selectors are set to 0 and 3, respectively.

SYSCALL sets the CPL to 0, regardless of the values of bits 33:32 of the STAR register. There are no permission checks based on the CPL, real mode, or virtual-8086 mode. SYSCALL and SYSRET must be enabled by setting EFER.SCE to 1.

It is the responsibility of the operating system to keep the descriptors in memory that correspond to the CS and SS selectors loaded by the SYSCALL and SYSRET instructions consistent with the segment base, limit, and attribute values forced by these instructions.

Legacy x86 Mode. In legacy x86 mode, when SYSCALL is executed, the EIP of the instruction following the SYSCALL is copied into the ECX register. Bits 31:0 of the SYSCALL/SYSRET target address register (STAR) are copied into the EIP register. (The STAR register is model-specific register C000_0081h.)

New selectors are loaded, without permission checking (see above), as follows:

- Bits 47:32 of the STAR register specify the selector that is copied into the CS register.
- Bits 47:32 of the STAR register + 8 specify the selector that is copied into the SS register.
- The CS_base and the SS_base are both forced to zero.
- The CS_limit and the SS_limit are both forced to 4 Gbyte.
- The CS segment attributes are set to execute/read 32-bit code with a CPL of zero.
- The SS segment attributes are set to read/write and expand-up with a 32-bit stack referenced by ESP.

Long Mode. When long mode is activated, the behavior of the SYSCALL instruction depends on whether the calling software is in 64-bit mode or compatibility mode. In 64-bit mode, SYSCALL saves the RIP of the instruction following the SYSCALL into RCX and loads the new RIP from LSTAR bits 63:0. (The LSTAR register is model-specific register C000_0082h.) In compatibility mode, SYSCALL saves the RIP of the instruction following the SYSCALL into RCX and loads the new RIP from CSTAR bits 63:0. (The CSTAR register is model-specific register C000_0083h.)

New selectors are loaded, without permission checking (see above), as follows:

- Bits 47:32 of the STAR register specify the selector that is copied into the CS register.
- Bits 47:32 of the STAR register + 8 specify the selector that is copied into the SS register.
- The CS_base and the SS_base are both forced to zero.
- The CS_limit and the SS_limit are both forced to 4 Gbyte.
- The CS segment attributes are set to execute/read 64-bit code with a CPL of zero.
- The SS segment attributes are set to read/write and expand-up with a 64-bit stack referenced by RSP.

The WRMSR instruction loads the target RIP into the LSTAR and CSTAR registers. If an RIP written by WRMSR is not in canonical form, a general-protection exception (#GP) occurs.

How SYSCALL and SYSRET handle rFLAGS, depends on the processor's operating mode.

In legacy mode, SYSCALL treats EFLAGS as follows:

- EFLAGS.IF is cleared to 0.
- EFLAGS.RF is cleared to 0.
- EFLAGS.VM is cleared to 0.

In long mode, SYSCALL treats RFLAGS as follows:

- The current value of RFLAGS is saved in R11.
- RFLAGS is masked using the value stored in SYSCALL_FLAG_MASK.
- RFLAGS.RF is cleared to 0.

For further details on the SYSCALL and SYSRET instructions and their associated MSR registers (STAR, LSTAR, CSTAR, and SYSCALL_FLAG_MASK), see "Fast System Call and Return" in Volume 2.

Support for the SYSCALL instruction is indicated by CPUID Fn8000_0001_EDX[SysCallSysRet] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

Instruction Encoding

Mnemonic	Opcode	Description
SYSCALL	0F 05	Call operating system.

Action

```
// See “Pseudocode Definition” on page 57.

SYSCALL_START:

    IF (MSR_EFER.SCE == 0)          // Check if syscall/sysret are enabled.
        EXCEPTION [#UD]

    IF (LONG_MODE)
        SYSCALL_LONG_MODE
    ELSE // (LEGACY_MODE)
        SYSCALL_LEGACY_MODE


SYSCALL_LONG_MODE:

    RCX.q = next_RIP
    R11.q = RFLAGS    // with rf cleared

    IF (64BIT_MODE)
        temp_RIP.q = MSR_LSTAR
    ELSE // (COMPATIBILITY_MODE)
        temp_RIP.q = MSR_CSTAR

    CS.sel   = MSR_STAR.SYSCALL_CS AND 0xFFFC
    CS.attr  = 64-bit code,dpl0  // Always switch to 64-bit mode in long mode.
    CS.base  = 0x00000000
    CS.limit = 0xFFFFFFFF

    SS.sel   = MSR_STAR.SYSCALL_CS + 8
    SS.attr  = 64-bit stack,dpl0
    SS.base  = 0x00000000
    SS.limit = 0xFFFFFFFF

    RFLAGS = RFLAGS AND ~MSR_SFMASK
    RFLAGS.RF = 0

    IF (ShadowStacksEnabled at current CPL)
       PL3_SSP = SSP

    CPL = 0

    IF (ShadowStacksEnabled at current CPL)
       SSP = 0
```

```
    RIP = temp_RIP
    EXIT


SYSCALL_LEGACY_MODE:

    RCX.d = next_RIP

    temp_RIP.d = MSR_STAR.EIP

    CS.sel   = MSR_STAR.SYSCALL_CS AND 0xFFFC
    CS.attr  = 32-bit code,dpl0  // Always switch to 32-bit mode in legacy mode.
    CS.base  = 0x00000000
    CS.limit = 0xFFFFFFFF

    SS.sel   = MSR_STAR.SYSCALL_CS + 8
    SS.attr  = 32-bit stack,dpl0
    SS.base  = 0x00000000
    SS.limit = 0xFFFFFFFF

    RFLAGS.VM,IF,RF=0

    CPL = 0

    RIP = temp_RIP
    EXIT
```

Related Instructions

SYSRET, SYSENTER, SYSEXIT

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
M	M	M	M	0	0	M	M	M	M	M	M	M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SYSCALL and SYSRET instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[SysCallSysRet] = 0.
	X	X	X	The system call extension bit (SCE) of the extended feature enable register (EFER) is set to 0. (The EFER register is MSR C000_0080h.)

SYSENTER System Call

Transfers control to a fixed entry point in an operating system. It is designed for use by system and application software implementing a flat-segment memory model. This instruction is valid only in legacy mode.

Three model-specific registers (MSRs) are used to specify the target address and stack pointers for the SYSENTER instruction, as well as the CS and SS selectors of the called and returned procedures:

- MSR_SYSENTER_CS: Contains the CS selector of the called procedure. The SS selector is set to MSR_SYSENTER_CS + 8.
- MSR_SYSENTER_ESP: Contains the called procedure’s stack pointer.
- MSR_SYSENTER_EIP: Contains the offset into the CS of the called procedure.

The hidden portions of the CS and SS segment registers are not loaded from the descriptor table as they would be using a legacy x86 CALL instruction. Instead, the hidden portions are forced by the processor to the following values:

- The CS and SS base values are forced to 0.
- The CS and SS limit values are forced to 4 Gbytes.
- The CS segment attributes are set to execute/read 32-bit code with a CPL of zero.
- The SS segment attributes are set to read/write and expand-up with a 32-bit stack referenced by ESP.

System software must create corresponding descriptor-table entries referenced by the new CS and SS selectors that match the values described above.

The return EIP and application stack are not saved by this instruction. System software must explicitly save that information.

An invalid-opcode exception occurs if this instruction is used in long mode. Software should use the SYSCALL (and SYSRET) instructions in long mode. If SYSENTER is used in real mode, a #GP is raised.

For additional information on this instruction, see “SYSENTER and SYSEXIT (Legacy Mode Only)” in Volume 2.

Support for the SYSENTER instruction is indicated by CPUID Fn0000_0001_EDX[SysEnterSysExit] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

Instruction Encoding

Mnemonic	Opcode	Description
SYSENTER	0F 34	Call operating system.

Related Instructions

SYSCALL, SYSEXIT, SYSRET

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
				0						0						
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or zero is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SYSENTER and SYSEXIT instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SysEnterSysExit] = 0.
			X	This instruction is not recognized in long mode.
General protection, #GP	X			This instruction is not recognized in real mode.
		X	X	MSR_SYSENTER_CS was a null selector.

SYSEXIT System Return

Returns from the operating system to an application. It is a low-latency system return instruction designed for use by system and application software implementing a flat-segment memory model.

This is a privileged instruction. The current privilege level must be zero to execute this instruction. An invalid-opcode exception occurs if this instruction is used in long mode. Software should use the SYSRET (and SYSCALL) instructions when running in long mode.

When a system procedure performs a SYSEXIT back to application software, the CS selector is updated to point to the second descriptor entry after the SYSENTER CS value (MSR SYSENTER_CS+16). The SS selector is updated to point to the third descriptor entry after the SYSENTER CS value (MSR SYSENTER_CS+24). The CPL is forced to 3, as are the descriptor privilege levels.

The hidden portions of the CS and SS segment registers are not loaded from the descriptor table as they would be using a legacy x86 RET instruction. Instead, the hidden portions are forced by the processor to the following values:

- The CS and SS base values are forced to 0.
- The CS and SS limit values are forced to 4 Gbytes.
- The CS segment attributes are set to 32-bit read/execute at CPL 3.
- The SS segment attributes are set to read/write and expand-up with a 32-bit stack referenced by ESP.

System software must create corresponding descriptor-table entries referenced by the new CS and SS selectors that match the values described above.

The following additional actions result from executing SYSEXIT:

- EIP is loaded from EDX.
- ESP is loaded from ECX.

System software must explicitly load the return address and application software-stack pointer into the EDX and ECX registers prior to executing SYSEXIT.

For additional information on this instruction, see “SYSENTER and SYSEXIT (Legacy Mode Only)” in Volume 2.

Support for the SYSEXIT instruction is indicated by CPUID Fn0000_0001_EDX[SysEnterSysExit] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

Instruction Encoding

Mnemonic	Opcode	Description
SYSEXIT	0F 35	Return from operating system to application.

Related Instructions

SYSCALL, SYSENTER, SYSRET

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
					0											
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SYSENTER and SYSEXIT instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SysEnterSysExit] = 0.
			X	This instruction is not recognized in long mode.
General protection, #GP	X	X		This instruction is only recognized in protected mode.
			X	CPL was not 0.
			X	MSR_SYSENTER_CS was a null selector.

SYSRET Fast System Return

Returns from the operating system to an application. It is a low-latency system return instruction designed for use by system and application software implementing a flat segmentation memory model.

The SYSCALL and SYSRET instructions are low-latency system call and return control-transfer instructions that assume that the operating system implements a flat-segment memory model. By eliminating unneeded checks, and by loading pre-determined values into the CS and SS segment registers (both visible and hidden portions), calls to and returns from the operating system are greatly simplified. These instructions can be used in protected mode and are particularly well-suited for use in 64-bit mode, which requires implementation of a paged, flat-segment memory model.

This instruction has been optimized by reducing the number of checks and memory references that are normally made so that a call or return takes substantially fewer internal clock cycles when compared to the CALL/RET instruction method.

It is assumed that the base, limit, and attributes of the Code Segment will remain flat for all processes and for the operating system, and that only the current privilege level for the selector of the calling process should be changed from a current privilege level of 0 to a new privilege level of 3. It is also assumed (but not checked) that the RPL of the SYSCALL and SYSRET target selectors are set to 0 and 3, respectively.

SYSRET sets the CPL to 3, regardless of the values of bits 49:48 of the star register. SYSRET can only be executed in protected mode at CPL 0. SYSCALL and SYSRET must be enabled by setting EFER.SCE to 1.

It is the responsibility of the operating system to keep the descriptors in memory that correspond to the CS and SS selectors loaded by the SYSCALL and SYSRET instructions consistent with the segment base, limit, and attribute values forced by these instructions.

When a system procedure performs a SYSRET back to application software, the CS selector is updated from bits 63:50 of the STAR register (STAR.SYSRET_CS) as follows:

- If the return is to 32-bit mode (legacy or compatibility), CS is updated with the value of STAR.SYSRET_CS.
- If the return is to 64-bit mode, CS is updated with the value of STAR.SYSRET_CS + 16.

In both cases, the CPL is forced to 3, effectively ignoring STAR bits 49:48. The SS selector is updated to point to the next descriptor-table entry after the CS descriptor (STAR.SYSRET_CS + 8), and its RPL is not forced to 3.

The hidden portions of the CS and SS segment registers are not loaded from the descriptor table as they would be using a legacy x86 RET instruction. Instead, the hidden portions are forced by the processor to the following values:

- The CS base value is forced to 0.
- The CS limit value is forced to 4 Gbytes.

- The CS segment attributes are set to execute-read 32 bits or 64 bits (see below).
- The SS segment base, limit, and attributes are not modified.

When SYSCALLed system software is running in 64-bit mode, it has been entered from either 64-bit mode or compatibility mode. The corresponding SYSRET needs to know the mode to which it must return. Executing SYSRET in non-64-bit mode or with a 16- or 32-bit operand size returns to 32-bit mode with a 32-bit stack pointer. Executing SYSRET in 64-bit mode with a 64-bit operand size returns to 64-bit mode with a 64-bit stack pointer.

The instruction pointer is updated with the return address based on the operating mode in which SYSRET is executed:

- If returning to 64-bit mode, SYSRET loads RIP with the value of RCX.
- If returning to 32-bit mode, SYSRET loads EIP with the value of ECX.

How SYSRET handles RFLAGS depends on the processor's operating mode:

- If executed in 64-bit mode, SYSRET loads the lower-32 RFLAGS bits from R11[31:0] and clears the upper 32 RFLAGS bits.
- If executed in legacy mode or compatibility mode, SYSRET sets EFLAGS.IF.

For further details on the SYSCALL and SYSRET instructions and their associated MSR registers (STAR, LSTAR, and CSTAR), see "Fast System Call and Return" in Volume 2.

Support for the SYSRET instruction is indicated by CPUID Fn8000_0001_EDX[SysCallSysRet] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

Instruction Encoding

Mnemonic	Opcode	Description
SYSRET	0F 07	Return from operating system.

Action

```
// See "Pseudocode Definition" on page 57.

SYSRET_START:

    IF (MSR_EFER.SCE == 0)          // Check if syscall/sysret are enabled.
        EXCEPTION [#UD]


    IF ((!PROTECTED_MODE) || (CPL != 0))
        EXCEPTION [#GP(0)]              // SYSRET requires protected mode, cpl0

    IF (64BIT_MODE)
        SYSRET_64BIT_MODE
    ELSE // (!64BIT_MODE)
```

```
        SYSRET_NON_64BIT_MODE

SYSRET_64BIT_MODE:

    IF (OPERAND_SIZE == 64)              // Return to 64-bit mode.
    {
        CS.sel   = (MSR_STAR.SYSRET_CS + 16) OR 3
        CS.base  = 0x00000000
        CS.limit = 0xFFFFFFFF
        CS.attr  = 64-bit code,dpl3

        temp_RIP.q = RCX
    }
    ELSE                                 // Return to 32-bit compatibility mode.
    {
        CS.sel   = MSR_STAR.SYSRET_CS OR 3
        CS.base  = 0x00000000
        CS.limit = 0xFFFFFFFF
        CS.attr  = 32-bit code,dpl3

        temp_RIP.d = RCX
    }

    SS.sel = MSR_STAR.SYSRET_CS + 8      // SS selector is changed,
                                         // SS base, limit, attributes unchanged.

    RFLAGS.q = R11       // RF=0,VM=0
    CPL = 3

    IF (ShadowStacksEnabled at current CPL)
        SSP = PL3_SSP

    RIP = temp_RIP
    EXIT

SYSRET_NON_64BIT_MODE:

    CS.sel   = MSR_STAR.SYSRET_CS OR 3 // Return to 32-bit legacy protected mode.
    CS.base  = 0x00000000
    CS.limit = 0xFFFFFFFF
    CS.attr  = 32-bit code,dpl3

    temp_RIP.d = RCX

    SS.sel = MSR_STAR.SYSRET_CS + 8    // SS selector is changed.
                                        // SS base, limit, attributes unchanged.
    RFLAGS.IF = 1
    CPL = 3

    IF (ShadowStacksEnabled at current CPL)
        SSP = PL3_SSP
```

```
RIP = temp_RIP
EXIT
```

Related Instructions

SYSCALL, SYSENTER, SYSEXIT

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
M	M	M	M		0	M	M	M	M	M	M	M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SYSCALL and SYSRET instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[SysCallSysRet] = 0.
	X	X	X	The system call extension bit (SCE) of the extended feature enable register (EFER) is set to 0. (The EFER register is MSR C000_0080h.)
General protection, #GP	X	X		This instruction is only recognized in protected mode.
			X	CPL was not 0.

TLBSYNC Synchronize TLB Invalidations

TLBSYNC acts as a synchronizing instruction to ensure that all logical processors in a system have responded to an INVLPGB previously executed by the current logical processor. Upon execution of an INVLPGB, the processor does not wait for confirmation that the other processors have performed the specified TLB invalidation. A TLBSYNC is therefore required before software can move forward with the knowledge that all requested invalidations have been completed in the system. A TLBSYNC also ensures that memory instructions using the translations invalidated by those prior INVLPGB instructions have retired and writes using the translations have drained from the write combining buffers.

The TLBSYNC instruction is weakly ordered with respect to data and instruction prefetches.

The TLBSYNC instruction is strongly ordered with respect to surrounding loads and stores.

TLBSYNC is a serializing instruction and is privileged. It can only be executed at CPL 0. TLBSYNC is only supported in guests if enabled by hypervisor in the VMCB.

Mnemonic	Opcode	Description
TLBSYNC	0F 01 FF	Synchronize broadcasted TLB Invalidations

Related Instructions

INVLPGB

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported as indicated by CPUID Fn8000_0008_EBX[INVLPGB] = 0
	X	X		Instruction is only recognized in protected mode
			X	The hypervisor has not enabled Guest usage of this instruction.
General protection, #GP			X	CPL was not 0

VERR Verify Segment for Reads

Verifies whether a code or data segment specified by the segment selector in the 16-bit register or memory operand is readable from the current privilege level. The zero flag (ZF) is set to 1 if the specified segment is readable. Otherwise, ZF is cleared.

A segment is readable if all of the following apply:

- the selector is not a null selector.
- the descriptor is within the GDT or LDT limit.
- the segment is a data segment or readable code segment.
- the descriptor DPL is greater than or equal to both the CPL and RPL, or the segment is a conforming code segment.

The processor does not recognize the VERR instruction in real or virtual-8086 mode.

Mnemonic	Opcode	Description
VERR *reg/mem16*	0F 00 /4	Set the zero flag (ZF) to 1 if the segment selected can be read.

Related Instructions

ARPL, LAR, LSL, VERW

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
													M			
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Stack, #SS			X	A memory address exceeded the stack segment limit or is non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

VERW Verify Segment for Write

Verifies whether a data segment specified by the segment selector in the 16-bit register or memory operand is writable from the current privilege level. The zero flag (ZF) is set to 1 if the specified segment is writable. Otherwise, ZF is cleared.

A segment is writable if all of the following apply:

- the selector is not a null selector.
- the descriptor is within the GDT or LDT limit.
- the segment is a writable data segment.
- the descriptor DPL is greater than or equal to both the CPL and RPL.

The processor does not recognize the VERW instruction in real or virtual-8086 mode.

Mnemonic	Opcode	Description
VERW *reg/mem16*	0F 00 /5	Set the zero flag (ZF) to 1 if the segment selected can be written.

Related Instructions

ARPL, LAR, LSL, VERR

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
													M			
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X		This instruction is only recognized in protected mode.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP			X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to access memory.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
Alignment check, #AC			X	An unaligned memory reference was performed while alignment checking was enabled.

VMLOAD Load State from VMCB

Loads a subset of processor state from the VMCB specified by the system-physical address in the rAX register. The portion of RAX used to form the address is determined by the effective address size.

The VMSAVE and VMLOAD instructions complement the state save/restore abilities of VMRUN and #VMEXIT, providing access to hidden state that software is otherwise unable to access, plus some additional commonly-used state.

This is a Secure Virtual Machine (SVM) instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000_0001_ECX[SVM] = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 165.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume 2: System Instructions*, order# 24593.

Mnemonic	Opcode	Description
VMLOAD rAX	0F 01 DA	Load additional state from VMCB.

Action

```
IF ((MSR_EFER.SVME == 0) || (!PROTECTED_MODE))
    EXCEPTION [#UD]       //  This instruction can only be executed in protected
                            //  mode with SVM enabled

IF (CPL != 0)               //  This instruction is only allowed at CPL 0
    EXCEPTION [#GP]

IF (rAX contains an unsupported system-physical address)
    EXCEPTION [#GP]

Load from a VMCB at system-physical address rAX:
      FS, GS, TR, LDTR (including all hidden state)
      KernelGsBase
      STAR, LSTAR, CSTAR, SFMASK
      SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP
```

Related Instructions

VMSAVE

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SVM instructions are not supported as indicated by CPUID Fn8000_0001_ECX[SVM] = 0.
			X	Secure Virtual Machine was not enabled (EFER.SVME=0).
	X	X		The instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not 0.
			X	rAX referenced a physical address above the maximum supported physical address.
			X	The address in rAX was not aligned on a 4Kbyte boundary.

VMMCALL Call VMM
VMGEXIT SEV-ES Exit to VMM

VMMCALL and VMGEXIT provide a mechanism for a non-SEV-ES and an SEV-ES guest, respectively, to explicitly communicate with the VMM by generating a #VMEXIT.

A non-intercepted VMMCALL unconditionally raises a #UD exception. VMGEXIT is always intercepted and unconditionally causes a #VMEXIT.

VMMCALL and VMGEXIT instructions are allowed in all modes and at all privilege levels. These instructions generate a #UD exception if SVM is not enabled. See “Enabling SVM” in AMD64 Architecture Programmer’s Manual Volume 2: System Instructions, order# 24593.

VMMCALL and VMGEXIT are Secure Virtual Machine (SVM) instructions. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000_0001_ECX[SVM] = 1. Support for VMGEXIT instruction is indicated by CPUID Fn8000_001F_EAX[SEV-ES] = 1. The VMGEXIT encoding is interpreted as VMMCALL on processors that do not explicitly support VMGEXIT, including legacy processors, or if VMGEXIT instruction is not executed by an SEV-ES guest. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 165.

Mnemonic	Opcode	Description
VMMCALL	0F 01 D9	Explicit communication with the VMM.
VMGEXIT	F2/F3 0F 01 D9	Explicit communication with the VMM for SEV-ES VMs.

Related Instructions

None.

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SVM instructions are not supported as indicated by CPUID Fn8000_0001_ECX[SVM] = 0.
	X	X	X	Secure Virtual Machine was not enabled (EFER.SVME=0).
	X	X	X	VMMCALL was not intercepted.

VMRUN Run Virtual Machine

Starts execution of a guest instruction stream. The physical address of the *virtual machine control block* (VMCB) describing the guest is taken from the rAX register (the portion of RAX used to form the address is determined by the effective address size). The physical address of the VMCB must be aligned on a 4KB boundary.

VMRUN saves a subset of host processor state to the host state-save area specified by the physical address in the VM_HSAVE_PA MSR. VMRUN then loads guest processor state (and control information) from the VMCB at the physical address specified in rAX. The processor then executes guest instructions until one of several *intercept* events (specified in the VMCB) is triggered. When an intercept event occurs, the processor stores a snapshot of the guest state back into the VMCB, reloads the host state, and continues execution of host code at the instruction following the VMRUN instruction.

This is a Secure Virtual Machine (SVM) instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000_0001_ECX[SVM] = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 165.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume 2: System Instructions*, order# 24593.

The VMRUN instruction is not supported in System Management Mode. Processor behavior resulting from an attempt to execute this instruction from within the SMM handler is undefined.

Instruction Encoding

Mnemonic	Opcode	Description
VMRUN rAX	0F 01 D8	Performs a world-switch to guest.

Action

```
IF ((MSR_EFER.SVME == 0) || (!PROTECTED_MODE))
    EXCEPTION [#UD]       //  This instruction can only be executed in protected
                          //  mode with SVM enabled

IF (CPL != 0)             //  This instruction is only allowed at CPL 0
    EXCEPTION [#GP]

IF (rAX contains an unsupported physical address)
    EXCEPTION [#GP]

IF (intercepted(VMRUN))
    #VMEXIT (VMRUN)
remember VMCB address (delivered in rAX) for next #VMEXIT
save host state to physical memory indicated in the VM_HSAVE_PA MSR:
      ES.sel
      CS.sel
      SS.sel
```

```
      DS.sel
      GDTR.{base,limit}
      IDTR.{base,limit}
      EFER
      CR0
      CR4
      CR3
      // host CR2 is not saved
      RFLAGS
      RIP
      RSP
      RAX

from the VMCB at physical address rAX, load control information:
       intercept vector
       TSC_OFFSET
       interrupt control (v_irq, v_intr_*, v_tpr)
       EVENTINJ field
       ASID

IF(nested paging supported)
    NP_ENABLE
    IF (NP_ENABLE == 1)
       nCR3

from the VMCB at physical address rAX, load guest state:
             ES.{base,limit,attr,sel}
             CS.{base,limit,attr,sel}
             SS.{base,limit,attr,sel}
             DS.{base,limit,attr,sel}
             GDTR.{base,limit}
             IDTR.{base,limit}
             EFER
             CR0
             CR4
             CR3
             CR2
      IF (NP_ENABLE == 1)
            gPAT            //  Leaves host hPAT register unchanged.
             RFLAGS
             RIP
             RSP
             RAX
             DR7
             DR6
             CPL            //  0 for real mode, 3 for v86 mode, else as loaded.
      INTERRUPT_SHADOW

IF (LBR virtualization supported)
    LBR_VIRTUALIZATION_ENABLE
    IF (LBR_VIRTUALIZATION_ENABLE == 1)
```

```
        save LBR state to the host save area
             DBGCTL
             BR_FROM
             BR_TO
             LASTEXCP_FROM
             LASTEXCP_TO
        load LBR state from the VMCB
             DBGCTL
             BR_FROM
             BR_TO
             LASTEXCP_FROM
             LASTEXCP_TO

IF (guest state consistency checks fail)
    #VMEXIT(INVALID)

Execute command stored in TLB_CONTROL.

GIF = 1           // allow interrupts in the guest
IF (EVENTINJ.V)
      cause exception/interrupt in guest
else
      jump to first guest instruction
```

Upon #VMEXIT, the processor performs the following actions in order to return to the host execution context:

```
GIF = 0
save guest state to VMCB:
      ES.{base,limit,attr,sel}
      CS.{base,limit,attr,sel}
      SS.{base,limit,attr,sel}
      DS.{base,limit,attr,sel}
      GDTR.{base,limit}
      IDTR.{base,limit}
      EFER
      CR4
      CR3
      CR2
      CR0
      if (nested paging enabled)
          gPAT
      RFLAGS
      RIP
      RSP
      RAX
      DR7
      DR6
      CPL
      INTERRUPT_SHADOW
save additional state and intercept information:
      V_IRQ, V_TPR
```

```
      EXITCODE
      EXITINFO1
      EXITINFO2
      EXITINTINFO
clear EVENTINJ field in VMCB

prepare for host mode by clearing internal processor state bits:
      clear intercepts
      clear v_irq
      clear v_intr_masking
      clear tsc_offset
      disable nested paging
      clear ASID to zero

reload host state
      GDTR.{base,limit}
      IDTR.{base,limit}
      EFER
      CR0
      CR0.PE = 1  // saved copy of CR0.PE is ignored
      CR4
      CR3
      if (host is in PAE paging mode)
           reloaded host PDPEs
      // Do not reload host CR2 or PAT
      RFLAGS
      RIP
      RSP
      RAX
      DR7 = “all disabled”
      CPL = 0
      ES.sel; reload segment descriptor from GDT
      CS.sel; reload segment descriptor from GDT
      SS.sel; reload segment descriptor from GDT
      DS.sel; reload segment descriptor from GDT

if (LBR virtualization supported)
    LBR_VIRTUALIZATION_ENABLE
    if (LBR_VIRTUALIZATION_ENABLE == 1)
        save LBR state to the VMCB:
            DBGCTL
            BR_FROM
            BR_TO
            LASTEXCP_FROM
            LASTEXCP_TO
        load LBR state from the host save area:
            DBGCTL
            BR_FROM
            BR_TO
            LASTEXCP_FROM
            LASTEXCP_TO
```

```
if (illegal host state loaded, or exception while loading host state)
      shutdown
else
      execute first host instruction following the VMRUN
```

Related Instructions

VMLOAD, VMSAVE.

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SVM instructions are not supported as indicated by CPUID Fn8000_0001_ECX[SVM] = 0.
			X	Secure Virtual Machine was not enabled (EFER.SVME=0).
	X	X		The instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not 0.
			X	rAX referenced a physical address above the maximum supported physical address.
			X	The address in rAX was not aligned on a 4Kbyte boundary.

VMSAVE Save State to VMCB

Stores a subset of the processor state into the VMCB specified by the system-physical address in the rAX register (the portion of RAX used to form the address is determined by the effective address size).

The VMSAVE and VMLOAD instructions complement the state save/restore abilities of VMRUN and #VMEXIT, providing access to hidden state that software is otherwise unable to access, plus some additional commonly-used state.

This is a Secure Virtual Machine (SVM) instruction. Support for the SVM architecture and the SVM instructions is indicated by CPUID Fn8000_0001_ECX[SVM] = 1. For more information on using the CPUID instruction, see the reference page for the CPUID instruction on page 165.

This instruction generates a #UD exception if SVM is not enabled. See “Enabling SVM” in *AMD64 Architecture Programmer’s Manual Volume 2: System Instructions*, order# 24593.

Instruction Encoding

Mnemonic	Opcode	Description
VMSAVE rAX	0F 01 DB	Save additional guest state to VMCB.

Action

```
IF ((MSR_EFER.SVME == 0) || (!PROTECTED_MODE))
     EXCEPTION [#UD]      // This instruction can only be executed in protected
                            //  mode with SVM enabled

IF (CPL != 0)               // This instruction is only allowed at CPL 0
    EXCEPTION [#GP]

IF (rAX contains an unsupported system-physical address)
    EXCEPTION [#GP]

Store to a VMCB at system-physical address rAX:
      FS, GS, TR, LDTR (including all hidden state)
      KernelGsBase
      STAR, LSTAR, CSTAR, SFMASK
      SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP
```

Related Instructions

VMLOAD

rFLAGS Affected

None.

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SVM instructions are not supported as indicated by CPUID Fn8000_0001_ECX[SVM] = 0.
			X	Secure Virtual Machine was not enabled (EFER.SVME=0).
	X	X		The instruction is only recognized in protected mode.
General protection, #GP			X	CPL was not 0.
			X	rAX referenced a physical address above the maximum supported physical address.
			X	The address in rAX was not aligned on a 4Kbyte boundary.

WBINVD Writeback and Invalidate Caches
WBNOINVD Writeback With No Invalidate

WBINVD writes all modified lines in all levels of cache associated with this processor to main memory and invalidates the caches. This may or may not include lower level caches associated with another processor that shares any level of this processor's cache hierarchy. WBNOINVD does not invalidate the caches, instead leaving all (or most) cache lines in the cache hierarchy in non-modified state, but in all other respects it behaves the same as WBINVD.

CPUID Fn8000_001D_EDX[WBINVD]_x*N* indicates the behavior of the operation at various levels of the cache hierarchy, for both WBINVD and WBNOINVD, with respect to lower branches in the cache hierarchy. If the feature bit is 0, the instruction causes the write back and (for WBINVD) invalidation of all lower level caches of other processors sharing the designated level of cache. If the feature bit is 1, the instruction does not necessarily cause the write back and invalidation of all lower level caches of other processors sharing the designated level of cache. See Appendix E, “Obtaining Processor Information Via the CPUID Instruction,” on page 593 for more information on using the CPUID function.

The INVD instruction can be used when cache coherence with memory is not important.

These instructions do not invalidate TLB caches.

These are privileged instructions. The current privilege level of a procedure invalidating the processor’s internal caches must be zero.

WBINVD and WBNOINVD are serializing instructions

Support for WBNOINVD is indicated by CPUID Fn8000_0008_EBX[WBNOINVD] = 1. However, the encoding of WBNOINVD results in it being interpreted as WBINVD on processors that do not explicitly support WBNOINVD, including legacy processors. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

On some processor implementations, WBINVD and WBNOINVD can be made interruptible by setting EFER.INTWB to 1. When this bit is set, the processor periodically checks for all types of interrupts (SMI, INTR, NMI, etc.) while flushing the caches. If an interrupt is observed, the processor stops flushing the caches, saves the instruction pointer and transfers control to the interrupt handler. Upon returning (via an IRET), the processor restarts the flush process from the beginning as lines will have been modified and cached while executing the interrupt handler. Support for setting EFER.INTWB is indicated by CPUID Fn8008_0008_EBX[INT_WBINVD] (bit 13) = 1.

Mnemonic	Opcode	Description
WBINVD	0F 09	Write modified cache lines to main memory, invalidate internal caches, and trigger external cache flushes.
WBNOINVD	F3 0F 09	Write modified cache lines to main memory and trigger external cache flushes.

Related Instructions

CLFLUSH, CLWB, INVD

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP		X	X	CPL was not 0.

WRMSR Write to Model-Specific Register

Writes data to 64-bit model-specific registers (MSRs). These registers are widely used in performance-monitoring and debugging applications, as well as testability and program execution tracing.

This instruction writes the contents of the EDX:EAX register pair into a 64-bit model-specific register specified in the ECX register. The 32 bits in the EDX register are mapped into the high-order bits of the model-specific register and the 32 bits in EAX form the low-order 32 bits.

This instruction must be executed at a privilege level of 0 or a general protection fault #GP(0) will be raised. This exception is also generated if an attempt is made to specify a reserved or unimplemented model-specific register in ECX.

WRMSR is a serializing instruction for most MSRs, however some x2APIC and AVIC MSRs may have relaxed serialization semantics. See the APIC and AVIC sections in volume 2 for details.

Support for the WRMSR instruction is indicated by CPUID Fn0000_0001_EDX[MSR] = 1 OR CPUID Fn8000_0001_EDX[MSR] = 1. For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

The CPUID instruction can provide model information useful in determining the existence of a particular MSR.

See “Model-Specific Registers (MSRs)” in *Volume 2: System Programming*, for more information about model-specific registers, machine check architecture, performance monitoring and debug registers.

Mnemonic	Opcode	Description
WRMSR	0F 30	Write EDX:EAX to the MSR specified by ECX.

Related Instructions

RDMSR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The WRMSR instruction is not supported, as indicated by CPUID Fn0000_0001_EDX[MSR] = 0 OR CPUID Fn8000_0001_EDX[MSR] = 0.
General protection, #GP		X	X	CPL was not 0.
	X		X	The value in ECX specifies a reserved or unimplemented MSR address.
	X		X	Writing 1 to any bit that must be zero (MBZ) in the MSR.
	X		X	Writing a non-canonical value to a MSR that can only be written with canonical values.

WRPKRU Write Protection Key Rights

Writes the contents of the 32-bit Protection Key Rights (PKRU) register with the value in EAX. This instruction forces strong memory ordering between load and store instructions preceding the WRPKRU, and load and store instructions that follow the WRPKRU.

This instruction must be executed with ECX=0 and EDX=0, otherwise a general protection fault (#GP) is generated. The upper 32 bits of RCX and RDX are ignored. The WRPKRU instruction ignores operand size overrides.

Memory protection keys must be enabled (CR4.PKE=1), otherwise executing this instruction generates an invalid opcode fault (#UD).

Software can check that system software has enabled memory protection keys (CR4.PKE=1) by testing CPUID Function 0000_0007h_ECX[OSPKE]. (See Section 5, “Protection Key Rights for User Pages” in AMD64 Architecture Programmer’s Manual Volume 2 for more information on memory protection keys.)

WRPKRU can be executed at any privilege level.

Mnemonic	Opcode	Description
WRPKRU	0F 01 EF	Write the value in EAX to the PKRU MSR

Related Instructions

RDPKRU

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	CR4.PKE=0
General protection, #GP			X	ECX was not zero or EDX was not zero

WRSS Write to Shadow Stack

Writes 4 or 8 bytes from the source register operand to the specified address in a shadow stack page. The operand size is 8 bytes in 64-bit mode (when REX.W set to 1) and 4 bytes in all other cases.

If shadow stacks are not enabled at the current privilege level, or if WRSS is not enabled at the current privilege level a #UD exception is generated.

Mnemonic	Opcode	Description
WRSS *mem32*, *reg32*	66 0F 38 F6	Write 4 bytes to shadow stack at mem32
WRSSQ *mem64*, *reg64*	66 0F 38 F6	Write 8 bytes to shadow stack at mem64

Action

```
// see "Pseudocode Definition" on page 57

IF (CPL == 3)
   {
   IF ((CR4.CET && U_CET.SH_STK_EN) == 0)
         EXCEPTION [#UD]
   IF (U_CET.WR_SSTK_EN == 0)
         EXCEPTION [#UD]        // WRSS not enabled in U_CET
   }
ELSE  // CPL <3
   {
   IF ((CR4.CET && S_CET.SH_STK_EN) == 0)
         EXCEPTION [#UD]
   IF (S_CET.WR_SSTK_EN == 0)
         EXCEPTION [#UD]        // WRSS not enabled in S_CET
   }

IF (OPERAND_SIZE == 64)
   {
   temp_LinAdr = Linear_Address(mem64)
   IF (temp_LinAdr is 8-byte aligned)
      SSTK_WRITE_MEM.q[temp_LinAdr] = reg64[63:0] // write reg64
                                                  // to shadow stack
   ELSE
      EXCEPTION [#GP(0)]
   }
ELSE
   {
   temp_LinAdr = Linear_Address(mem32)
   IF (tmp_LinAdr is 4-byte aligned)
      SSTK_WRITE_MEM.d[temp_LinAdr] = reg32[31:0] // write reg32
                                                  // to shadow stack
   ELSE
      EXCEPTION [#GP(0)]
   }
```

```
EXIT
```

Related Instructions

WRUSS

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction is only recognized in protected mode.
			X	CR4.CET = 0.
			X	Shadow stacks are not enabled at the current privilege level.
			X	If CPL == 3 and U_CET.WR_SHSTK_EN = 0.
			X	If CPL !=3 and S_CET.WR_SHSTK_EN = 0.
			X	If mod=11b (register destination was specified).
General protection, #GP			X	Address not 8-byte aligned for 64-bit operand size.
			X	Address not 4-byte aligned for 32-bit operand size.
			X	A memory address exceeded a data segment limit.
			X	In long mode, the address of the memory operand was non-canonical.
			X	A null data segment was used to reference memory.
			X	A non-writeable data segment was used.
			X	An execute-only code segment was used to reference memory.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
Page fault, #PF			X	A page fault resulted from the execution of the instruction.
			X	The destination was not a shadow stack page.

WRUSS Write to User Shadow Stack

Writes 4 or 8 bytes from the source register operand to the specified address in a user shadow stack page. The write is performed with user-mode shadow stack semantics. The operand size is 8 bytes in 64-bit mode (when REX.W set to 1) and 4 bytes in all other cases.

The destination must be a user shadow stack page, otherwise a #PF exception is generated. WRUSS is a privileged instruction and must be executed with CPL=0, otherwise a #GP exception is generated.

Mnemonic	Opcode	Description
WRUSSD *mem32*, *reg32*	66 0F 38 F5	Write 4 bytes to user shadow stack
WRUSSQ *mem64*, *reg64*	66 0F 38 F5	Write 8 bytes to user shadow stack

Action

```
// see "Pseudocode Definition" on page 57

IF (CR4.CET == 0)
      EXCEPTION [#UD]
IF (CPL != 0)
      EXCEPTION [#GP(0)]

IF (OPERAND_SIZE == 64)
   {
   temp_LinAdr = Linear_Address(mem64)
   IF (temp_LinAdr is 8-byte aligned)
      SSTK_WRITE_MEM.q[tmp_LinAdr] = reg64[63:0] // write as user access
   ELSE
      EXCEPTION [#GP(0)]
   }
ELSE
   {
    temp_LinAdr = Linear_Address(mem32)
   IF (tmp_LinAdr is 4-byte aligned)
      SSTK_WRITE_MEM.d[temp_LinAdr] = reg32[31:0] // write as user access
   ELSE
      EXCEPTION [#GP(0)]
   }

EXIT
```

Related Instructions

WRSS

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction is only recognized in protected mode.
			X	CR4.CET = 0.
			X	If mod=11b (register destination was specified).
General protection, #GP			X	If CPL ! = 0.
			X	Address not 8-byte aligned for 64-bit operand size.
			X	Address not 4-byte aligned for 32-bit operand size.
			X	A memory address exceeded a data segment limit .
			X	In long mode, the address of the memory operand was non-canonical.
			X	A null data segment was used to reference memory.
			X	A non-writeable data segment was used.
			X	An execute-only code segment was used to reference memory.
Stack, #SS			X	A memory address exceeded the stack segment limit or was non-canonical.
Page fault, #PF			X	The linear address is not a user shadow stack page.
			X	A page fault resulted from the execution of the instruction.

Appendix A Opcode and Operand Encodings

This appendix specifies the opcode and operand encodings for each instruction in the AMD64 instruction set. As discussed in Chapter 1, “Instruction Encoding,” the basic operation and implied operand type(s) of an instruction are encoded by the binary value of the opcode byte. The correspondence between an opcode binary value and its meaning is provided by the *opcode map*.

Each opcode map has 256 entries and can encode up to 256 different operations. Since the AMD64 instruction set comprises more than 256 instructions, multiple opcode maps are utilized to encode the instruction set. A particular opcode map is selected using the instruction encoding syntax diagrammed in Figure 1-1 on page 2. For each opcode map, values may be reserved or utilized for purposes other than encoding an instruction operation.

To preserve compatibility with future instruction architectural extensions, reserved opcodes should not be used. If a means to reliably cause an invalid-opcode exception (#UD) is required, software should use one of the UD*x* opcodes. These opcodes are set aside for this purpose and will not be used for future instructions. The UD opcodes are located on the secondary opcode map at code points B9h, 0Bh, and FFh.

The following section provides a key to the notation used in the opcode maps to specify the implied operand types.

Opcode-Syntax Notation

In the opcode maps which follow, each table entry represents a specific form of an instruction, identifying the instruction by its mnemonic and listing the operand or operands peculiar to that opcode. If a register-based operand is specified by the opcode itself, the operand is represented directly using the register mnemonic as defined in “Summary of Registers and Data Types” on page 38. If the operand is encoded in one or more bytes following the opcode byte, the following special notation is used to represent the operand and its encoding in more generic terms.

This special notation, used exclusively in the opcode maps, is composed of three parts:

- an initial capital letter that represents the operand source / destination (register-based, memory-based, or immediate) and how it is encoded in the instruction (either as an immediate, or via the ModRM.reg, ModRM.{mod,r/m}, or VEX/XOP.vvvv fields). For register-based operands, the inital letter also specifies the register type (General-purpose, MMX, YMM/XMM, debug, or control register).
- one, two, or three letter modifier (in lowercase) that represents the data type (for example, byte, word, quadword, packed single-precision floating-point vector).
- *x*, which indicates for an SSE instruction that the instruction supports both vector sizes (128 bits and 256 bits). The specific vector size is encoded in the VEX/XOP.L field. L=0 indicates 128 bits and L=1 indicates 256 bits.

The following list describes the meaning of each letter that is used in the first position of the operand notation:

A A far pointer encoded in the instruction. No ModRM byte in the instruction encoding.

B General-purpose register specified by the VEX or XOP vvvv field.

C Control register specified by the ModRM.reg field.

D Debug register specified by the ModRM.reg field.

E General purpose register or memory operand specified by the r/m field of the ModRM byte. For memory operands, the ModRM byte may be followed by a SIB byte to specify one of the indexed register-indirect addressing forms.

F rFLAGS register.

G General purpose register specified by the ModRM.reg field.

H YMM or XMM register specified by the VEX/XOP.vvvv field.

I Immediate value encoded in the instruction immediate field.

J The instruction encoding includes a relative offset that is added to the rIP.

L YMM or XMM register specified using the most-significant 4 bits of an 8-bit immediate value. In legacy or compatibility mode the most significant bit is ignored.

M A memory operand specified by the {mod, r/m} field of the ModRM byte. ModRM.mod ≠ 11b.

*M** A sparse array of memory operands addressed using the VSIB addressing mode. See “VSIB Addressing” in Volume 4.

N 64-bit MMX register specified by the ModRM.r/m field. The ModRM.mod field must be 11b.

O The offset of an operand is encoded in the instruction. There is no ModRM byte in the instruction encoding. Indexed register-indirect addressing using the SIB byte is not supported.

P 64-bit MMX register specified by the ModRM.reg field.

Q 64-bit MMX-register or memory operand specified by the {mod, r/m} field of the ModRM byte. For memory operands, the ModRM byte may be followed by a SIB byte to specify one of the indexed register-indirect addressing forms.

R General purpose register specified by the ModRM.r/m field. The ModRM.mod field must be 11b.

S Segment register specified by the ModRM.reg field.

U YMM/XMM register specified by the ModRM.r/m field. The ModRM.mod field must be 11b.

V YMM/XMM register specified by the ModRM.reg field.

W YMM/XMM register or memory operand specified by the {mod, r/m} field of the ModRM byte. For memory operands, the ModRM byte may be followed by a SIB byte to specify one of the indexed register-indirect addressing forms.

X A memory operand addressed by the DS.rSI registers. Used in string instructions.

Y A memory operand addressed by the ES.rDI registers. Used in string instructions.

The following list provides the key for the second part of the operand notation:

a Two 16-bit or 32-bit memory operands, depending on the effective operand size. Used in the BOUND instruction.

b A byte, irrespective of the effective operand size.

c A byte or a word, depending on the effective operand size.

d A doubleword (32 bits), irrespective of the effective operand size.

do A double octword (256 bits), irrespective of the effective operand size.

i A 16-bit integer.

j A 32-bit integer.

m A bit mask of size equal to the source operand.

mn Where *n* = 2,4,8, or 16. A bit mask of size *n*.

o An octword (128 bits), irrespective of the effective operand size.

o.q Operand is either the upper or lower half of a 128-bit value.

p A 32- or 48-bit far pointer, depending on 16- or 32-bit effective operand size.

pb Vector with byte-wide (8-bit) elements (packed byte).

pd A double-precision (64-bit) floating-point vector operand (packed double-precision).

pdw Vector composed of 32-bit doublewords.

ph A half-precision (16-bit) floating-point vector operand (packed half-precision)

pi Vector composed of 16-bit integers (packed integer).

pj Vector composed of 32-bit integers (packed double integer).

pk Vector composed of 8-bit integers (packed half-word integer).

pq Vector composed of 64-bit integers (packed quadword integer).

pqw Vector composed of 64-bit quadwords (packed quadword).

ps A single-precision floating-point vector operand (packed single-precision).

pw Vector composed of 16-bit words (packed word).

q A quadword (64 bits), irrespective of the effective operand size.

s A 6-byte or 10-byte pseudo-descriptor.

sd A scalar double-precision floating-point operand (scalar double).

sj A scalar doubleword (32-bit) integer operand (scalar double integer).

ss A scalar single-precision floating-point operand (scalar single).

v A word, doubleword, or quadword (in 64-bit mode), depending on the effective operand size.

w A word, irrespective of the effective operand size.

x Instruction supports both vector sizes (128 bits or 256 bits). Size is encoded using the VEX/XOP.L field. (L=0: 128 bits; L=1: 256 bits). This symbol may be appended to *ps* or *pd* to represent a packed single- or double-precision floating-point vector of either size; or to *pk*, *pi*, *pj*, or *pq*, to represent a packed 8-bit, 16-bit, 32-bit, or 64-bit packed integer vector of either size.

y A doubleword or quadword depending on effective operand size.

z A word if the effective operand size is 16 bits, or a doubleword if the effective operand size is 32 or 64 bits.

For some instructions, fields in the ModRM or SIB byte are used as encoding extensions. This is indicated using the following notation:

/n A ModRM-byte *reg* field or SIB-byte *base* field, where *n* is a value between zero (000b) and 7 (111b).

For SSE instructions that take scalar operands, VEX/XOP.L field is ignored.

For immediates and memory-based operands, only the size and not the datatype is indicated. Operand widths and datatypes are specified based on the source operands. For instructions where the result overwrites one of the source registers, the data width and datatype of the result may not match that of the source register. See individual instruction descriptions for more details.

A.1 Opcode Maps

In all of the following opcode maps, cells shaded grey represent reserved opcodes.

A.1.1 Legacy Opcode Maps

Primary Opcode Map. Tables A-1 and A-2 below show the primary opcode map (known in legacy terminology as one-byte opcodes).

Table A-1 below shows those instructions for which the low nibble is in the range 0–7h. Table A-2 on page 508 shows those instructions for which the low nibble is in the range 8–Fh. In both tables, the rows show the full range (0–Fh) of the high nibble, and the columns show the specified range of the low nibble.

Table A-1. Primary Opcode Map (One-byte Opcodes), Low Nibble 0–7h

Nibble[1]	0	1	2	3	4	5	6	7
0	ADD Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH ES[3]	POP ES[3]
1	ADC Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS[3]	POP SS[3]
2	AND Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	seg ES[6]	DAA[3]
3	XOR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	seg SS[6]	AAA[3]
4	INC / REX prefix[5] eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA[3] PUSHD[3]	POPA[3] POPD[3]	BOUND [3] Gv, Ma	ARPL[3] Ew, Gw MOVSXD[4] Gv, Ez	seg FS prefix	seg GS prefix	operand size override prefix	address size override prefix
7	JO Jb	JNO Jb	JB Jb	JNB Jb	JZ Jb	JNZ Jb	JBE Jb	JNBE Jb
8	Group 1[2] Eb, Ib	Ev, Iz	Eb, Ib[3]	Ev, Ib	TEST Eb, Gb	Ev, Gv	XCHG Eb, Gb	Ev, Gv
9	XCHG r8, rAX NOP,PAUSE	rCX/r9, rAX	rDX/r10, rAX	rBX/r11, rAX	rSP/r12, rAX	rBP/r13, rAX	rSI/r14, rAX	rDI/r15, rAX
A	MOV AL, Ob	rAX, Ov	Ob, AL	Ov, rAX	MOVSB Yb, Xb	MOVSW/D/Q Yv, Xv	CMPSB Xb, Yb	CMPSW/D/Q Xv, Yv
B	MOV AL, Ib r8b, Ib	CL, Ib r9b, Ib	DL, Ib r10b, Ib	BL, Ib r11b, Ib	AH, Ib r12b, Ib	CH, Ib r13b, Ib	DH, Ib r14b, Ib	BH, Ib r15b, Ib
C	Group 2[2] Eb, Ib	Ev, Ib	RET near Iw		LES[3] Gz, Mp VEX escape prefix	LDS[3] Gz, Mp VEX escape prefix	Group 11[2] Eb, Ib	Ev, Iz
D	Group 2[2] Eb, 1	Ev, 1	Eb, CL	Ev, CL	AAM Ib[3]	AAD Ib[3]	invalid	XLAT XLATB
E	LOO-PNE/NZJb	LOOPE/Z Jb	LOOP Jb	JrCXZ Jb	IN AL, Ib	eAX, Ib	OUT Ib, AL	Ib, eAX
F	LOCK Prefix	INT1	REPNE Prefix	REP / REPE Prefix	HLT	CMC	Group 3[2] Eb	Ev

Notes:

1. *Rows in this table show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal).*
2. *An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-6 on page 515 for details.*
3. *Invalid in 64-bit mode.*
4. *Valid only in 64-bit mode.*
5. *Used as REX prefixes in 64-bit mode.*
6. *This is a null prefix in 64-bit mode.*

Table A-2. Primary Opcode Map (One-byte Opcodes), Low Nibble 8–Fh

Nibble[1]	8	9	A	B	C	D	E	F
0	OR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH CS[3]	escape to secondary opcode map
1	SBB Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH DS[3]	POP DS[3]
2	SUB Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	seg CS[6]	DAS[3]
3	CMP Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	seg DS[6]	AAS[3]
4	DEC[3] / REX prefix[5] eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	POP rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSH Iz	IMUL Gv, Ev, Iz	PUSH Ib	IMUL Gv, Ev, Ib	INSB Yb, DX	INSW/D Yz, DX	OUTS/ OUTSB DX, Xb	OUTS OUTSW/D DX, Xz
7	JS Jb	JNS Jb	JP Jb	JNP Jb	JL Jb	JNL Jb	JLE Jb	JNLE Jb
8	MOV Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	Mw/Rv, Sw	LEA Gv, M	MOV Sw, Ew	Group 1a[2] XOP escape prefix
9	CBW, CWDE CDQE	CWD, CDQ, CQO	CALL[3] Ap	WAIT FWAIT	PUSHF/D/Q Fv	POPF/D/Q Fv	SAHF	LAHF
A	TEST AL, Ib	rAX, Iz	STOSB Yb, AL	STOSW/D/Q Yv, rAX	LODSB AL, Xb	LODSW/D/Q rAX, Xv	SCASB AL, Yb	SCASW/D/Q rAX, Yv
B	MOV rAX, Iv r8, Iv	rCX, Iv r9, Iv	rDX, Iv r10, Iv	rBX, Iv r11, Iv	rSP, Iv r12, Iv	rBP, Iv r13, Iv	rSI, Iv r14, Iv	rDI, Iv r15, Iv
C	ENTER Iw, Ib	LEAVE	RET far Iw		INT3	INT Ib	INTO[3]	IRET, IRETD, IRETQ
D	x87 instructions see Table A-15 on page 526							
E	CALL Jz	JMP Jz	Ap[3]	Jb	IN AL, DX	eAX, DX	OUT DX, AL	DX, eAX
F	CLC	STC	CLI	STI	CLD	STD	Group 4[2] Eb	Group 5[2]

Notes:

1. *Rows in this table show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal).*
2. *An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-6 on page 515 for details.*
3. *Invalid in 64-bit mode.*
4. *Valid only in 64-bit mode.*
5. *Used as REX prefixes in 64-bit mode.*
6. *This is a null prefix in 64-bit mode.*

Secondary Opcode Map. As described in "Encoding Syntax" on page 1, the escape code 0Fh indicates the switch from the primary to the secondary opcode map. In legacy terminology, the secondary opcode map is presented as a listing of "two-byte" opcodes where the first byte is 0Fh. Tables A-3 and A-4 show the secondary opcode map.

Table A-3 below shows those instructions for which the low nibble is in the range 0–7h. Table A-4 on page 512 shows those instructions for which the low nibble is in the range 8–Fh. In both tables, the rows show the full range (0–Fh) of the high nibble, and the columns show the specified range of the low nibble. Note the added column labeled “prefix.”

For the secondary opcode map shown below, the legacy prefixes 66h, F2h, and F3 are repurposed to provide additional opcode encoding space. For those rows that utilize them, the presence of a 66h, F2h, or F3h prefix changes the operation or the operand types specified by the corresponding opcode value.

As discussed in “Encoding Extensions Using the ModRM Byte” on page 515, some opcode values represent a group of instructions. This is denoted in the map entry by “Group *n*”, where n = [1:17,P]. Instructions within a group are encoded by the reg field of the ModRM byte. These encodings are specified in Table A-7 on page 517. For some opcodes, both the reg and the r/m field of the ModRM byte are used to extend the encoding. See Table A-8 on page 519.

Table A-3. Secondary Opcode Map (Two-byte Opcodes), Low Nibble 0–7h

Prefix	Nibble[1]	0	1	2	3	4	5	6	7
n/a	0	Group 6[2]	Group 7[2]	LAR Gv, Ew	LSL Gv, Ew		SYSCALL	CLTS	SYSRET
none	1	MOVUPS Vps, Wps	MOVUPS Wps, Vps	MOVLPS Vq, Mq MOVHLPS Vo.q, Uo.q	MOVLPS Mq, Vq	UNPCKLPS Vps,Wps	UNPCKHPS Vps,Wps	MOVHPS Vo.q, Mq MOVLHPS Vo.q, Uo.q	MOVHPS Mq, Vo.q
F3		MOVSS Vss, Wss	MOVSS Wss, Vss	MOVSLDUP Vps, Wps				MOVSHDUP Vps, Wps	
66		MOVUPD Vpd, Wpd	MOVUPD Wpd, Vpd	MOVLPD Vo.q, Mq	MOVLPD Mq, Vo.q	UNPCKLPD Vo.q, Wo.q	UNPCKHPD Vo.q, Wo.q	MOVHPD Vo.q, Mq	MOVHPD Mq, Vo.q
F2		MOVSD Vsd, Wsd	MOVSD Wsd, Vsd	MOVDDUP Vo, Wsd					
n/a	2	MOV[4] Rd/q, Cd/q	MOV[4] Rd/q, Dd/q	MOV[4] Cd/q, Rd/q	MOV[4] Dd/q, Rd/q				
n/a	3	WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER[3]	SYSEXIT[3]		
n/a	4	CMOVO Gv, Ev	CMOVNO Gv, Ev	CMOVB Gv, Ev	CMOVNB Gv, Ev	CMOVZ Gv, Ev	CMOVNZ Gv, Ev	CMOVBE Gv, Ev	CMOVNBE Gv, Ev
none	5	MOVMSKPS Gd, Ups	SQRTPS Vps, Wps	RSQRTPS Vps, Wps	RCPPS Vps, Wps	ANDPS Vps, Wps	ANDNPS Vps, Wps	ORPS Vps, Wps	XORPS Vps, Wps
F3			SQRTSS Vss, Wss	RSQRTSS Vss, Wss	RCPSS Vss, Wss				
66		MOVMSKPD Gd, Upd	SQRTPD Vpd, Wpd			ANDPD Vpd, Wpd	ANDNPD Vpd, Wpd	ORPD Vpd, Wpd	XORPD Vpd, Wpd
F2			SQRTSD Vsd, Wsd						
none	6	PUNPCK-LBW Pq, Qd	PUNPCK-LWD Pq, Qd	PUNPCK-LDQ Pq, Qd	PACKSSWB Ppi, Qpi	PCMPGTB Ppk, Qpk	PCMPGTW Ppi, Qpi	PCMPGTD Ppj, Qpj	PACKUSWB Ppi, Qpi
F3									
66		PUNPCK-LBW Vo.q, Wo.q	PUNPCK-LWD Vo.q, Wo.q	PUNPCK-LDQ Vo.q, Wo.q	PACKSSWB Vpi, Wpi	PCMPGTB Vpk, Wpk	PCMPGTW Vpi, Wpi	PCMPGTD Vpj, Wpj	PACKUSWB Vpi, Wpi
F2									
none	7	PSHUFW Pq, Qq, Ib	Group 12[2]	Group 13[2]	Group 14[2]	PCMPEQB Ppk, Qpk	PCMPEQW Ppi, Qpi	PCMPEQD Ppj, Qpj	EMMS
F3		PSHUFHW Vq, Wq, Ib							
66		PSHUFD Vo, Wo, Ib				PCMPEQB Vpk, Wpk	PCMPEQW Vpi, Wpi	PCMPEQD Vpj, Wpj	
F2		PSHUFLW Vq, Wq, Ib							

Notes:

1. *Rows show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal). All opcodes in this map are immediately preceeded in the instruction encoding by the escape byte 0Fh.*
2. *An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-7 on page 517 for details.*
3. *Invalid in long mode.*
4. *Operand size is based on processor mode.*

Table A-3. Secondary Opcode Map (Two-byte Opcodes), Low Nibble 0–7h (continued)

Prefix	Nibble[1]	0	1	2	3	4	5	6	7
n/a	8	JO Jz	JNO Jz	JB Jz	JNB Jz	JZ Jz	JNZ Jz	JBE Jz	JNBE Jz
n/a	9	SETO Eb	SETNO Eb	SETB Eb	SETNB Eb	SETZ Eb	SETNZ Eb	SETBE Eb	SETNBE Eb
n/a	A	PUSH FS	POP FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
n/a	B	CMPXCHG Eb, Gb	CMPXCHG Ev, Gv	LSS Gz, Mp	BTR Ev, Gv	LFS Gz, Mp	LGS Gz, Mp	MOVZX Gv, Eb	MOVZX Gv, Ew
none	C	XADD Eb, Gb	XADD Ev, Gv	CMPPS Vps, Wps, Ib	MOVNTI My, Gy	PINSRW Pq, Ry/Mw, Ib	PEXTRW Gd, Nq, Ib	SHUFPS Vps, Wps, Ib	Group 9[2] Mq
F3				CMPSS Vss, Wss, Ib					
66				CMPPD Vpd, Wpd, Ib		PINSRW Vo, Ry/Mw, Ib	PEXTRW Gd, Uo, Ib	SHUFPD Vpd, Wpd, Ib	
F2				CMPSD Vsd, Wsd, Ib					
none	D		PSRLW Pq, Qq	PSRLD Pq, Qq	PSRLQ Pq, Qq	PADDQ Pq, Qq	PMULLW Pq, Qq		PMOVMSKB Gd, Nq
F3								MOVQ2DQ Vo, Nq	
66		ADDSUBPD Vpd, Wpd	PSRLW Vo, Wo	PSRLD Vo, Wo	PSRLQ Vo, Wo	PADDQ Vo, Wo	PMULLW Vo, Wo	MOVQ Wq, Vq	PMOVMSKB Gd, Uo
F2		ADDSUBPS Vps, Wps						MOVDQ2Q Pq, Uq	
none	E	PAVGB Pq, Qq	PSRAW Pq, Qq	PSRAD Pq, Qq	PAVGW Pq, Qq	PMULHUW Pq, Qq	PMULHW Pq, Qq		MOVNTQ Mq, Pq
F3								CVTDQ2PD Vpd, Wpj	
66		PAVGB Vo, Wo	PSRAW Vo, Wo	PSRAD Vo, Wo	PAVGW Vo, Wo	PMULHUW Vo, Wo	PMULHW Vo, Wo	CVTTPD2DQ Vpj, Wpd	MOVNTDQ Mo, Vo
F2								CVTPD2DQ Vpj, Wpd	
none	F		PSLLW Pq, Qq	PSLLD Pq, Qq	PSLLQ Pq, Qq	PMULUDQ Pq, Qq	PMADDWD Pq, Qq	PSADBW Pq, Qq	MASKMOVQ Pq, Nq
F3									
66			PSLLW Vpw, Wo.q	PSLLD Vpwd, Wo.q	PSLLQ Vpqw, Wo.q	PMULUDQ Vpj, Wpj	PMADDWD Vpi, Wpi	PSADBW Vpk, Wpk	MASKMOVDQU Vpb, Upb
F2		LDDQU Vo, Mo							

Notes:

1. *Rows show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal). All opcodes in this map are immediately preceeded in the instruction encoding by the escape byte 0Fh.*
2. *An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-7 on page 517 for details.*
3. *Invalid in long mode.*
4. *Operand size is based on processor mode.*

Table A-4. Secondary Opcode Map (Two-byte Opcodes), Low Nibble 8–Fh

Prefix	Nibble[1]	8	9	A	B	C	D	E	F
n/a	0	INVD	WBINVD (F3) WBNOINVD		UD2		Group P[2] PREFETCH	FEMMS	3DNow! See "3DNow!™ Opcodes" on page 522
n/a	1	Group 16[2]	NOP[3]	NOP[3]	NOP[3]	NOP[3]	NOP[3] (F3) RDSSP reg=1, mod=11	NOP[3]	NOP[3]
none	2	MOVAPS Vps, Wps	Wps, Vps	CVTPI2PS Vps, Qpj	MOVNTPS Mo, Vps	CVTTPS2PI Ppj, Wps	CVTPS2PI Ppj, Wps	UCOMISS Vss, Wss	COMISS Vss, Wss
F3				CVTSI2SS Vss, Ey	MOVNTSS Md, Vss	CVTTSS2SI Gy, Wss	CVTSS2SI Gy, Wss		
66		MOVAPD Vpd, Wpd	Wpd, Vpd	CVTPI2PD Vpd, Qpj	MOVNTPD Mo, Vpd	CVTTPD2PI Ppj, Wpd	CVTPD2PI Ppj, Wpd	UCOMISD Vsd, Wsd	COMISD Vsd, Wsd
F2				CVTSI2SD Vsd, Ey	MOVNTSD Mq, Vsd	CVTTSD2SI Gy, Wsd	CVTSD2SI Gy, Wsd		
n/a	3	Escape to 0F_38h opcode map		Escape to 0F_3Ah opcode map					
n/a	4	CMOVS Gv, Ev	CMOVNS Gv, Ev	CMOVP Gv, Ev	CMOVNP Gv, Ev	CMOVL Gv, Ev	CMOVNL Gv, Ev	CMOVLE Gv, Ev	CMOVNLE Gv, Ev
none	5	ADDPS Vps, Wps	MULPS Vps, Wps	CVTPS2PD Vpd, Wps	CVTDQ2PS Vps, Wo	SUBPS Vps, Wps	MINPS Vps, Wps	DIVPS Vps, Wps	MAXPS Vps, Wps
F3		ADDSS Vss, Wss	MULSS Vss, Wss	CVTSS2SD Vsd, Wss	CVTTPS2DQ Vo, Wps	SUBSS Vss, Wss	MINSS Vss, Wss	DIVSS Vss, Wss	MAXSS Vss, Wss
66		ADDPD Vpd, Wpd	MULPD Vpd, Wpd	CVTPD2PS Vps, Wpd	CVTPS2DQ Vo, Wps	SUBPD Vpd, Wpd	MINPD Vpd, Wpd	DIVPD Vpd, Wpd	MAXPD Vpd, Wpd
F2		ADDSD Vsd, Wsd	MULSD Vsd, Wsd	CVTSD2SS Vss, Wsd		SUBSD Vsd, Wsd	MINSD Vsd, Wsd	DIVSD Vsd, Wsd	MAXSD Vsd, Wsd
none	6	PUNPCK-HBW Pq, Qd	PUNPCK-HWD Pq, Qd	PUNPCK-HDQ Pq, Qd	PACKSSDW Pq, Qq			MOVD Py, Ey	MOVQ Pq, Qq
F3									MOVDQU Vo, Wo
66		PUNPCK-HBW Vo, Wq	PUNPCK-HWD Vo, Wq	PUNPCK-HDQ Vo, Wq	PACKSSDW Vo, Wo	PUNPCK-LQDQ Vo, Wq	PUNPCKH-QDQ Vo, Wq	MOVD Vy, Ey	MOVDQA Vo, Wo
F2									

Notes:

1. *Rows show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal). All opcodes in this map are immediately preceeded in the instruction encoding by the escape byte 0Fh.*
2. *An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-7 on page 517 for details.*
3. *This instruction takes a ModRM byte.*

Table A-4. Secondary Opcode Map (Two-byte Opcodes), Low Nibble 8–Fh

Prefix	Nibble[1]	8	9	A	B	C	D	E	F
none	7							MOVD Ey, Py	MOVQ Qq, Pq
F3								MOVQ Vq, Wq	MOVDQU Wo, Vo
66		Group 17[2]	EXTRQ Vo.q, Uo			HADDPD Vpd, Wpd	HSUBPD Vpd, Wpd	MOVD Ey, Vy	MOVDQA Wo, Vo
F2		INSERTQ Vo.q, Uo.q, Ib, Ib	INSERTQ Vo.q, Uo			HADDPS Vps, Wps	HSUBPS Vps, Wps		
n/a	8	JS Jz	JNS Jz	JP Jz	JNP Jz	JL Jz	JNL Jz	JLE Jz	JNLE Jz
n/a	9	SETS Eb	SETNS Eb	SETP Eb	SETNP Eb	SETL Eb	SETNL Eb	SETLE Eb	SETNLE Eb
n/a	A	PUSH GS	POP GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	Group 15[2]	IMUL Gv, Ev
none	B		Group 10[2]	Group 8[2] Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSX Gv, Eb	MOVSX Gv, Ew
F3		POPCNT Gv, Ev				TZCNT Gv, Ev	LZCNT Gv, Ev		
F2									
n/a	C	BSWAP rAX/r8	BSWAP rCX/r9	BSWAP rDX/r10	BSWAP rBX/r11	BSWAP rSP/r12	BSWAP rBP/r13	BSWAP rSI/r14	BSWAP rDI/r15
none	D	PSUBUSB Pq, Qq	PSUBUSW Pq, Qq	PMINUB Pq, Qq	PAND Pq, Qq	PADDUSB Pq, Qq	PADDUSW Pq, Qq	PMAXUB Pq, Qq	PANDN Pq, Qq
F3									
66		PSUBUSB Vo, Wo	PSUBUSW Vo, Wo	PMINUB Vo, Wo	PAND Vo, Wo	PADDUSB Vo, Wo	PADDUSW Vo, Wo	PMAXUB Vo, Wo	PANDN Vo, Wo
F2									
none	E	PSUBSB Pq, Qq	PSUBSW Pq, Qq	PMINSW Pq, Qq	POR Pq, Qq	PADDSB Pq, Qq	PADDSW Pq, Qq	PMAXSW Pq, Qq	PXOR Pq, Qq
F3									
66		PSUBSB Vo, Wo	PSUBSW Vo, Wo	PMINSW Vo, Wo	POR Vo, Wo	PADDSB Vo, Wo	PADDSW Vo, Wo	PMAXSW Vo, Wo	PXOR Vo, Wo
F2									

Notes:

1. *Rows show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal). All opcodes in this map are immediately preceeded in the instruction encoding by the escape byte 0Fh.*
2. *An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-7 on page 517 for details.*
3. *This instruction takes a ModRM byte.*

Table A-4. Secondary Opcode Map (Two-byte Opcodes), Low Nibble 8–Fh

Prefix	Nibble[1]	8	9	A	B	C	D	E	F
none	F	PSUBB Pq, Qq	PSUBW Pq, Qq	PSUBD Pq, Qq	PSUBQ Pq, Qq	PADDB Pq, Qq	PADDW Pq, Qq	PADDD Pq, Qq	UD0
F3									
66		PSUBB Vo, Wo	PSUBW Vo, Wo	PSUBD Vo, Wo	PSUBQ Vo, Wo	PADDB Vo, Wo	PADDW Vo, Wo	PADDD Vo, Wo	
F2									

Notes:
1. *Rows show the high opcode nibble, columns show the low opcode nibble (both in hexadecimal). All opcodes in this map are immediately preceeded in the instruction encoding by the escape byte 0Fh.*
2. *An opcode extension is specified using the reg field of the ModRM byte (ModRM bits [5:3]) which follows the opcode. See Table A-7 on page 517 for details.*
3. *This instruction takes a ModRM byte.*

rFLAGS Condition Codes for CMOV*cc*, J*cc*, and SET*cc* Instructions. Table A-5 shows the rFLAGS condition codes specified by the low nibble in the opcode of the CMOV*cc*, J*cc*, and SET*cc* instructions.

Table A-5. rFLAGS Condition Codes for CMOV*cc*, J*cc*, and SET*cc*

Low Nibble of Opcode (hex)	rFLAGS Value	*cc* Mnemonic	Arithmetic Type	Condition(s)
0	OF = 1	O	Signed	Overflow
1	OF = 0	NO		No Overflow
2	CF = 1	B, C, NAE	Unsigned	Below, Carry, Not Above or Equal
3	CF = 0	NB, NC, AE		Not Below, No Carry, Above or Equal
4	ZF = 1	Z, E		Zero, Equal
5	ZF = 0	NZ, NE		Not Zero, Not Equal
6	CF = 1 or ZF = 1	BE, NA		Below or Equal, Not Above
7	CF = 0 and ZF = 0	NBE, A		Not Below or Equal, Above
8	SF = 1	S	Signed	Sign
9	SF = 0	NS		Not Sign
A	PF = 1	P, PE	n/a	Parity, Parity Even
B	PF = 0	NP, PO		Not Parity, Parity Odd
C	(SF xor OF) = 1	L, NGE	Signed	Less than, Not Greater than or Equal to
D	(SF xor OF) = 0	NL, GE		Not Less than, Greater than or Equal to
E	(SF xor OF) = 1 or ZF = 1	LE, NG		Less than or Equal to, Not Greater than
F	(SF xor OF) = 0 and ZF = 0	NLE, G		Not Less than or Equal to, Greater than

Encoding Extensions Using the ModRM Byte. The ModRM byte, which immediately follows the opcode byte, is used in certain instruction encodings to provide additional opcode bits with which to define the function of the instruction. ModRM bytes have three fields—*mod*, *reg*, and *r/m*, as shown in Figure A-1.

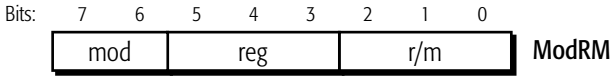


Figure A-1. ModRM-Byte Fields

In most cases, the *reg* field (bits [5:3]), and in some cases, the r/m field (bits [2:0]) provide the additional bits used to extend the encodings of the opcode byte. In the case of the x87 floating-point instructions, the entire ModRM byte is used to extend the opcode encodings.

Table A-6 shows how the ModRM.*reg* field is used to extend the range of opcodes in the primary opcode map. The opcode ranges are organized into *groups* of opcode extensions. The group number is shown in the left-most column. These groups are referenced in the primary opcode map shown in Table A-1 on page 507 and Table A-2 on page 508. An entry of "n.a." in the Prefix column means that prefixes are not applicable to the opcodes in that row. Prefixes only apply to certain 64-bit media and SSE instructions.

Table A-7 on page 517 shows how the ModRM.*reg* field is used to extend the range of the opcodes in the secondary opcode map.

The /0 through /7 notation for the ModRM *reg* field (bits [5:3]) in the tables below means that the three-bit field contains a value from zero (000b) to 7 (111b).

Table A-6. ModRM.reg Extensions for the Primary Opcode Map[1]

Group Number	Prefix	Opcode	ModRM *reg* Field							
			/0	/1	/2	/3	/4	/5	/6	/7
Group 1	n/a	80	ADD Eb, Ib	OR Eb, Ib	ADC Eb, Ib	SBB Eb, Ib	AND Eb, Ib	SUB Eb, Ib	XOR Eb, Ib	CMP Eb, Ib
		81	ADD Ev, Iz	OR Ev, Iz	ADC Ev, Iz	SBB Ev, Iz	AND Ev, Iz	SUB Ev, Iz	XOR Ev, Iz	CMP Ev, Iz
		82	ADD Eb, Ib[2]	OR Eb, Ib[2]	ADC Eb, Ib[2]	SBB Eb, Ib[2]	AND Eb, Ib[2]	SUB Eb, Ib[2]	XOR Eb, Ib[2]	CMP Eb, Ib[2]
		83	ADD Ev, Ib	OR Ev, Ib	ADC Ev, Ib	SBB Ev, Ib	AND Ev, Ib	SUB Ev, Ib	XOR Ev, Ib	CMP Ev, Ib

Notes:
1. *See Table A-7 on page 517 for ModRM extensions for the secondary (two-byte) ocode map.*
2. *Invalid in 64-bit mode.*
3. *This instruction takes a ModRM byte.*
4. *Reserved prefetch encodings are aliased to the /0 encoding (PREFETCH Exclusive) for future compatibility.*
5. *Redundant encoding generally unsupported by tools..*

Table A-6. ModRM.reg Extensions for the Primary Opcode Map[1] (continued)

Group Number	Prefix	Opcode	ModRM *reg* Field							
			/0	/1	/2	/3	/4	/5	/6	/7
Group 1a	n/a	8F	POP Ev	XOP						
Group 2	n/a	C0	ROL Eb, Ib	ROR Eb, Ib	RCL Eb, Ib	RCR Eb, Ib	SHL/SAL Eb, Ib	SHR Eb, Ib	SHL/SAL[5] Eb, Ib	SAR Eb, Ib
		C1	ROL Ev, Ib	ROR Ev, Ib	RCL Ev, Ib	RCR Ev, Ib	SHL/SAL Ev, Ib	SHR Ev, Ib	SHL/SAL[5] Ev, Ib	SAR Ev, Ib
		D0	ROL Eb, 1	ROR Eb, 1	RCL Eb, 1	RCR Eb, 1	SHL/SAL Eb, 1	SHR Eb, 1	SHL/SAL[5] Eb, 1	SAR Eb, 1
		D1	ROL Ev, 1	ROR Ev, 1	RCL Ev, 1	RCR Ev, 1	SHL/SAL Ev, 1	SHR Ev, 1	SHL/SAL[5] Ev, 1	SAR Ev, 1
		D2	ROL Eb, CL	ROR Eb, CL	RCL Eb, CL	RCR Eb, CL	SHL/SAL Eb, CL	SHR Eb, CL	SHL/SAL[5] Eb, CL	SAR Eb, CL
		D3	ROL Ev, CL	ROR Ev, CL	RCL Ev, CL	RCR Ev, CL	SHL/SAL Ev, CL	SHR Ev, CL	SHL/SAL[5] Ev, CL	SAR Ev, CL
Group 3	n/a	F6	TEST Eb,Ib		NOT Eb	NEG Eb	MUL Eb	IMUL Eb	DIV Eb	IDIV Eb
		F7	TEST Ev,Iz		NOT Ev	NEG Ev	MUL Ev	IMUL Ev	DIV Ev	IDIV Ev
Group 4	n/a	FE	INC Eb	DEC Eb						
Group 5	n/a	FF	INC Ev	DEC Ev	CALL Ev	CALL Mp	JMP Ev	JMP Mp	PUSH Ev	
Group 11	n/a	C6	MOV Eb, Ib							
	n/a	C7	MOV Ev, Iz							

Notes:

1. *See Table A-7 on page 517 for ModRM extensions for the secondary (two-byte) ocode map.*
2. *Invalid in 64-bit mode.*
3. *This instruction takes a ModRM byte.*
4. *Reserved prefetch encodings are aliased to the /0 encoding (PREFETCH Exclusive) for future compatibility.*
5. *Redundant encoding generally unsupported by tools..*

Table A-7. ModRM.reg Extensions for the Secondary Opcode Map

Group Number	Prefix	Opcode	ModRM *reg* Field							
			/0	/1	/2	/3	/4	/5	/6	/7
Group 6	n/a	0F 00	SLDT Mw/Rv	STR Mw/Rv	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
Group 7	n/a	0F 01	SGDT Ms	SIDT Ms MONITOR[1] MWAIT	LGDT Ms XGETBV[1] XSETBV	LIDT Ms SVM[1]	SMSW Mw / Rv	RSTORSSP[1] (mod!=11)	LMSW Ew	INVLPG Mb SWAPGS[1] RDTSCP
Group 8	n/a	0F BA					BT Ev, Ib	BTS Ev, Ib	BTR Ev, Ib	BTC Ev, Ib
Group 9	none	0F C7		CMPX-CHG8B Mq					RDRAND Rv	RDSEED Rv
	66			CMPX-CHG16B Mo						
	F2									
	F3									RDPID Rd/q
Group 10	n/a	0F B9	UD1							
Group 12	none	0F 71			PSRLW Nq, Ib		PSRAW Nq, Ib		PSLLW Nq, Ib	
	66				PSRLW Uo, Ib		PSRAW Uo, Ib		PSLLW Uo, Ib	
	F2, F3									
Group 13	none	0F 72			PSRLD Nq, Ib		PSRAD Nq, Ib		PSLLD Nq, Ib	
	66				PSRLD Uo, Ib		PSRAD Uo, Ib		PSLLD Uo, Ib	
	F2, F3									
Group 14	none	0F 73			PSRLQ Nq, Ib				PSLLQ Nq, Ib	
	66				PSRLQ Uo, Ib	PSRLDQ Uo, Ib			PSLLQ Uo, Ib	PSLLDQ Uo, Ib
	F2, F3									

Notes:

1. *Opcode is extended further using the r/m field of the ModRM byte in conjunction with the reg field. See Table A-8 on page 519 for ModRM.r/m extensions of this opcode.*
2. *Invalid in 64-bit mode.*
3. *This instruction takes a ModRM byte.*
4. *Reserved prefetch encodings are aliased to the /0 encoding (PREFETCH Exclusive) for future compatibility.*
5. *ModRM.mod = 11b.*
6. *ModRM.mod ≠ 11b.*
7. *ModRM.mod ≠ 11b, ModRM.mod = 11b is an invalid encoding.*

Table A-7. ModRM.reg Extensions for the Secondary Opcode Map

Group Number	Prefix	Opcode	ModRM *reg* Field							
			/0	/1	/2	/3	/4	/5	/6	/7
Group 15	none	0F AE	FXSAVE M	FXRSTOR M	LDMXCSR Md	STMXCSR Md	XSAVE M[6]	LFENCE[5] XRSTOR M[6]	MFENCE[5] XSAVE-OPT M[6]	SFENCE[5] CLFLUSH Mb[6]
	F3		RDFSBASE Rv	RDGSBASE Rv	WRFSBASE Rv	WRGS-BASE Rv		INCSSP	CLRSSBSY	
	F2									
	66								CLWB Mb[6]	
Group 16	n/a.	0F 18	PREFETCH NTA	PREFETCH T0	PREFETCH T1	PREFETCH T2	NOP	NOP	NOP	NOP
Group 17	66	0F 78	EXTRQ Vo.q, Ib, Ib							
	none, F2, F3									
Group P	n/a.	0F 0D	PREFETCH Exclusive	PREFETCH Modified	PREFETCH[4]	PREFETCH Modified	PREFETCH[4]	PREFETCH[4]	PREFETCH[4]	PREFETCH[4]

Notes:

1. *Opcode is extended further using the r/m field of the ModRM byte in conjunction with the reg field. See Table A-8 on page 519 for ModRM.r/m extensions of this opcode.*
2. *Invalid in 64-bit mode.*
3. *This instruction takes a ModRM byte.*
4. *Reserved prefetch encodings are aliased to the /0 encoding (PREFETCH Exclusive) for future compatibility.*
5. *ModRM.mod = 11b.*
6. *ModRM.mod ≠ 11b.*
7. *ModRM.mod ≠ 11b, ModRM.mod = 11b is an invalid encoding.*

Secondary Opcode Map, ModRM Extensions for Opcode 01h . Table A-8 below shows the ModRM byte encodings for the 01h opcode. In the table the full ModRM byte is listed below the instruction in hexadecimal, with ellipses representing the [0Fh, 01h] opcode bytes.

Table A-8. Opcode 01h ModRM Extensions

reg Field	Prefix	ModRM.r/m Field							
		0	1	2	3	4	5	6	7
/1	none	MONITOR (...C8)	MWAIT (...C9)	CLAC (...CA)	STAC (...CB)				
/2	none	XGETBV (...D0)	XSETBV (...D1)						
/3	none	VMRUN (...D8)	VMMCALL (...D9)	VMLOAD (...DA)	VMSAVE (...DB)	STGI (...DC)	CLGI (...DD)	SKINIT (...DE)	INVLPGA (...DF)
	F3		VMGEXIT (...D9)						
	F2								
/5	none							RDPKRU	WRPKRU
	F3	SETSSBSY		SAVE-PREVSSP					
/7	none	SWAPGS (...F8)	RDTSCP (...F9)	MON...ITORX (FA)	MWAITX (...FB)		RDPRU (...FD)		
	F3			MCOMMIT (F3...FA)				RMPADJUST (F3...FE)	PSMASH (F3...FF)
	F2							RMPUPDATE (F2...FE)	PVALIDATE (F2...FF)
ModRM.mod = 11b									

0F_38h and 0F_3Ah Opcode Maps. The 0F_38h and 0F_3Ah opcode maps are used primarily to encode the legacy SSE instructions. In legacy terminology, these maps are presented as three-byte opcodes where the first two bytes are {0Fh, 38h} and {0Fh, 3Ah} respectively.

In these maps the legacy prefixes F2h and F3h are repurposed to provide additional opcode encoding space. In rows [0:E] the legacy prefix 66h is also used to modify the opcode. However, in row F, 66h is used as an operand-size override. See the CRC32 instruction as an example.

The 0F_38h opcode map is presented below in Tables A-9 and A-10. The 0F_3Ah opcode map is presented in Tables A-11 and A-12.

Table A-9. 0F_38h Opcode Map, Low Nibble = [0h:7h]

Prefix	Opcode	x0	x1	x2	x3	x4	x5	x6	x7
none	0x	PSHUFB Ppb, Qpb	PHADDW Ppi, Qpi	PHADDD Ppj, Qpj	PHADDSW Ppi, Qpi	PMADDUBSW Ppk, Qpk	PHSUBW Ppi, Qpi	PHSUBD Ppj, Qpj	PHSUBSW Ppi, Qpi
66		PSHUFB PVb, Wpb	PHADDW Vpi, Wpi	PHADDD Vpj, Vpj	PHADDSW Vpi, Wpi	PMADDUBSW Vpk, Wpk	PHSUBW Vpi, Wpi	PHSUBD Vpj, Wpj	PHSUBSW Vpi, Wpi
none	1x								
66		PBLENDVB Vpb, Wpb				BLENDVPS Vps, Wps	PBLENDVB Vpb, Wpb		PTEST Vo, Wo
none	2x								
66		PMOVSXBW Vpi, Wpk	PMOVSXBD Vpj, Wpk	PMOVSXBQ Vpq, Wpk	PMOVSXWD Vpj, Wpi	PMOVSXWQ Vpq, Wpi	PMOVSXDQ Vpq, Wpj		
none	3x								
66		PMOVZXBW Vpi, Wpk	PMOVZXBD Vpj, Wpk	PMOVZXBQ Vpq, Wpk	PMOVZXWD Vpj, Wpi	PMOVZXWQ Vpq, Wpi	PMOVZXDQ Vpq, Wpj		PCMPGTQ Vpq, Wpq
none	4x								
66		PMULLD Vpj, Wpj	PHMINPOSUW Vpi, Wpi						
. . .	5x-Ex	. . .							
none	Fx	MOVBE Gv, Mv	MOVBE Mv, Gv					WRSS My, Gy	
F2		CRC32 Gy, Eb	CRC32 Gy, Ev						
66		MOVBE Gv, Mv	MOVBE Gv, Mv				WRUSS My, Gy		
66 and F2		CRC32 Gy, Eb	CRC32 Gy, Ev						

Table A-10. 0F_38h Opcode Map, Low Nibble = [8h:Fh]

Prefix	Opcode	x8	x9	xA	xB	xC	xD	xE	xF
none	0x	PSIGNB Ppk, Qpk	PSIGNW Ppi, Qpi	PSIGND Ppj, Qpj	PMULHRSW Ppi, Qpi				
66		PSIGNB Vpk, Wpk	PSIGNW Vpi, Wpi	PSIGND Vpj, Wpj	PMULHRSW Vpi, Wpi				
none	1x					PABSB Ppk, Qpk	PABSW Ppi, Qpi	PABSD Ppj, Qpj	
66						PABSB Vpk, Wpk	PABSW Vpi, Wpi	PABSD Vpj, Wpj	
none	2x								
66		PMULDQ Vpq, Wpj	PCMPEQQ Vpq, Wpq	MOVNTDQA Vo, Mo	PACKUSDW Vpi, Wpj				
none	3x								
66		PMINSB Vpk, pk	PMINSD Vpj, Wpj	PMINUW Vpi, Wpi	PMINUD Vpj, Wpj	PMAXSB Vpk, Wpk	PMAXSD Vpj, Wpj	PMAXUW Vpi, Wpi	PMAXUD Vpj, Wpj
	4xh-Cxh	. . .							
66	Dx				AESIMC Vo, Wo	AESENC Vo, Wo	AESENCLAST Vo, Wo	AESDEC Vo, Wo	AESDECLAST Vo, Wo
. . .	Exh-Fxh	. . .							

Table A-11. 0F_3Ah Opcode Map, Low Nibble = [0h:7h]

Prefix	Opcode	x0	x1	x2	x3	x4	x5	x6	x7
n/a	0x								
none	1x								
66						PEXTRB Mb, Vpk, Ib PEXTRB Ry, Vpk, Ib	PEXTRW Mw, Vpw, Ib PEXTRW Ry, Vpw, Ib	PEXTRD Ed, Vpj, Ib PEXTRQ[1] Eq, Vpq, Ib	EXTRACTPS Md, Vps, Ib EXTRACTPS Ry, Vps, Ib
none	2x								
66		PINSRB Vpk, Mb, Ib PINSRB Vpk, Ry, Ib	INSERTPS Vps, Md, Ib INSERTPS Vps, Uo, Ib	PINSRD Vpj, Ed, Ib PINSRQ[1] Vpq, Eq, Ib					
. . .	3x	. . .							
none	4x								
66		DPPS Vps, Wps, Ib	DPPD Vpd, Wpd, Ib	MPSADBW Vpk, Wpk, Ib		PCLMULQDQ Vpq, Wpq, Ib			
n/a	5x								
none	6x								
66		PCMPESTRM Vo, Wo, Ib	PCMPESTRI Vo, Wo, Ib	PCMPISTRM Vo, Wo, Ib	PCMPISTRI Vo, Wo, Ib				
. . .	7x-Ex	. . .							
n/a	Fx								

Note 1: When REX prefix is present

Table A-12. 0F_3Ah Opcode Map, Low Nibble = [8h:Fh]

66	2x	PINSRB Vpk, Mb, Ib PINSRB Vpk, Ry, Ib	INSERTPS Vps, Md, Ib INSERTPS Vps, Uo, Ib	PINSRD Vpj, Ed, Ib PINSRQ[1] Vpq, Eq, Ib					
. . .	3x	. . .							
none	4x								
66		DPPS Vps, Wps, Ib	DPPD Vpd, Wpd, Ib	MPSADBW Vpk, Wpk, Ib		PCLMULQDQ Vpq, Wpq, Ib			

A.1.2 3DNow!™ Opcodes

The 64-bit media instructions include the MMX™ instructions and the AMD 3DNow!™ instructions. The MMX instructions are encoded using two opcode bytes, as described in “Secondary Opcode Map” on page 508.

The 3DNow! instructions are encoded using two 0Fh opcode bytes and an immediate byte that is located at the last byte position of the instruction encoding. Thus, the format for 3DNow! instructions is:

```
0Fh 0Fh [ModRM] [SIB] [displacement] imm8_opcode
```

Table A-13 and Table A-14 on page 524 show the immediate byte following the opcode bytes for 3DNow! instructions. In these tables, rows show the high nibble of the immediate byte, and columns show the low nibble of the immediate byte. Table A-13 shows the immediate bytes whose low nibble is in the range 0–7h. Table A-14 shows the same for immediate bytes whose low nibble is in the range 8–Fh.

Byte values shown as *reserved* in these tables have implementation-specific functions, which can include an invalid-opcode exception.

Table A-13. Immediate Byte for 3DNow!™ Opcodes, Low Nibble 0–7h

Nibble[1]	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								
8								
9	PFCMPGE Pq, Qq				PFMIN Pq, Qq		PFRCP Pq, Qq	PFRSQRT Pq, Qq
A	PFCMPGT Pq, Qq				PFMAX Pq, Qq		PFRCPIT1 Pq, Qq	PFRSQIT1 Pq, Qq
B	PFCMPEQ Pq, Qq				PFMUL Pq, Qq		PFRCPIT2 Pq, Qq	PMULHRW Pq, Qq
C								
D								
E								
F								

Notes:

1. *All 3DNow!™ opcodes consist of two 0Fh bytes. This table shows the immediate byte for 3DNow! opcodes. Rows show the high nibble of the immediate byte. Columns show the low nibble of the immediate byte.*

Table A-14. Immediate Byte for 3DNow!™ Opcodes, Low Nibble 8–Fh

Nibble[1]	8	9	A	B	C	D	E	F
0					PI2FW Pq, Qq	PI2FD Pq, Qq		
1					PF2IW Pq, Qq	PF2ID Pq, Qq		
2								
3								
4								
5								
6								
7								
8			PFNACC Pq, Qq				PFPNACC Pq, Qq	
9			PFSUB Pq, Qq				PFADD Pq, Qq	
A			PFSUBR Pq, Qq				PFACC Pq, Qq	
B				PSWAPD Pq, Qq				PAVGUSB Pq, Qq
C								
D								
E								
F								

Notes:

1. *All 3DNow!™ opcodes consist of two 0Fh bytes. This table shows the immediate byte for 3DNow! opcodes. Rows show the high nibble of the immediate byte. Columns show the low nibble of the immediate byte.*

A.1.3 x87 Encodings

All x87 instructions begin with an opcode byte in the range D8h to DFh, as shown in Table A-2 on page 508. These opcodes are followed by a ModRM byte that further defines the opcode. Table A-15 shows both the opcode byte and the ModRM byte for each x87 instruction.

There are two significant ranges for the ModRM byte for x87 opcodes: 00–BFh and C0–FFh. When the value of the ModRM byte falls within the first range, 00–BFh, the opcode uses only the *reg* field to further define the opcode. When the value of the ModRM byte falls within the second range, C0–FFh, the opcode uses the entire ModRM byte to further define the opcode.

Byte values shown as *reserved* or *invalid* in Table A-15 have implementation-specific functions, which can include an invalid-opcode exception.

The basic instructions FNSTENV, FNSTCW, FNCLEX, FNINIT, FNSAVE, FNSTSW, and FNSTSW do not check for possible floating point exceptions before operating. Utility versions of these mnemonics are provided that insert an FWAIT (opcode 9B) before the corresponding non-waiting instruction. These are FSTENV, FSTCW, FCLEX, FINIT, FSAVE, and FSTSW. For further information on wait and non-waiting versions of these instructions, see their corresponding pages in Volume 5.

Table A-15. x87 Opcodes and ModRM Extensions

Opcode	ModRM *mod* Field	ModRM *reg* Field							
		/0	/1	/2	/3	/4	/5	/6	/7
D8	!11	00–BF							
		FADD mem32-real	FMUL mem32real	FCOM mem32real	FCOMP mem32real	FSUB mem32real	FSUBR mem32-real	FDIV mem32real	FDIVR mem32real
	11	C0 FADD ST(0), ST(0)	C8 FMUL ST(0), ST(0)	D0 FCOM ST(0), ST(0)	D8 FCOMP ST(0), ST(0)	E0 FSUB ST(0), ST(0)	E8 FSUBR ST(0), ST(0)	F0 FDIV ST(0), ST(0)	F8 FDIVR ST(0), ST(0)
		C1 FADD ST(0), ST(1)	C9 FMUL ST(0), ST(1)	D1 FCOM ST(0), ST(1)	D9 FCOMP ST(0), ST(1)	E1 FSUB ST(0), ST(1)	E9 FSUBR ST(0), ST(1)	F1 FDIV ST(0), ST(1)	F9 FDIVR ST(0), ST(1)
		C2 FADD ST(0), ST(2)	CA FMUL ST(0), ST(2)	D2 FCOM ST(0), ST(2)	DA FCOMP ST(0), ST(2)	E2 FSUB ST(0), ST(2)	EA FSUBR ST(0), ST(2)	F2 FDIV ST(0), ST(2)	FA FDIVR ST(0), ST(2)
		C3 FADD ST(0), ST(3)	CB FMUL ST(0), ST(3)	D3 FCOM ST(0), ST(3)	DB FCOMP ST(0), ST(3)	E3 FSUB ST(0), ST(3)	EB FSUBR ST(0), ST(3)	F3 FDIV ST(0), ST(3)	FB FDIVR ST(0), ST(3)
		C4 FADD ST(0), ST(4)	CC FMUL ST(0), ST(4)	D4 FCOM ST(0), ST(4)	DC FCOMP ST(0), ST(4)	E4 FSUB ST(0), ST(4)	EC FSUBR ST(0), ST(4)	F4 FDIV ST(0), ST(4)	FC FDIVR ST(0), ST(4)
		C5 FADD ST(0), ST(5)	CD FMUL ST(0), ST(5)	D5 FCOM ST(0), ST(5)	DD FCOMP ST(0), ST(5)	E5 FSUB ST(0), ST(5)	ED FSUBR ST(0), ST(5)	F5 FDIV ST(0), ST(5)	FD FDIVR ST(0), ST(5)
		C6 FADD ST(0), ST(6)	CE FMUL ST(0), ST(6)	D6 FCOM ST(0), ST(6)	DE FCOMP ST(0), ST(6)	E6 FSUB ST(0), ST(6)	EE FSUBR ST(0), ST(6)	F6 FDIV ST(0), ST(6)	FE FDIVR ST(0), ST(6)
		C7 FADD ST(0), ST(7)	CF FMUL ST(0), ST(7)	D7 FCOM ST(0), ST(7)	DF FCOMP ST(0), ST(7)	E7 FSUB ST(0), ST(7)	EF FSUBR ST(0), ST(7)	F7 FDIV ST(0), ST(7)	FF FDIVR ST(0), ST(7)

Table A-15. x87 Opcodes and ModRM Extensions (continued)

Opcode	ModRM *mod* Field	ModRM *reg* Field							
		/0	/1	/2	/3	/4	/5	/6	/7
D9	!11	00–BF							
		FLD mem32-real		FST mem32real	FSTP mem32real	FLDENV mem14/28env	FLDCW mem16	FNSTENV mem14/28env	FNSTCW mem16
	11	C0 FLD ST(0), ST(0)	C8 FXCH ST(0), ST(0)	D0 FNOP	D8 reserved	E0 FCHS	E8 FLD1	F0 F2XM1	F8 FPREM
		C1 FLD ST(0), ST(1)	C9 FXCH ST(0), ST(1)	D1 invalid	D9 reserved	E1 FABS	E9 FLDL2T	F1 FYL2X	F9 FYL2XP1
		C2 FLD ST(0), ST(2)	CA FXCH ST(0), ST(2)	D2 invalid	DA reserved	E2 invalid	EA FLDL2E	F2 FPTAN	FA FSQRT
		C3 FLD ST(0), ST(3)	CB FXCH ST(0), ST(3)	D3 invalid	DB reserved	E3 invalid	EB FLDPI	F3 FPATAN	FB FSINCOS
		C4 FLD ST(0), ST(4)	CC FXCH ST(0), ST(4)	D4 invalid	DC reserved	E4 FTST	EC FLDLG2	F4 FXTRACT	FC FRNDINT
		C5 FLD ST(0), ST(5)	CD FXCH ST(0), ST(5)	D5 invalid	DD reserved	E5 FXAM	ED FLDLN2	F5 FPREM1	FD FSCALE
		C6 FLD ST(0), ST(6)	CE FXCH ST(0), ST(6)	D6 invalid	DE reserved	E6 invalid	EE FLDZ	F6 FDECSTP	FE FSIN
		C7 FLD ST(0), ST(7)	CF FXCH ST(0), ST(7)	D7 invalid	DF reserved	E7 invalid	EF invalid	F7 FINCSTP	FF FCOS

Table A-15. x87 Opcodes and ModRM Extensions (continued)

Opcode	ModRM *mod* Field	ModRM *reg* Field							
		/0	/1	/2	/3	/4	/5	/6	/7
DA	!11	00–BF							
		FIADD mem32int	FIMUL mem32int	FICOM mem32int	FICOMP mem32int	FISUB mem32int	FISUBR mem32int	FIDIV mem32int	FIDIVR mem32int
	11	C0 FCMOVB ST(0), ST(0)	C8 FCMOVE ST(0), ST(0)	D0 FCMOVBE ST(0), ST(0)	D8 FCMOVU ST(0), ST(0)	E0 invalid	E8 invalid	F0 invalid	F8 invalid
		C1 FCMOVB ST(0), ST(1)	C9 FCMOVE ST(0), ST(1)	D1 FCMOVBE ST(0), ST(1)	D9 FCMOVU ST(0), ST(1)	E1 invalid	E9 FUCOMPP	F1 invalid	F9 invalid
		C2 FCMOVB ST(0), ST(2)	CA FCMOVE ST(0), ST(2)	D2 FCMOVBE ST(0), ST(2)	DA FCMOVU ST(0), ST(2)	E2 invalid	EA invalid	F2 invalid	FA invalid
		C3 FCMOVB ST(0), ST(3)	CB FCMOVE ST(0), ST(3)	D3 FCMOVBE ST(0), ST(3)	DB FCMOVU ST(0), ST(3)	E3 invalid	EB invalid	F3 invalid	FB invalid
		C4 FCMOVB ST(0), ST(4)	CC FCMOVE ST(0), ST(4)	D4 FCMOVBE ST(0), ST(4)	DC FCMOVU ST(0), ST(4)	E4 invalid	EC invalid	F4 invalid	FC invalid
		C5 FCMOVB ST(0), ST(5)	CD FCMOVE ST(0), ST(5)	D5 FCMOVBE ST(0), ST(5)	DD FCMOVU ST(0), ST(5)	E5 invalid	ED invalid	F5 invalid	FD invalid
		C6 FCMOVB ST(0), ST(6)	CE FCMOVE ST(0), ST(6)	D6 FCMOVBE ST(0), ST(6)	DE FCMOVU ST(0), ST(6)	E6 invalid	EE invalid	F6 invalid	FE invalid
		C7 FCMOVB ST(0), ST(7)	CF FCMOVE ST(0), ST(7)	D7 FCMOVBE ST(0), ST(7)	DF FCMOVU ST(0), ST(7)	E7 invalid	EF invalid	F7 invalid	FF invalid

Table A-15. x87 Opcodes and ModRM Extensions (continued)

Opcode	ModRM *mod* Field	ModRM *reg* Field							
		/0	/1	/2	/3	/4	/5	/6	/7
DB	!11	00–BF							
		FILD mem32int	FISTTP mem32int	FIST mem32int	FISTP mem32int	invalid	FLD mem80-real	invalid	FSTP mem80real
	11	C0 FCMOVNB ST(0), ST(0)	C8 FCMOVNE ST(0), ST(0)	D0 FCMOVNBE ST(0), ST(0)	D8 FCMOVNU ST(0), ST(0)	E0 reserved	E8 FUCOMI ST(0), ST(0)	F0 FCOMI ST(0), ST(0)	F8 invalid
		C1 FCMOVNB ST(0), ST(1)	C9 FCMOVNE ST(0), ST(1)	D1 FCMOVNBE ST(0), ST(1)	D9 FCMOVNU ST(0), ST(1)	E1 reserved	E9 FUCOMI ST(0), ST(1)	F1 FCOMI ST(0), ST(1)	F9 invalid
		C2 FCMOVNB ST(0), ST(2)	CA FCMOVNE ST(0), ST(2)	D2 FCMOVNBE ST(0), ST(2)	DA FCMOVNU ST(0), ST(2)	E2 FNCLEX	EA FUCOMI ST(0), ST(2)	F2 FCOMI ST(0), ST(2)	FA invalid
		C3 FCMOVNB ST(0), ST(3)	CB FCMOVNE ST(0), ST(3)	D3 FCMOVNBE ST(0), ST(3)	DB FCMOVNU ST(0), ST(3)	E3 FNINIT	EB FUCOMI ST(0), ST(3)	F3 FCOMI ST(0), ST(3)	FB invalid
		C4 FCMOVNB ST(0), ST(4)	CC FCMOVNE ST(0), ST(4)	D4 FCMOVNBE ST(0), ST(4)	DC FCMOVNU ST(0), ST(4)	E4 reserved	EC FUCOMI ST(0), ST(4)	F4 FCOMI ST(0), ST(4)	FC invalid
		C5 FCMOVNB ST(0), ST(5)	CD FCMOVNE ST(0), ST(5)	D5 FCMOVNBE ST(0), ST(5)	DD FCMOVNU ST(0), ST(5)	E5 invalid	ED FUCOMI ST(0), ST(5)	F5 FCOMI ST(0), ST(5)	FD invalid
		C6 FCMOVNB ST(0), ST(6)	CE FCMOVNE ST(0), ST(6)	D6 FCMOVNBE ST(0), ST(6)	DE FCMOVNU ST(0), ST(6)	E6 invalid	EE FUCOMI ST(0), ST(6)	F6 FCOMI ST(0), ST(6)	FE invalid
		C7 FCMOVNB ST(0), ST(7)	CF FCMOVNE ST(0), ST(7)	D7 FCMOVNBE ST(0), ST(7)	DF FCMOVNU ST(0), ST(7)	E7 invalid	EF FUCOMI ST(0), ST(7)	F7 FCOMI ST(0), ST(7)	FF invalid

Table A-15. x87 Opcodes and ModRM Extensions (continued)

Opcode	ModRM *mod* Field	ModRM *reg* Field							
		/0	/1	/2	/3	/4	/5	/6	/7
DC	!11	00–BF FADD mem64-real	FMUL mem64real	FCOM mem64real	FCOMP mem64real	FSUB mem64real	FSUBR mem64-real	FDIV mem64real	FDIVR mem64real
	11	C0 FADD ST(0), ST(0)	C8 FMUL ST(0), ST(0)	D0 reserved	D8 reserved	E0 FSUBR ST(0), ST(0)	E8 FSUB ST(0), ST(0)	F0 FDIVR ST(0), ST(0)	F8 FDIV ST(0), ST(0)
		C1 FADD ST(1), ST(0)	C9 FMUL ST(1), ST(0)	D1 reserved	D9 reserved	E1 FSUBR ST(1), ST(0)	E9 FSUB ST(1), ST(0)	F1 FDIVR ST(1), ST(0)	F9 FDIV ST(1), ST(0)
		C2 FADD ST(2), ST(0)	CA FMUL ST(2), ST(0)	D2 reserved	DA reserved	E2 FSUBR ST(2), ST(0)	EA FSUB ST(2), ST(0)	F2 FDIVR ST(2), ST(0)	FA FDIV ST(2), ST(0)
		C3 FADD ST(3), ST(0)	CB FMUL ST(3), ST(0)	D3 reserved	DB reserved	E3 FSUBR ST(3), ST(0)	EB FSUB ST(3), ST(0)	F3 FDIVR ST(3), ST(0)	FB FDIV ST(3), ST(0)
		C4 FADD ST(4), ST(0)	CC FMUL ST(4), ST(0)	D4 reserved	DC reserved	E4 FSUBR ST(4), ST(0)	EC FSUB ST(4), ST(0)	F4 FDIVR ST(4), ST(0)	FC FDIV ST(4), ST(0)
		C5 FADD ST(5), ST(0)	CD FMUL ST(5), ST(0)	D5 reserved	DD reserved	E5 FSUBR ST(5), ST(0)	ED FSUB ST(5), ST(0)	F5 FDIVR ST(5), ST(0)	FD FDIV ST(5), ST(0)
		C6 FADD ST(6), ST(0)	CE FMUL ST(6), ST(0)	D6 reserved	DE reserved	E6 FSUBR ST(6), ST(0)	EE FSUB ST(6), ST(0)	F6 FDIVR ST(6), ST(0)	FE FDIV ST(6), ST(0)
		C7 FADD ST(7), ST(0)	CF FMUL ST(7), ST(0)	D7 reserved	DF reserved	E7 FSUBR ST(7), ST(0)	EF FSUB ST(7), ST(0)	F7 FDIVR ST(7), ST(0)	FF FDIV ST(7), ST(0)

Table A-15. x87 Opcodes and ModRM Extensions (continued)

Opcode	ModRM *mod* Field	ModRM *reg* Field							
		/0	/1	/2	/3	/4	/5	/6	/7
DD	!11	00–BF							
		FLD mem64-real	FISTTP mem64int	FST mem64real	FSTP mem64real	FRSTOR mem98/108env	invalid	FNSAVE mem98/108env	FNSTSW mem16
	11	C0 FFREE ST(0)	C8 reserved	D0 FST ST(0)	D8 FSTP ST(0)	E0 FUCOM ST(0), ST(0)	E8 FUCOMP ST(0)	F0 invalid	F8 invalid
		C1 FFREE ST(1)	C9 reserved	D1 FST ST(1)	D9 FSTP ST(1)	E1 FUCOM ST(1), ST(0)	E9 FUCOMP ST(1)	F1 invalid	F9 invalid
		C2 FFREE ST(2)	CA reserved	D2 FST ST(2)	DA FSTP ST(2)	E2 FUCOM ST(2), ST(0)	EA FUCOMP ST(2)	F2 invalid	FA invalid
		C3 FFREE ST(3)	CB reserved	D3 FST ST(3)	DB FSTP ST(3)	E3 FUCOM ST(3), ST(0)	EB FUCOMP ST(3)	F3 invalid	FB invalid
		C4 FFREE ST(4)	CC reserved	D4 FST ST(4)	DC FSTP ST(4)	E4 FUCOM ST(4), ST(0)	EC FUCOMP ST(4)	F4 invalid	FC invalid
		C5 FFREE ST(5)	CD reserved	D5 FST ST(5)	DD FSTP ST(5)	E5 FUCOM ST(5), ST(0)	ED FUCOMP ST(5)	F5 invalid	FD invalid
		C6 FFREE ST(6)	CE reserved	D6 FST ST(6)	DE FSTP ST(6)	E6 FUCOM ST(6), ST(0)	EE FUCOMP ST(6)	F6 invalid	FE invalid
		C7 FFREE ST(7)	CF reserved	D7 FST ST(7)	DF FSTP ST(7)	E7 FUCOM ST(7), ST(0)	EF FUCOMP ST(7)	F7 invalid	FF invalid

Table A-15. x87 Opcodes and ModRM Extensions (continued)

Opcode	ModRM *mod* Field	ModRM *reg* Field							
		/0	/1	/2	/3	/4	/5	/6	/7
DE	!11	00–BF							
		FIADD mem16int	FIMUL mem16int	FICOM mem16int	FICOMP mem16int	FISUB mem16int	FISUBR mem16int	FIDIV mem16int	FIDIVR mem16int
	11	C0 FADDP ST(0), ST(0)	C8 FMULP ST(0), ST(0)	D0 reserved	D8 invalid	E0 FSUBRP ST(0), ST(0)	E8 FSUBP ST(0), ST(0)	F0 FDIVRP ST(0), ST(0)	F8 FDIVP ST(0), ST(0)
		C1 FADDP ST(1), ST(0)	C9 FMULP ST(1), ST(0)	D1 reserved	D9 FCOMPP	E1 FSUBRP ST(1), ST(0)	E9 FSUBP ST(1), ST(0)	F1 FDIVRP ST(1), ST(0)	F9 FDIVP ST(1), ST(0)
		C2 FADDP ST(2), ST(0)	CA FMULP ST(2), ST(0)	D2 reserved	DA invalid	E2 FSUBRP ST(2), ST(0)	EA FSUBP ST(2), ST(0)	F2 FDIVRP ST(2), ST(0)	FA FDIVP ST(2), ST(0)
		C3 FADDP ST(3), ST(0)	CB FMULP ST(3), ST(0)	D3 reserved	DB invalid	E3 FSUBRP ST(3), ST(0)	EB FSUBP ST(3), ST(0)	F3 FDIVRP ST(3), ST(0)	FB FDIVP ST(3), ST(0)
		C4 FADDP ST(4), ST(0)	CC FMULP ST(4), ST(0)	D4 reserved	DC invalid	E4 FSUBRP ST(4), ST(0)	EC FSUBP ST(4), ST(0)	F4 FDIVRP ST(4), ST(0)	FC FDIVP ST(4), ST(0)
		C5 FADDP ST(5), ST(0)	CD FMULP ST(5), ST(0)	D5 reserved	DD invalid	E5 FSUBRP ST(5), ST(0)	ED FSUBP ST(5), ST(0)	F5 FDIVRP ST(5), ST(0)	FD FDIVP ST(5), ST(0)
		C6 FADDP ST(6), ST(0)	CE FMULP ST(6), ST(0)	D6 reserved	DE invalid	E6 FSUBRP ST(6), ST(0)	EE FSUBP ST(6), ST(0)	F6 FDIVRP ST(6), ST(0)	FE FDIVP ST(6), ST(0)
		C7 FADDP ST(7), ST(0)	CF FMULP ST(7), ST(0)	D7 reserved	DF invalid	E7 FSUBRP ST(7), ST(0)	EF FSUBP ST(7), ST(0)	F7 FDIVRP ST(7), ST(0)	FF FDIVP ST(7), ST(0)

Table A-15. x87 Opcodes and ModRM Extensions (continued)

Opcode	ModRM *mod* Field	ModRM *reg* Field							
		/0	/1	/2	/3	/4	/5	/6	/7
DF	!11	00–BF							
		FILD mem16int	FISTTP mem16int	FIST mem16int	FISTP mem16int	FBLD mem80dec	FILD mem64int	FBSTP mem80dec	FISTP mem64int
	11	C0 reserved	C8 reserved	D0 reserved	D8 reserved	E0 FNSTSW AX	E8 FUCOMIP ST(0), ST(0)	F0 FCOMIP ST(0), ST(0)	F8 invalid
		C1 reserved	C9 reserved	D1 reserved	D9 reserved	E1 invalid	E9 FUCOMIP ST(0), ST(1)	F1 FCOMIP ST(0), ST(1)	F9 invalid
		C2 reserved	CA reserved	D2 reserved	DA reserved	E2 invalid	EA FUCOMIP ST(0), ST(2)	F2 FCOMIP ST(0), ST(2)	FA invalid
		C3 reserved	CB reserved	D3 reserved	DB reserved	E3 invalid	EB FUCOMIP ST(0), ST(3)	F3 FCOMIP ST(0), ST(3)	FB invalid
		C4 reserved	CC reserved	D4 reserved	DC reserved	E4 invalid	EC FUCOMIP ST(0), ST(4)	F4 FCOMIP ST(0), ST(4)	FC invalid
		C5 reserved	CD reserved	D5 reserved	DD reserved	E5 invalid	ED FUCOMIP ST(0), ST(5)	F5 FCOMIP ST(0), ST(5)	FD invalid
		C6 reserved	CE reserved	D6 reserved	DE reserved	E6 invalid	EE FUCOMIP ST(0), ST(6)	F6 FCOMIP ST(0), ST(6)	FE invalid
		C7 reserved	CF reserved	D7 reserved	DF reserved	E7 invalid	EF FUCOMIP ST(0), ST(7)	F7 FCOMIP ST(0), ST(7)	FF invalid

A.1.4 rFLAGS Condition Codes for x87 Opcodes

Table A-16 shows the rFLAGS condition codes specified by the opcode and ModRM bytes of the FCMOV*cc* instructions.

Table A-16. rFLAGS Condition Codes for FCMOV*cc*

Opcode (hex)	ModRM *mod* Field	ModRM *reg* Field	rFLAGS Value	cc Mnemonic	Condition
DA	11	000	CF = 1	B	Below
		001	ZF = 1	E	Equal
		010	CF = 1 or ZF = 1	BE	Below or Equal
		011	PF = 1	U	Unordered
DB		000	CF = 0	NB	Not Below
		001	ZF = 0	NE	Not Equal
		010	CF = 0 and ZF = 0	NBE	Not Below or Equal
		011	PF = 0	NU	Not Unordered

A.1.5 Extended Instruction Opcode Maps

The following sections present the VEX and the XOP extended instruction opcode maps. The VEX.map_select field of the three-byte VEX encoding escape sequence selects VEX opcode maps: 01h, 02h, or 03h. The two-byte VEX encoding escape sequence implicitly selects the VEX map 01h.

The XOP.map_select field selects between the three XOP maps: 08h, 09h or 0Ah.

VEX Opcode Maps. Tables A-17 – A-23 below present the VEX opcode maps and Table A-24 on page 542 presents the VEX opcode groups.

Table A-17. VEX Opcode Map 1, Low Nibble = [0h:7h]

Opcode	x0	x1	x2	x3	x4	x5	x6	x7
00	. . .							
1x	VMOVUPS[2] Vpsx, Wpsx	VMOVUPS[2] Wpsx, Vpsx	VMOVLPS Vps, Hps, Mq VMOVHLPS Vps, Hps, Ups	VMOVLPS Mq, Vps	VUNPCKLPS[2] Vpsx, Hpsx, Wpsx	VUNPCKHPS[2] Vpsx, Hpsx, Wpsx	VMOVHPS Vps, Hps, Mq VMOVLHPS Vps, Hps, Ups	VMOVHPS Mq, Vps
	VMOVUPD[2] Vpdx, Wpdx	VMOVUPD[2] Wpdx, Vpdx	VMOVLPD Vo, Ho, Mq	VMOVLPD Mq, Vo	VUNPCKLPD[2] Vpdx, Hpdx, Wpdx	VUNPCKHPD[2] Vpdx, Hpdx, Wpdx	VMOVHPD Vpd, Hpd, Mq	VMOVHPD Mq, Vpd
	VMOVSS[3] Vss, Md VMOVSS Vss, Hss, Uss	VMOVSS[3] Md, Vss VMOVSS Uss, Hss, Vss	VMOVSLDUP[2] Vpsx, Wpsx				VMOVSHDUP[2] Vpsx, Wpsx	
	VMOVSD[3] Vsd, Mq VMOVSD Vsd, Hsd, Usd	VMOVSD[3] Mq, Vsd VMOVSD Usd, Hsd, Vsd	VMOVDDUP Vo, Wq (L=0) Vdo, Wdo (L=1)					
2x–4x	. . .							
5x	VMOVMSKPS[2] Gy, Upsx	VSQRTPS[2] Vpsx, Wpsx	VRSQRTPS[2] Vpsx, Wpsx	VRCPPS[2] Vpsx, Wpsx	VANDPS[2] Vpsx, Hpsx, Wpsx	VANDNPS[2] Vpsx, Hpsx, Wpsx	VORPS[2] Vpsx, Hpsx, Wpsx	VXORPS[2] Vpsx, Hpsx, Wpsx
	VMOVMSKPD[2] Gy, Updx	VSQRTPD[2] Vpdx, Wpdx			VANDPD[2] Vpdx, Hpdx, Wpdx	VANDNPD[2] Vpdx, Hpdx, Wpdx	VORPD[2] Vpdx, Hpdx, Wpdx	VXORPD[2] Vpdx, Hpdx, Wpdx
		VSQRTSS[3] Vo, Ho, Wss	VRSQRTSS[3] Vo, Ho, Wss	VRCPSS[3] Vo, Ho, Wss				
		VSQRTSD[3] Vo, Ho, Wsd						
6x								
	VPUNPCKLBW[2] Vpbx, Hpbx, Wpbx	VPUNPCKLWD[2] Vpwx, Hpwx, Wpwx	VPUNPCKLDQ[2] Vpdwx, Hpdwx, Wpdwx	VPACKSSWB[2] Vpkx, Hpix, Wpix	VPCMPGTB[2] Vpbx, Hpkx, Wpkx	VPCMPGTW[2] Vpwx, Hpix, Wpix	VPCMPGTD[2] Vpdwx, Hpjx, Wpjx	VPACKUSWB[2] Vpkx, Hpix, Wpix
7x								VZEROUPPER (L=0) VZEROALL (L=1)
	VPSHUFD[2] Vpdwx, Wpdwx, Ib	VEX group #12	VEX group #13	VEX group #14	VPCMPEQB[2] Vpbx, Hpkx, Wpkx	VPCMPEQW[2] Vpwx, Hpix, Wpix	VPCMPEQD[2] Vpdwx, Hpjx, Wpjx	
	VPSHUFHW[2] Vpwx, Wpwx, Ib							
	VPSHUFLW[2] Vpwx, Wpwx, Ib							
8x–Bx	. . .							
Cx			VCMPccPS[1] Vpdw, Hps, Wps, Ib				VSHUFPS[2] Vpsx, Hpsx, Wpsx, Ib	
			VCMPccPD[1] Vpqw, Hpd, Wpd, Ib		VPINSRW Vpw, Hpw, Mw, Ib Vpw, Hpw, Rd, Ib	VPEXTRW Gw, Upw, Ib	VSHUFPD[2] Vpdx, Hpdx, Wpdx, Ib	
			VCMPccSS[1] Vd, Hss, Wss, Ib					
			VCMPccSD[1] Vq, Hsd, Wsd, Ib					

Note 1: The condition codes are: EQ, LT, LE, UNORD, NEQ, NLT, NLE, and ORD; encoded as [00:07h] using Ib.
VEX encoding adds: EQ_UQ, NGE, NGT, FALSE, NEQ_OQ, GE, GT, TRUE [08:0Fh];
EQ_OS, LT_OQ, LE_OQ, UNORD_S, NEQ_US, NLT_UQ, NLE_UQ, ORD_S [10h:17h]; and
EQ_US, NGE_UQ, NGT_UQ, FALSE_OS, NEQ_OS, GE_OQ, GT_OQ, TRUE_US [18:1Fh].
Note 2: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L = 0, size is 128 bits; when L = 1, size is 256 bits.
Note 3: Operands are scalars. VEX.L bit is ignored.

Table A-18. VEX Opcode Map 1, Low Nibble = [0h:7h] Continued

/EX.pp	Opcode	x0	x1	x2	x3	x4	x5	x6	x7
00	Dx								
01		VADDSUBPD[2] Vpdx, Hpdx, Wpdx	VPSRLW[2] Vpwx, Hpwx, Wx	VPSRLD[2] Vpdwx, Hpdwx, Wx	VPSRLQ[2] Vpqwx, Hpqwx, Wx	VPADDQ[2] Vpq, Hpq, Wpq	VPMULLW[2] Vpix, Hpix, Wpix	VMOVQ Wq, Vq (VEX.L=1)	VPMOVMSKB[2] Gy, Upbx
10									
11		VADDSUBPS[2] Vpsx, Hpsx, Wpsx							
00	Ex								
01		VPAVGB[2] Vpkx, Hpkx, Wpkx	VPSRAW[2] Vpwx, Hpwx, Wx	VPSRAD[2] Vpdwx, Hpdwx, Wx	VPAVGW[2] Vpix, Hpix, Wpix	VPMULHUW[2] Vpi, Hpi, Wpi	VPMULHW Vpi, Hpi, Wpi	VCVTTPD2DQ[2] Vpjx, Wpdx	VMOVNTDQ Mo, Vo (L=0) Mdo, Vdo (L=1)
10								VCVTDQ2PD[2] Vpdx, Wpjx	
11								VCVTPD2DQ[2] Vpjx, Wpdx	
00	Fx								
01			VPSLLW[2] Vpwx, Hpwx, Wo.qx	VPSLLD[2] Vpdwx, Hpdwx, Wo.qx	VPSLLQ[2] Vpqwx, Hpqwx, Wo.qx	VPMULUDQ[2] Vpqx, Hpjx, Wpjx	VPMADDWD[2] Vpjx, Hpix, Wpix	VPSADBW[2] Vpix, Hpkx, Wpkx	VMASKMOVDQU Vpb, Upb
10									
11		VLDDQU Vo, Mo (L=0) Vdo, Mdo (L=1)							

Note 1: The condition codes are: EQ, LT, LE, UNORD, NEQ, NLT, NLE, and ORD; encoded as [00:07h] using Ib.
VEX encoding adds: EQ_UQ, NGE, NGT, FALSE, NEQ_OQ, GE, GT, TRUE [08:0Fh];
EQ_OS, LT_OQ, LE_OQ, UNORD_S, NEQ_US, NLT_UQ, NLE_UQ, ORD_S [10h:17h]; and
EQ_US, NGE_UQ, NGT_UQ, FALSE_OS, NEQ_OS, GE_OQ, GT_OQ, TRUE_US [18:1Fh].
Note 2: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L = 0, size is 128 bits; when L = 1, size is 256 bits.
Note 3: Operands are scalars. VEX.L bit is ignored.

Table A-19. VEX Opcode Map 1, Low Nibble = [8h:Fh]

VEX.pp	Opcode	x8	x9	xA	xB	xC	xD	xE	xF
. . .	0x-1x	. . .							
00	2x	VMOVAPS[1] Vpsx, Wpsx	VMOVAPS[1] Wpsx, Vpsx		VMOVNTPS[1] Mpsx, Vpsx			VUCOMISS[2] Vss, Wss	VCOMISS[2] Vss, Wss
01		VMOVAPD[1] Vpdx, Wpdx	VMOVAPD[1] Wpdx, Vpdx		VMOVNTPD[1] Mpdx, Vpdx			VUCOMISD[2] Vsd, Wsd	VCOMISD[2] Vsd, Wsd
10				VCVTSI2SS[2] Vo, Ho, Ey		VCVTTSS2SI[2] Gy, Wss	VCVTSS2SI[2] Gy, Wss		
11				VCVTSI2SD[2] Vo, Ho, Ey		VCVTTSD2SI[2] Gy, Wsd	VCVTSD2SI[2] Gy, Wsd		
. . .	3x-4x	. . .							
00	5x	VADDPS[1] Vpsx, Hpsx, Wpsx	VMULPS[1] Vpsx, Hpsx, Wpsx	VCVTPS2PD[1] Vpdx, Wpsx	VCVTDQ2PS[1] Vpsx, Wpjx	VSUBPS[1] Vpsx, Hpsx, Wpsx	VMINPS[1] Vpsx, Hpsx, Wpsx	VDIVPS[1] Vpsx, Hpsx, Wpsx	VMAXPS[1] Vpsx, Hpsx, Wpsx
01		VADDPD[1] Vpdx, Hpdx, Wpdx	VMULPD[1] Vpdx, Hpdx, Wpdx	VCVTPD2PS[1] Vpsx, Wpdx	VCVTPS2DQ[1] Vpjx, Wpsx	VSUBPD[1] Vpdx, Hpdx, Wpdx	VMINPD[1] Vpdx, Hpdx, Wpdx	VDIVPD[1] Vpdx, Hpdx, Wpdx	VMAXPD[1] Vpdx, Hpdx, Wpdx
10		VADDSS[2] Vss, Hss, Wss	VMULSS[2] Vss, Hss, Wss	VCVTSS2SD[2] Vo, Ho, Wss	VCVTTPS2DQ[1] Vpjx, Wpsx	VSUBSS[2] Vss, Hss, Wss	VMINSS[2] Vss, Hss, Wss	VDIVSS[2] Vss, Hss, Wss	VMAXSS[2] Vss, Hss, Wss
11		VADDSD[2] Vsd, Hsd, Wsd	VMULSD[2] Vsd, Hsd, Wsd	VCVTSD2SS[2] Vo, Ho, Wsd		VSUBSD[2] Vsd, Hsd, Wsd	VMINSD[2] Vsd, Hsd, Wsd	VDIVSD[2] Vsd, Hsd, Wsd	VMAXSD[2] Vsd, Hsd, Wsd
00	6x								
01		VPUNPCKHBW[1] Vpbx, Hpbx, Wpbx	VPUNPCKHWD[1] Vpwx, Hpwx, Wpwx	VPUNPCKHDQ[1] Vpdwx, Hpdwx, Wpdwx	VPACKSSDW[1] Vpix, Hpjx, Wpjx	VPUNPCKLQDQ[1] Vpqwx, Hpqwx, Wpqwx	VPUNPCKHQDQ[1] Vpqwx, Hpqwx, Wpqwx	VMOVD VMOVQ Vo, Ey (VEX.L=0)	VMOVDQA[1] Vpqwx, Wpqwx
10									VMOVDQU[1] Vpqwx, Wpqwx
11									
00	7x								
01						VHADDPD[1] Vpdx, Hpdx, Wpdx	VHSUBPD[1] Vpdx, Hpdx, Wpdx	VMOVD VMOVQ Ey, Vo (VEX.L=1)	VMOVDQA[1] Wpqwx, Vpqwx
10								VMOVQ Vq, Wq (VEX.L=0)	VMOVDQU[1] Wpqwx, Vpqwx
11						VHADDPS[1] Vpsx, Hpsx, Wpsx	VHSUBPS[1] Vpsx, Hpsx, Wpsx		
. . .	8x-9x	. . .							
n/a	Ax							VEX group #15	
. . .	Bx-Cx	. . .							
00	Dx								
01		VPSUBUSB[1] Vpkx, Hpkx, Wpkx	VPSUBUSW[1] Vpix, Hpix, Wpix	VPMINUB[1] Vpkx, Hpkx, Wpkx	VPAND[1] Vx, Hx, Wx	VPADDUSB[1] Vpkx, Hpkx, Wpkx	VPADDUSW[1] Vpix, Hpix, Wpix	VPMAXUB[1] Vpkx, Hpkx, Wpkx	VPANDN[1] Vx, Hx, Wx
00	Ex								
01		VPSUBSB[1] Vpkx, Hpkx, Wpkx	VPSUBSW[1] Vpix, Hpix, Wpix	VPMINSW[1] Vpix, Hpix, Wpix	VPOR[1] Vx, Hx, Wx	VPADDSB[1] Vpkx, Hpkx, Wpkx	VPADDSW[1] Vpix, Hpix, Wpix	VPMAXSW[1] Vpix, Hpix, Wpix	VPXOR[1] Vx, Hx, Wx
00	Fx								
01		VPSUBB[1] Vpkx, Hpkx, Wpkx	VPSUBW[1] Vpix, Hpix, Wpix	VPSUBD[1] Vpxj, Hpjx, Wpjx	VPSUBQ[1] Vpqx, Hpqx, Wpqx	VPADDB[1] Vpkx, Hpkx, Wpkx	VPADDW[1] Vpix, Hpix, Wpix	VPADDD[1] Vpjx, Hpjx, Wpjx	

Note 1: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L = 0, size is 128 bits; when L = 1, size is 256 bits.
Note 2: Operands are scalars. VEX.L bit is ignored.

Table A-20. VEX Opcode Map 2, Low Nibble = [0h:7h]

VEX.pp	Opcode	x0	x1	x2	x3	x4	x5	x6	x7
01	0x	VPSHUFB[1] Vpbx, Hpbx, Wpbx	VPHADDW[1] Vpix, Hpix, Wpix	VPHADDD[1] Vpjx, Hpjx, Wpjx	VPHADDSW[1] Vpix, Hpix, Wpix	VPMADDUBSW[1] Vpix, Hpkx, Wpkx	VPHSUBW[1] Vpix, Hpix, Wpix	VPHSUBD[1] Vpjx, Hpjx, Wpjx	VPHSUBSW[1] Vpix, Hpix, Wpix
01	1x				VCVTPH2PS[1] Vpsx, Wphx			VPERMPS Vps, Hd, Wps	VPTEST[1,4] Vx, Wx
01	2x	VPMOVSXBW[1] Vpix, Wpkx	VPMOVSXBD[1] Vpjx, Wpkx	VPMOVSXBQ[1] Vpqx, Wpkx	VPMOVSXWD[1] Vpjx, Wpix	VPMOVSXWQ[1] Vpqx, Wpix	VPMOVSXDQ[1] Vpqx, Wpjx		
01	3x	VPMOVZXBW[1] Vpix, Wpkx	VPMOVZXBD[1] Vpjx, Wpkx	VPMOVZXBQ[1] Vpqx, Wpkx	VPMOVZXWD[1] Vpjx, Wpix	VPMOVZXWQ[1] Vpqx, Wpix	VPMOVZXDQ[1] Vpqx, Wpjx	VPERMD Vd, Hd, Wd	VPCMPGTQ[1] Vpqx, Hpqx, Wpqx
01	4x	VPMULLD[1] Vpjx, Hpjx, Wpxj	VPHMINPOSUW Vo, Wpi				VPSRLV- D[1] Vx, Hx, Wx (W=0) Q[1] Vx, Hx, Wx (W=1)	VPSRAVD[1] Vpdwx, Hpdwx, Wpdwx	VPSLLV- D[1] Vx, Hx, Wx (W=0) Q[1] Vx, Hx, Wx (W=1)
...	5x-8x	...							
01	9x	[5]VPGATHERD- D[1] Vx, M*d, Hpdw (W=0) Q[1] Vx, M*q, Hpqwx (W=1)	[5]VPGATHERQ- D[1] Vx, M*d, Hpdw (W=0) Q[1] Vx, M*q, Hpqw (W=1)	[5]VGATHERD- PS[1] Vx,M*ps,Hpsx (W=0) PD[1] Vx,M*pd,Hpdx (W=1)	[5]VGATHERQ- PS[1] Vx,M*ps,Hps (W=0) PD[1] Vx,M*pd,Hpdx (W=1)			[2]VFMADDSUB132- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	[3]VFMSUBADD132- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)
01	Ax							VFMADDSUB213- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFMSUBADD213- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)
01	Bx							VFMADDSUB231- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFMSUBADD231- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)
...	Cx-Ex	...							
00	Fx			ANDN Gy, By, Ey	VEX group #17		BZHI Gy, Ey, By		BEXTR Gy, Ey, By
01							PEXT Gy, By, Ey		SHLX Gy, Ey, By
10									SARX Gy, Ey, By
11							PDEP Gy, By, Ey	MULX Gy, By, Ey	SHRX Gy, Ey, By

Note 1: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L = 0, size is 128 bits; when L = 1, size is 256 bits.
Note 2: For all VFMADDSUBnnnPS instructions, the data type is packed single-precision floating point.
For all VFMADDSUBnnnPD instructions, the data type is packed double-precision floating point.
Note 3: For all VFMSUBADDnnnPS instructions, the data type is packed single-precision floating point.
For all VFMSUBADDnnnPD instructions, the data type is packed double-precision floating point.
Note 4: Operands are treated a bit vectors.
Note 5: Uses VSIB addressing mode.

Table A-21. VEX Opcode Map 2, Low Nibble = [8h:Fh]

VEX.pp	Opcode	x8	x9	xA	xB	xC	xD	xE	xF
01	0x	VPSIGNB[1] Vpkx, Hpkx, Wpkx	VPSIGNW[1] Vpi, Hpi, Wpi	VPSIGND[1] Vpjx, Hpjx, Wpjx	VPMULHRSW[1] Vpix, Hpix, Wpix	VPERMILPS[1] Vpsx, Hpsx, Wpdwx	VPERMILPD[1] Vpdx, Hpdx, Wpqwx	VTESTPS[1] Vpsx, Wpsx	VTESTPD[1] Vpdx, Wpdx
01	1x	VBROADCASTSS[1] Vps, Wss	VBROADCASTSD Vpd, Wsd (VEX.L=1)	VBROADCASTF128 Vdo, Mo (VEX.L=1)		VPABSB[1] Vpkx, Wpkx	VPABSW[1] Vpix, Wpix	VPABSD[1] Vpjx, Wpjx	
01	2x	VPMULDQ[1] Vpqx, Hpjx , Wpjx	VPCMPEQQ[1] Vpqx, Hpqx, Wpqx	VMOVNTDQA[1] Vx, Mx	VPACKUSDW[1] Vpix, Hpjx, Wpjx	VMASKMOVPS[1] Vpsx, Hx, Mpsx	VMASKMOVPD[1] Vpdx, Hx, Mpdx	VMASKMOVPS[1] Mpsx, Hx, Vpsx	VMASKMOVPD[1] Mpdx, Hx, Vpdx
01	3x	VPMINSB[1] Vpkx, Hpkx, Wpkx	VPMINSD[1] Vpjx, Hpjx, Wpjx	VPMINUW[1] Vpix, Hpix, Wpix	VPMINUD[1] Vpjx, Hpjx, Wpjx	VPMAXSB[1] Vpkx, Hpkx, Wpkx	VPMAXSD[1] Vpxj, Hpjx, Wpjx	VPMAXUW[1] Vpix, Hpix, Wpix	VPMAXUD[1] Vpjx, Hpjx, Wpjx
...	4x	...							
01	5x	VPBROADCASTD[1] Vx, Wd	VPBROADCASTQ[1] Vx, Wq	VBROADCASTI128 Vdo, Mo					
...	6x	...							
01	7x	VPBROADCASTB[1] Vx, Wb	VPBROADCASTW[1] Vx, Ww						
01	8x					VPMASKMOV-D[1] Vx, Hx, Mx (W=0) Q[1] Vx, Hx, Mx (W=1)		VPMASKMOV-D[1] Mx, Hx, Vx (W=0) Q[1] Mx, Hx, Vx (W=1)	
01	9x	[3]VFMADD132- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFMADD132- SS[2] Vo,Ho,Wd (W=0) SD[2] Vo,Ho,Wq (W=1)	[4]VFMSUB132- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFMSUB132- SS[2] Vo,Ho,Wd (W=0) SD[2] Vo,Ho,Wq (W=1)	VFNMADD132- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFNMADD132- SS[2] Vo,Ho,Wd (W=0) SD[2] Vo,Ho,Wq (W=1)	VFNMSUB132- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFNMSUB132- SS[2] Vo,Ho,Wd (W=0) SD[2] Vo,Ho,Wq (W=1)
01	Ax	VFMADD213- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFMADD213- SS[2] Vo,Ho,Wd (W=0) SD[2] Vo,Ho,Wq (W=1)	VFMSUB213- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFMSUB213- SS[2] Vo,Ho,Wd (W=0) SD[2] Vo,Ho,Wq (W=1)	VFNMADD213- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFNMADD213- SS[2] Vo,Ho,Wd (W=0) SD[2] Vo,Ho,Wq (W=1)	VFNMSUB213- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFNMSUB213- SS[2] Vo,Ho,Wd (W=0) SD[2] Vo,Ho,Wq (W=1)
01	Bx	VFMADD231- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFMADD231- SS[2] Vo,Ho,Wd (W=0) SD[2] Vo,Ho,Wq (W=1)	VFMSUB231- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFMSUB231- SS[2] Vo,Ho,Wd (W=0) SD[2] Vo,Ho,Wq (W=1)	VFNMADD231- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFNMADD231- SS[2] Vo,Ho,Wd (W=0) SD[2] Vo,Ho,Wq (W=1)	VFNMSUB231- PS[1] Vx,Hx,Wx (W=0) PD[1] Vx,Hx,Wx (W=1)	VFNMSUB231- SS[2] Vo,Ho,Wd (W=0) SD[2] Vo,Ho,Wq (W=1)
...	Cx	...							
01	Dx				VAESIMC Vo, Wo	VAESENC Vo, Ho, Wo	VAESENCLAST Vo, Ho, Wo	VAESDEC Vo, Ho, Wo	VAESDECLAST Vo, Ho, Wo
...	Ex-Fx	...							

Note 1: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L = 0, size is 128 bits; when L = 1, size is 256 bits.
Note 2: Operands are scalars. VEX.L bit is ignored.
Note 3: For all VFMADDnnnPS instructions, the data type is packed single-precision floating point.
For all VFMADDnnnPD instructions, the data type is packed double-precision floating point.
Note 4: For all VFMSUBnnnPS instructions, the data type is packed single-precision floating point.
For all VFMSUBnnnPD instructions, the data type is packed double-precision floating point.

Table A-22. VEX Opcode Map 3, Low Nibble = [0h:7h]

VEX.pp	Nibble	x0	x1	x2	x3	x4	x5	x6	x7
00	0x								
01		VPERMQ Vq, Wq, Ib	VPERMPD Vpd, Wpd, Ib	VPBLENDD[1] Vpdwx, Hpdwx, Wpdwx, Ib		VPERMILPS[1] Vpsx, Wpsx, Ib	VPERMILPD[1] Vpdx, Wpdx, Ib	VPERM2F128 Vdo, Ho, Wo, Ib (VEX.L=1)	
00	1x								
01						VPEXTRB Mb, Vpb, Ib VPEXTRB Ry, Vpb, Ib	VPEXTRW Mw, Vpw, Ib VPEXTRW Ry, Vpw, Ib	VPEXTRD Ed, Vpdw, Ib VPEXTRQ Eq, Vpqw, Ib	VEXTRACTPS Mss, Vps, Ib VEXTRACTPS Rss, Vps, Ib
00	2x								
01		VPINSRB Vpb, Hpb, Wb, Ib	VINSERTPS Vps, Hps, Ups/Md,	VPINSRD Vpdw, Hpdw, Ed, Ib (W=0) VPINSRQ Vpdw, Hpqw, Eq, Ib (W=1)					
. . .	3x	. . .							
00	4x								
01		VDPPS[1] Vpsx, Hpsx, Wpsx, Ib	VDPPD Vpd, Hpd, Wpd, Ib	VMPSADBW[1] Vpix, Hpkx, Wpkx, Ib		VPCLMULQDQ Vo, Hpq, Wpq, Ib		VPERM2I128 Vo, Ho, Wo, ib	
. . .	5x	. . .							
00	6x								
01		VPCMPESTRM Vo, Wo, Ib	VPCMPESTRI Vo, Wo, Ib	VPCMPISTRM Vo, Wo, Ib	VPCMPISTRI Vo, Wo, Ib				
. . .	7x-Ex	. . .							
10	Fx								
11		RORX Gy, Ey, ib							

Note 1: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L=0, size is 128 bits; when L=1, size is 256 bits.

Table A-23. VEX Opcode Map 3, Low Nibble = [8h:Fh]

VEX.pp	Opcode	x8	x9	xA	xB	xC	xD	xE	xF
01	0x	VROUNDPS[1] Vpsx, Wpsx, Ib	VROUNDPD[1] Vpdx, Wpdx, Ib	VROUNDSS Vss, Hss, Wss, Ib	VROUNDSD Vsd, Hsd, Wsd, Ib	VBLENDPS[1] Vpsx, Hpsx, Wpsx, Ib	VBLENDPD[1] Vpdx, Hpdx, Wpdx, Ib	VPBLENDW[1] Vpwx, Hpwx, Wpwx, Ib	VPALIGNR[1] Vpbx, Hpbx, Wpbx, Ib
01	1x	VINSERTF128 Vdo, Hdo, Wo, Ib	VEXTRACTF128 Wo, Vdo, Ib				VCVTPS2PH[1] Wph, Vps, Ib		
. . .	2x	. . .							
01	3x	VINSERTI128 Vdo, Hdo, Wo, Ib	VEXTRACTI128 Wo, Vdo, Ib						
01	4x	VPERMILzz2PS[1,2] Vpsx, Hpsx, Wpsx, Lpsx, Ib (W=0) Vpsx, Hpsx, Lpsx, Wpsx, Ib (W=1)	VPERMILzz2PD[1,2] Vpdx, Hpdx, Wpdx Lpdx, Ib (W=0) Vpdx, Hpdx, Lpdx, Wpdx, Ib (W=1)	VBLENDVPS[1] Vpsx, Hpsx, Wpsx, Lpdx	VBLENDVPD[1] Vpdx, Hpdx, Wpdx, Lpdx	VPBLENDVB[1] Vpbx, Hpbx, Wpbx, Lx			
01	5x					VFMADDSUBPS[1] Vpsx, Lpsx, Wpsx, Hpsx (W=0) Vpsx, Lpsx, Hpsx, Wpsx (W=1)	VFMADDSUBPD[1] Vpdx, Lpdx, Wpdx, Hpdx (W=0) Vpdx, Lpdx, Hpdx, Wpdx (W=1)	VFMSUBADDPS[1] Vpsx, Lpsx, Wpsx, Hpsx (W=0) Vpsx, Lpsx, Hpsx, Wpsx (W=1)	VFMSUBADDPD[1] Vpdx, Lpdx, Wpdx, Hpdx (W=0) Vpdx, Lpdx, Hpdx, Wpdx (W=1)
01	6x	VFMADDPS[1] Vpsx, Lpsx, Wpsx, Hpsx (W=0) Vpsx, Lpsx, Hpsx, Wpsx (W=1)	VFMADDPD[1] Vpdx, Lpdx, Wpdx, Hpdx (W=0) Vpdx, Lpdx, Hpdx, Wpdx (W=1)	VFMADDSS Vss, Lss, Wss, Hss (W=0) Vss, Lss, Hss, Wss (W=1)	VFMADDSD Vsd, Lsd, Wsd, Hsd (W=0) Vsd, Lsd, Hsd, Wsd (W=1)	VFMSUBPS[1] Vpsx, Lpsx, Wpsx, Hpsx (W=0) Vpsx, Lpsx, Hpsx, Wpsx (W=1)	VFMSUBPD[1] Vpdx, Lpdx, Wpdx, Hpdx (W=0) Vpdx, Lpdx, Hpdx, Wpdx (W=1)	VFMSUBSS Vss, Lss, Wss, Hss (W=0) Vss, Lss, Hss, Wss (W=1)	VFMSUBSD Vsd, Lsd, Wsd, Hsd (W=0) Vsd, Lsd, Hsd, Wsd (W=1)
01	7x	VFNMADDPS[1] Vpsx, Lpsx, Wpsx, Hpsx (W=0) Vpsx, Lpsx, Hpsx, Wpsx (W=1)	VFNMADDPD[1] Vpdx, Lpdx, Wpdx, Hpdx (W=0) Vpdx, Lpdx, Hpdx, Wpdx (W=1)	VFNMADDSS Vss, Lss, Wss, Hss (W=0) Vss, Lss, Hss, Wss (W=1)	VFNMADDSD Vsd, Lsd, Wsd, Hsd (W=0) Vsd, Lsd, Hsd, Wsd (W=1)	VFNMSUBPS[1] Vpsx, Lpsx, Wpsx, Hpsx (W=0) Vpsx, Lpsx, Hpsx, Wpsx (W=1)	VFNMSUBPD[1] Vpdx, Lpdx, Wpdx, Hpdx (W=0) Vpdx, Lpdx, Hpdx, Wpdx (W=1)	VFNMSUBSS Vss, Lss, Wss, Hss (W=0) Vss, Lss, Hss, Wss (W=1)	VFNMSUBSD Vsd, Lsd, Wsd, Hsd (W=0) Vsd, Lsd, Hsd, Wsd (W=1)
. . .	8x-Cx	. . .							
01	Dx								VAESKEYGEN-ASSIST Vo, Wo, Ib
…	Ex-Fx	…							

Note 1: Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L=0, size is 128 bits; when L=1, size is 256 bits.

Note 2: The zero match codes are TD, TD (alias), MO, and MZ. They are encoded as the zzzz field of the Ib, using 0...3h.

Table A-24. VEX Opcode Groups

Group			ModRM Byte							
Number	VEX Map, Opcode	VEX.pp	xx000xxx	xx001xxx	xx010xxx	xx011xxx	xx100xxx	xx101xxx	xx110xxx	xx111xxx
12	1 71	01			VPSRLW[1] Hpwx, Upwx, Ib		VPSRAW[1] Hpwx, Upwx, Ib		VPSLLW[1] Hpwx, Upwx, Ib	
13	1 72	01			VPSRLD[1] Hpdwx, Updwx, Ib		VPSRAD[1] Hpdwx, Updwx, Ib		VPSLLD[1] Hpdwx, Updwx, Ib	
14	1 73	01			VPSRLQ[1] Hpqwx, Upqwx, Ib	VPSRLDQ[1] Hpbx, Upbx, Ib			VPSLLQ[1] Hpqwx, Upqwx, Ib	VPSLLDQ[1] Hpbx, Upbx, Ib
15	1 AE	00			VLDMXCSR Md	VSTMXCSR Md				
17	2 F3	00		BLSR By, Ey	BLSMSK By, Ey	BLSI By, Ey				

Note: 1. Supports both 128 bit and 256 bit vector sizes. Vector size is specified using the VEX.L bit. When L = 0, size is 128 bits; when L = 1, size is 256 bits.

XOP Opcode Maps. Tables A-25 – A-30 below present the XOP opcode maps and Table A-31 on page 544 presents the VEX opcode groups.

Table A-25. XOP Opcode Map 8h, Low Nibble = [0h:7h]

XOP.pp	Opcode	x0	x1	x2	x3	x4	x5	x6	x7
. . .	0x-7x	. . .							
00	8x						VPMACSSWW Vo,Ho,Wo,Lo	VPMACSSWD Vo,Ho,Wo,Lo	VPMACSSDQL Vo,Ho,Wo,Lo
00	9x						VPMACSWW Vo,Ho,Wo,Lo	VPMACSWD Vo,Ho,Wo,Lo	VPMACSDQL Vo,Ho,Wo,Lo
00	Ax			VPCMOV Vx,Hx,Wx,Lx (W=0) Vx,Hx,Lx,Wx (W=1)	VPPERM Vo,Ho,Wo,Lo (W=0) Vo,Ho,Lo,Wo (W=1)			VPMADCSSWD Vo,Ho,Wo,Lo	
00	Bx							VPMADCSWD Vo,Ho,Wo,Lo	
00	Cx	VPROTB Vo,Wo,Ib	VPROTW Vo,Wo,Ib	VPROTD Vo,Wo,Ib	VPROTQ Vo,Wo,Ib				
. . .	Dx-Fx	. . .							

Table A-26. XOP Opcode Map 8h, Low Nibble = [8h:Fh]

XOP.pp	Opcode	x8	x9	xA	xB	xC	xD	xE	xF
. . .	0x-07x	. . .							
00	8x							VPMACSSDD Vo,Ho,Wo,Lo	VPMACSSDQH Vo,Ho,Wo,Lo
00	9x							VPMACSDD Vo,Ho,Wo,Lo	VPMACSDQH Vo,Ho,Wo,Lo
. . .	Ax-Bx	. . .							
00	Cx					VPCOMccB[1] Vo,Ho,Wo,Ib	VPCOMccW[1] Vo,Ho,Wo,Ib	VPCOMccD[1] Vo,Ho,Wo,Ib	VPCOMccQ[1] Vo,Ho,Wo,Ib
00	Dx								
00	Ex					VPCOMccUB[1] Vo,Ho,Wo,Ib	VPCOMccUW[1] Vo,Ho,Wo,Ib	VPCOMccUD[1] Vo,Ho,Wo,Ib	VPCOMccUQ[1] Vo,Ho,Wo,Ib
00	Fx								
	Note 1:	The condition codes are LT, LE, GT, GE, EQ, NEQ, FALSE, and TRUE. They are encoded via Ib, using 00...07h.							

Table A-27. XOP Opcode Map 9h, Low Nibble = [0h:7h]

XOP.pp	Opcode	x0	x1	x2	x3	x4	x5	x6	x7
00	0x		XOP group #1	XOP group #2					
00	1x			XOP group #3					
. . .	2x-7x	. . .							
00	8x	VFRCZPS Vx,Wx	VFRCZPD Vx,Wx	VFRCZSS Vq,Wss	VFRCZSD Vq,Wsd				
00	9x	VPROTB Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPROTW Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPROTD Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPROTQ Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHLB Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHLW Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHLD Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHLQ Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)
. . .	Ax-Bx	. . .							
00	Cx		VPHADDBW Vo,Wo	VPHADDBD Vo,Wo	VPHADDBQ Vo,Wo			VPHADDWD Vo,Wo	VPHADDWQ Vo,Wo
00	Dx		VPHADDUBWD Vo,Wo	VPHADDUBD Vo,Wo	VPHADDUBQ Vo,Wo			VPHADDUWD Vo,Wo	VPHADDUWQ Vo,Wo
00	Ex		VPHSUBBW Vo,Wo	VPHSUBWD Vo,Wo	VPHSUBDQ Vo,Wo				
. . .	Fx	. . .							

Table A-28. XOP Opcode Map 9h, Low Nibble = [8h:Fh]

XOP.pp	Opcode	x8	x9	xA	xB	xC	xD	xE	xF
. . .	0x-8x	. . .							
00	9x	VPSHAB Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHAW Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHAD Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)	VPSHAQ Vo,Wo,Ho (W=0) Vo,Ho,Wo (W=1)				
. . .	Ax-Bx	. . .							
00	Cx				VPHADDDQ Vo,Wo				
00	Dx				VPHADDUDQ Vo,Wo				
. . .	Ex-Fx	. . .							

Table A-29. XOP Opcode Map Ah, Low Nibble = [0h:7h]

XOP.pp	Opcode	x0	x1	x2	x3	x4	x5	x6	x7
. . .	0x	. . .							
00	1x	BEXTR Gy,Ey,Id		XOP group #4					
. . .	2x-Fx	. . .							

Table A-30. XOP Opcode Map Ah, Low Nibble = [8h:Fh]

XOP.pp	Opcode	x8	x9	xA	xB	xC	xD	xE	xF
n/a	0x-Fx								
Opcodes Reserved									

Table A-31. XOP Opcode Groups

		ModRM.reg							
Group		/0	/1	/2	/3	/4	/5	/6	/7
XOP 9 01	#1		BLCFILL By,Ey	BLSFILL By,Ey	BLCS By,Ey	TZMSK By,Ey	BLCIC By,Ey	BLSIC By,Ey	T1MSKC By,Ey
XOP 9 02	#2		BLCMSK By,Ey					BLCI By,Ey	
XOP 9 12	#3	LLWPCB Ry	SLWPCB Ry						
XOP A 12	#4	LWPINS By,Ed,Id	LWPVAL By,Ed,Id						

A.2 Operand Encodings

An operand is data that affects or is affected by the execution of an instruction. Operands may be located in registers, memory, or I/O ports. For some instructions, the location of one or more operands is implicitly specified based on the opcode alone. However, for most instructions, operands are specified using bytes that immediately follow the opcode byte. These bytes are designated the *mode-register-memory* (ModRM) byte, the *scale-index-base* (SIB) byte, the displacement byte(s), and the immediate byte(s). The presence of the SIB, displacement, and immediate bytes are optional depending on the instruction, and, for instructions that reference memory, the memory addressing mode.

The following sections describe the encoding of the ModRM and SIB bytes in various processor modes.

A.2.1 ModRM Operand References

Figure A-2 below shows the format of the ModRM byte. There are three fields—*mod, reg*, and *r/m*. The *reg* field is normally used to specify a register-based operand. The *mod* and *r/m* fields together provide a 5-bit field, augmented in 64-bit mode by the R and B bits of a REX, VEX, or XOP prefix, normally used to specify the location of a second memory- or register-based operand and, for a memory-based operand, the addressing mode.

As described in “Encoding Extensions Using the ModRM Byte” on page 515, certain instructions use either the reg field, the r/m field, or the entire ModRM byte to extend the opcode byte in the encoding of the instruction operation.

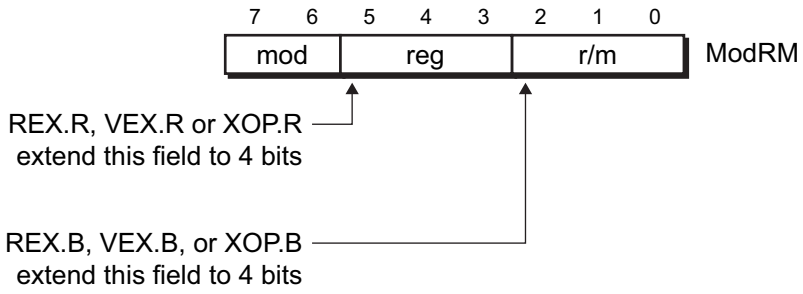


Figure A-2. ModRM-Byte Format

The two sections below describe the ModRM operand encodings, first for 16-bit references and then for 32-bit and 64-bit references.

16-Bit Register and Memory References. Table A-32 shows the notation and encoding conventions for register references using the ModRM *reg* field. This table is comparable to Table A-34 on page 548 but applies only when the address-size is 16-bit. Table A-33 on page 546 shows the

notation and encoding conventions for 16-bit memory references using the ModRM byte. This table is comparable to Table A-35 on page 549.

Table A-32. ModRM *reg* Field Encoding, 16-Bit Addressing

Mnemonic Notation	ModRM *reg* Field							
	/0	/1	/2	/3	/4	/5	/6	/7
reg8	AL	CL	DL	BL	AH	CH	DH	BH
reg16	AX	CX	DX	BX	SP	BP	SI	DI
reg32	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mmx	MMX0	MMX1	MMX2	MMX3	MMX4	MMX5	MMX6	MMX7
xmm	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
ymm	YMM0	YMM1	YMM2	YMM3	YMM4	YMM5	YMM6	YMM7
sReg	ES	CS	SS	DS	FS	GS	invalid	invalid
cReg	CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
dReg	DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7

Table A-33. ModRM Byte Encoding, 16-Bit Addressing

Effective Address	*ModRM mod* Field (binary)	ModRM *reg* Field[1]								*ModRM r/m* Field (binary)
		/0	/1	/2	/3	/4	/5	/6	/7	
		Complete ModRM Byte (hex)								
[BX] + [SI]	00	00	08	10	18	20	28	30	38	000
[BX] + [DI]		01	09	11	19	21	29	31	39	001
[BP] + [SI]		02	0A	12	1A	22	2A	32	3A	010
[BP] + [DI]		03	0B	13	1B	23	2B	33	3B	011
[SI]		04	0C	14	1C	24	2C	34	3C	100
[DI]		05	0D	15	1D	25	2D	35	3D	101
disp16		06	0E	16	1E	26	2E	36	3E	110
[BX]		07	0F	17	1F	27	2F	37	3F	111

Notes:
1. *See Table A-32 for complete specification of ModRM "reg" field.*

Table A-33. ModRM Byte Encoding, 16-Bit Addressing (continued)

Effective Address	*ModRM mod* Field (binary)	ModRM *reg* Field[1]								*ModRM r/m* Field (binary)
		/0	/1	/2	/3	/4	/5	/6	/7	
		Complete ModRM Byte (hex)								
[BX] + [SI] + *disp8*	01	40	48	50	58	60	68	70	78	000
[BX] + [DI] + *disp8*		41	49	51	59	61	69	71	79	001
[BP] + [SI] + *disp8*		42	4A	52	5A	62	6A	72	7A	010
[BP] + [DI] + *disp8*		43	4B	53	5B	63	6B	73	7B	011
[SI] + *disp8*		44	4C	54	5C	64	6C	74	7C	100
[DI] + *disp8*		45	4D	55	5D	65	6D	75	7D	101
[BP] + *disp8*		46	4E	56	5E	66	6E	76	7E	110
[BX] + *disp8*		47	4F	57	5F	67	6F	77	7F	111
[BX] + [SI] + *disp16*	10	80	88	90	98	A0	A8	B0	B8	000
[BX] + [DI] + *disp16*		81	89	91	99	A1	A9	B1	B9	001
[BP] + [SI] + *disp16*		82	8A	92	9A	A2	AA	B2	BA	010
[BP] + [DI] + *disp16*		83	8B	93	9B	A3	AB	B3	BB	011
[SI] + *disp16*		84	8C	94	9C	A4	AC	B4	BC	100
[DI] + *disp16*		85	8D	95	9D	A5	AD	B5	BD	101
[BP] + *disp16*		86	8E	96	9E	A6	AE	B6	BE	110
[BX] + *disp16*		87	8F	97	9F	A7	AF	B7	BF	111
AL/ AX/ EAX/ MMX0/ XMM0/ YMM0	11	C0	C8	D0	D8	E0	E8	F0	F8	000
CL/ CX/ ECX/ MMX1/ XMM1/ YMM1		C1	C9	D1	D9	E1	E9	F1	F9	001
DL/ DX/ EDX/ MMX2/ XMM2/ YMM2		C2	CA	D2	DA	E2	EA	F2	FA	010
BL/ BX/ EBX/ MMX3/ XMM3/ YMM3		C3	CB	D3	DB	E3	EB	F3	FB	011
AH/ SP/ ESP/ MMX4/ XMM4/ YMM4		C4	CC	D4	DC	E4	EC	F4	FC	100
CH/ BP/ EBP/ MMX5/ XMM5/ YMM5		C5	CD	D5	DD	E5	ED	F5	FD	101
DH/ SI/ ESI/ MMX6/ XMM6/ YMM6		C6	CE	D6	DE	E6	EE	F6	FE	110
BH/ DI/ EDI/ MMX7/ XMM7/ YMM7		C7	CF	D7	DF	E7	EF	F7	FF	111

Notes:

1. *See Table A-32 for complete specification of ModRM "reg" field.*

Register and Memory References for 32-Bit and 64-Bit Addressing. Table A-34 on page 548 shows the encoding for register references using the ModRM *reg* field. The first ten rows of Table A-34 show references when the REX.R bit is cleared to 0, and the last ten rows show references when the REX.R bit is set to 1. In this table, entries under the *Mnemonic Notation* heading correspond

to register notation described in "Mnemonic Syntax" on page 53, and the */r* notation under the *ModRM reg Field* heading corresponds to that described in "Opcode Syntax" on page 56.

Table A-34. ModRM *reg* Field Encoding, 32-Bit and 64-Bit Addressing

Mnemonic Notation	REX.R Bit	ModRM *reg* Field							
		/0	/1	/2	/3	/4	/5	/6	/7
reg8	0	AL	CL	DL	BL	AH/SPL	CH/BPL	DH/SIL	BH/DIL
reg16		AX	CX	DX	BX	SP	BP	SI	DI
reg32		EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
reg64		RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
mmx		MMX0	MMX1	MMX2	MMX3	MMX4	MMX5	MMX6	MMX7
xmm		XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
ymm		YMM0	YMM1	YMM2	YMM3	YMM4	YMM5	YMM6	YMM7
sReg		ES	CS	SS	DS	FS	GS	invalid	invalid
cReg		CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
dReg		DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7
reg8	1	R8B	R9B	R10B	R11B	R12B	R13B	R14B	R15B
reg16		R8W	R9W	R10W	R11W	R12W	R13W	R14W	R15W
reg32		R8D	R9D	R10D	R11D	R12D	R13D	R14D	R15D
reg64		R8	R9	R10	R11	R12	R13	R14	R15
mmx		MMX0	MMX1	MMX2	MMX3	MMX4	MMX5	MMX6	MMX7
xmm		XMM8	XMM9	XMM10	XMM11	XMM12	XMM13	XMM14	XMM15
ymm		YMM8	YMM9	YMM10	YMM11	YMM12	YMM13	YMM14	YMM15
sReg		ES	CS	SS	DS	FS	GS	invalid	invalid
cReg		CR8	CR9	CR10	CR11	CR12	CR13	CR14	CR15
dReg		DR8	DR9	DR10	DR11	DR12	DR13	DR14	DR15

Table A-35 on page 549 shows the encoding for 32-bit and 64-bit memory references using the ModRM byte. This table describes 32-bit and 64-bit addressing, with the REX.B bit set or cleared. The *Effective Address* is shown in the two left-most columns, followed by the binary encoding of the ModRM-byte *mod* field, followed by the eight possible hex values of the complete ModRM byte (one value for each binary encoding of the ModRM-byte *reg* field), followed by the binary encoding of the ModRM *r/m* field.

The /0 through /7 notation for the ModRM *reg* field (bits [5:3]) means that the three-bit field contains a value from zero (binary 000) to 7 (binary 111).

Table A-35. ModRM Byte Encoding, 32-Bit and 64-Bit Addressing

Effective Address		*ModRM mod* Field (binary)	ModRM *reg* Field[1]								*ModRM r/m* Field (binary)
			/0	/1	/2	/3	/4	/5	/6	/7	
REX.B = 0	REX.B = 1		Complete ModRM Byte (hex)								
[rAX]	[r8]	00	00	08	10	18	20	28	30	38	000
[rCX]	[r9]		01	09	11	19	21	29	31	39	001
[rDX]	[r10]		02	0A	12	1A	22	2A	32	3A	010
[rBX]	[r11]		03	0B	13	1B	23	2B	33	3B	011
SIB[2]	SIB[2]		04	0C	14	1C	24	2C	34	3C	100
[rIP] + *disp32* or *disp32*[3]	[rIP] + *disp32* or *disp32*[3]		05	0D	15	1D	25	2D	35	3D	101
[rSI]	[r14]		06	0E	16	1E	26	2E	36	3E	110
[rDI]	[r15]		07	0F	17	1F	27	2F	37	3F	111
[rAX] + *disp8*	[r8] + *disp8*	01	40	48	50	58	60	68	70	78	000
[rCX] + *disp8*	[r9] + *disp8*		41	49	51	59	61	69	71	79	001
[rDX] + *disp8*	[r10] + *disp8*		42	4A	52	5A	62	6A	72	7A	010
[rBX] + *disp8*	[r11] + *disp8*		43	4B	53	5B	63	6B	73	7B	011
[SIB] + *disp8*	[SIB] + *disp8*		44	4C	54	5C	64	6C	74	7C	100
[rBP] + *disp8*	[r13] + *disp8*		45	4D	55	5D	65	6D	75	7D	101
[rSI] + *disp8*	[r14] + *disp8*		46	4E	56	5E	66	6E	76	7E	110
[rDI] + *disp8*	[r15] + *disp8*		47	4F	57	5F	67	6F	77	7F	111
[rAX] + *disp32*	[r8] + *disp32*	10	80	88	90	98	A0	A8	B0	B8	000
[rCX] + *disp32*	[r9] + *disp32*		81	89	91	99	A1	A9	B1	B9	001
[rDX] + *disp32*	[r10] + *disp32*		82	8A	92	9A	A2	AA	B2	BA	010
[rBX] + *disp32*	[r11] + *disp32*		83	8B	93	9B	A3	AB	B3	BB	011
SIB + *disp32*	SIB + *disp32*		84	8C	94	9C	A4	AC	B4	BC	100
[rBP] + *disp32*	[r13] + *disp32*		85	8D	95	9D	A5	AD	B5	BD	101
[rSI] + *disp32*	[r14] + *disp32*		86	8E	96	9E	A6	AE	B6	BE	110
[rDI] + *disp32*	[r15] + *disp32*		87	8F	97	9F	A7	AF	B7	BF	111

Notes:

1. *See Table A-34 for complete specification of ModRM "reg" field.*
2. *If SIB.base = 5, the SIB byte is followed by four-byte disp32 field and addressing mode is absolute.*
3. *In 64-bit mode, the effective address is [rIP]+disp32. In all other modes, the effective address is disp32. If the address-size prefix is used in 64-bit mode to override 64-bit addressing, the [RIP]+disp32 effective address is truncated after computation to 32 bits.*

Table A-35. ModRM Byte Encoding, 32-Bit and 64-Bit Addressing (continued)

Effective Address		*ModRM mod* Field (binary)	ModRM *reg* Field[1]								*ModRM r/m* Field (binary)
			/0	/1	/2	/3	/4	/5	/6	/7	
REX.B = 0	REX.B = 1		Complete ModRM Byte (hex)								
AL/rAX/MMX0/XMM0/ YMM0	r8/MMX0/XMM8/ YMM8	11	C0	C8	D0	D8	E0	E8	F0	F8	000
CL/rCX/MMX1/XMM1/ YMM1	r9/MMX1/XMM9/ YMM9		C1	C9	D1	D9	E1	E9	F1	F9	001
DL/rDX/MMX2/XMM2/ YMM2	r10/MMX2/XMM10/ YMM10		C2	CA	D2	DA	E2	EA	F2	FA	010
BL/rBX/MMX3/XMM3/ YMM3	r11/MMX3/XMM11/ YMM11		C3	CB	D3	DB	E3	EB	F3	FB	011
AH/SPL/rSP/MMX4/ XMM4/YMM4	r12/MMX4/XMM12/ YMM12		C4	CC	D4	DC	E4	EC	F4	FC	100
CH/BPL/rBP/MMX5/ XMM5/YMM5	r13/MMX5/XMM13/ YMM13		C5	CD	D5	DD	E5	ED	F5	FD	101
DH/SIL/rSI/MMX6/ XMM6/YMM6	r14/MMX6/XMM14/ YMM14		C6	CE	D6	DE	E6	EE	F6	FE	110
BH/DIL/rDI/MMX7/ XMM7/YMM7	r15/MMX7/XMM15/ YMM15		C7	CF	D7	DF	E7	EF	F7	FF	111

Notes:

1. *See Table A-34 for complete specification of ModRM "reg" field.*
2. *If SIB.base = 5, the SIB byte is followed by four-byte disp32 field and addressing mode is absolute.*
3. *In 64-bit mode, the effective address is [rIP]+disp32. In all other modes, the effective address is disp32. If the address-size prefix is used in 64-bit mode to override 64-bit addressing, the [RIP]+disp32 effective address is truncated after computation to 32 bits.*

A.2.2 SIB Operand References

Figure A-3 on page 551 shows the format of a scale-index-base (SIB) byte. Some instructions have an SIB byte following their ModRM byte to define memory addressing for the complex-addressing modes described in "Effective Addresses" in Volume 1. The SIB byte has three fields—*scale*, *index*, and *base*—that define the scale factor, index-register number, and base-register number for 32-bit and 64-bit complex addressing modes. In 64-bit mode, the REX.B and REX.X bits extend the encoding of the SIB byte's *base* and *index* fields.

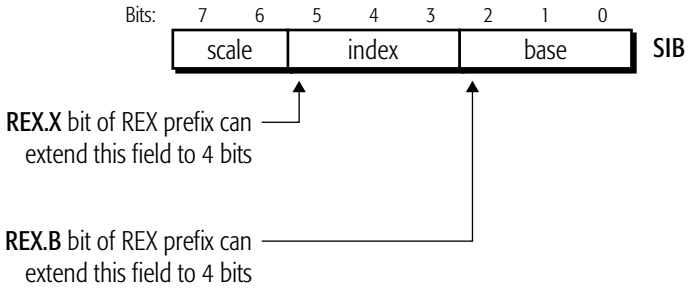


Figure A-3. SIB Byte Format

Table A-36 shows the encodings for the SIB byte's *base* field, which specifies the base register for addressing. Table A-37 on page 552 shows the encodings for the effective address referenced by a complete SIB byte, including its *scale* and *index* fields. The /0 through /7 notation for the SIB *base* field means that the three-bit field contains a value between zero (binary 000) and 7 (binary 111).

Table A-36. Addressing Modes: SIB *base* Field Encoding

REX.B Bit	ModRM *mod* Field	SIB *base* Field							
		/0	*/1*	*/2*	*/3*	*/4*	*/5*	*/6*	*/7*
0	00	[rAX]	[rCX]	[rDX]	[rBX]	[rSP]	*disp32*	[rSI]	[rDI]
	01						[rBP] + *disp8*		
	10						[rBP] + *disp32*		
1	00	[r8]	[r9]	[r10]	[r11]	[r12]	disp32	[r14]	[r15]
	01						[r13] + *disp8*		
	10						[r13] + *disp32*		

Table A-37. Addressing Modes: SIB Byte Encoding

Effective Address		SIB *scale* Field	SIB *index* Field	SIB *base* Field[1]							
				REX.B = 0	rAX	rCX	rDX	rBX	rSP	note 1	rSI	rDI
				REX.B = 1	r8	r9	r10	r11	r12	note 1	r14	r15
					/0	/1	/2	/3	/4	/5	/6	/7
REX.X = 0	REX.X = 1				Complete SIB Byte (hex)							
[rAX] + [*base*]	[r8] + [*base*]	00	000		00	01	02	03	04	05	06	07
[rCX] + [*base*]	[r9] + [*base*]		001		08	09	0A	0B	0C	0D	0E	0F
[rDX] + [*base*]	[r10] + [*base*]		010		10	11	12	13	14	15	16	17
[rBX] + [*base*]	[r11] + [*base*]		011		18	19	1A	1B	1C	1D	1E	1F
[*base*]	[r12] + [*base*]		100		20	21	22	23	24	25	26	27
[rBP] + [*base*]	[r13] + [*base*]		101		28	29	2A	2B	2C	2D	2E	2F
[rSI] + [*base*]	[r14] + [*base*]		110		30	31	32	33	34	35	36	37
[rDI] + [*base*]	[r15] + [*base*]		111		38	39	3A	3B	3C	3D	3E	3F
[rAX] * 2 + [*base*]	[r8] * 2 + [*base*]	01	000		40	41	42	43	44	45	46	47
[rCX] * 2 + [*base*]	[r9] * 2 + [*base*]		001		48	49	4A	4B	4C	4D	4E	4F
[rDX] * 2 + [base]	[r10] * 2 + [*base*]		010		50	51	52	53	54	55	56	57
[rBX] * 2 + [*base*]	[r11] * 2 + [*base*]		011		58	59	5A	5B	5C	5D	5E	5F
[*base*]	[r12] * 2 + [*base*]		100		60	61	62	63	64	65	66	67
[rBP] * 2 + [*base*]	[r13] * 2 + [*base*]		101		68	69	6A	6B	6C	6D	6E	6F
[rSI] * 2 + [*base*]	[r14] * 2 + [*base*]		110		70	71	72	73	74	75	76	77
[rDI] * 2 + [*base*]	[r15] * 2 + [*base*]		111		78	79	7A	7B	7C	7D	7E	7F
[rAX] * 4 + [*base*]	[r8] * 4 + [*base*]	10	000		80	81	82	83	84	85	86	87
[rCX] * 4 + [*base*]	[r9] * 4 + [*base*]		001		88	89	8A	8B	8C	8D	8E	8F
[rDX] * 4 + [*base*]	[r10] * 4 + [*base*]		010		90	91	92	93	94	95	96	97
[rBX] * 4 + [*base*]	[r11] * 4 + [*base*]		011		98	99	9A	9B	9C	9D	9E	9F
[*base*]	[r12] * 4 + [*base*]		100		A0	A1	A2	A3	A4	A5	A6	A7
[rBP]*4+[*base*]	[r13] * 4 + [*base*]		101		A8	A9	AA	AB	AC	AD	AE	AF
[rSI]*4+[*base*]	[r14] * 4 + [*base*]		110		B0	B1	B2	B3	B4	B5	B6	B7
[rDI]*4+[*base*]	[r15] * 4 + [*base*]		111		B8	B9	BA	BB	BC	BD	BE	BF

Notes:

1. *See Table A-36 on page 551 for complete specification of SIB base field.*

Table A-37. Addressing Modes: SIB Byte Encoding (continued)

Effective Address		SIB *scale* Field	SIB *index* Field	SIB *base* Field[1]							
				REX.B = 0	rAX	rCX	rDX	rBX	rSP	note 1	rSI	rDI
				REX.B = 1	r8	r9	r10	r11	r12	note 1	r14	r15
					/0	/1	/2	/3	/4	/5	/6	/7
REX.X = 0	REX.X = 1				Complete SIB Byte (hex)							
[rAX] * 8 + [*base*]	[r8] * 8 + [*base*]	11	000		C0	C1	C2	C3	C4	C5	C6	C7
[rCX] * 8 + [*base*]	[r9] * 8 + [*base*]		001		C8	C9	CA	CB	CC	CD	CE	CF
[rDX] * 8 + [*base*]	[r10] * 8 + [*base*]		010		D0	D1	D2	D3	D4	D5	D6	D7
[rBX] * 8 + [*base*]	[r11] * 8 + [*base*]		011		D8	D9	DA	DB	DC	DD	DE	DF
[*base*]	[r12] * 8 + [*base*]		100		E0	E1	E2	E3	E4	E5	E6	E7
[rBP] * 8 + [*base*]	[r13] * 8 + [*base*]		101		E8	E9	EA	EB	EC	ED	EE	EF
[rSI] * 8 + [*base*]	[r14] * 8 + [*base*]		110		F0	F1	F2	F3	F4	F5	F6	F7
[rDI] * 8 + [*base*]	[r15] * 8 + [*base*]		111		F8	F9	FA	FB	FC	FD	FE	FF

Notes:

1. *See Table A-36 on page 551 for complete specification of SIB base field.*

Appendix B General-Purpose Instructions in 64-Bit Mode

This appendix provides details of the general-purpose instructions in 64-bit mode and its differences from legacy and compatibility modes. The appendix covers only the general-purpose instructions (those described in *Chapter 3, "General-Purpose Instruction Reference"*). It does not cover the 128-bit media, 64-bit media, or x87 floating-point instructions because those instructions are not affected by 64-bit mode, other than in the access by such instructions to extended GPR and XMM registers when using a REX prefix.

B.1 General Rules for 64-Bit Mode

In 64-bit mode, the following general rules apply to instructions and their operands:

- **"Promoted to 64 Bit"**: If an instruction's operand size (16-bit or 32-bit) in legacy and compatibility modes depends on the CS.D bit and the operand-size override prefix, then the operand-size choices in 64-bit mode are extended from 16-bit and 32-bit to include 64 bits (with a REX prefix), or the operand size is fixed at 64 bits. Such instructions are said to be "*Promoted to 64 bits"* in Table B-1. However, byte-operand opcodes of such instructions are not promoted.
- **Byte-Operand Opcodes Not Promoted**: As stated above in "Promoted to 64 Bit", byte-operand opcodes of promoted instructions are not promoted. Those opcodes continue to operate only on bytes.
- **Fixed Operand Size**: If an instruction's operand size is fixed in legacy mode (thus, independent of CS.D and prefix overrides), that operand size is usually fixed at the same size in 64-bit mode. For example, CPUID operates on 32-bit operands, irrespective of attempts to override the operand size.
- **Default Operand Size**: The default operand size for most instructions is 32 bits, and a REX prefix must be used to change the operand size to 64 bits. However, two groups of instructions default to 64-bit operand size and do not need a REX prefix: (1) near branches and (2) all instructions, except far branches, that implicitly reference the RSP. See Table B-5 on page 583 for a list of all instructions that default to 64-bit operand size.
- **Zero-Extension of 32-Bit Results**: Operations on 32-bit operands in 64-bit mode zero-extend the high 32 bits of 64-bit GPR destination registers.
- **No Extension of 8-Bit and 16-Bit Results**: Operations on 8-bit and 16-bit operands in 64-bit mode leave the high 56 or 48 bits, respectively, of 64-bit GPR destination registers unchanged.
- **Shift and Rotate Counts**: When the operand size is 64 bits, shifts and rotates use one additional bit (6 bits total) to specify shift-count or rotate-count, allowing 64-bit shifts and rotates.
- **Immediates**: The maximum size of immediate operands is 32 bits, except that 64-bit immediates can be MOVed into 64-bit GPRs. Immediates that are less than 64 bits are a maximum of 32 bits, and are sign-extended to 64 bits during use.

- **Displacements and Offsets**: The maximum size of an address displacement or offset is 32 bits, except that 64-bit offsets can be used by specific MOV opcodes that read or write AL or rAX. Displacements and offsets that are less than 64 bits are a maximum of 32 bits, and are sign-extended to 64 bits during use.
- **Undefined High 32 Bits After Mode Change**: The processor does not preserve the upper 32 bits of the 64-bit GPRs across switches from 64-bit mode to compatibility or legacy modes. In compatibility or legacy mode, the upper 32 bits of the GPRs are undefined and not accessible to software.

B.2 Operation and Operand Size in 64-Bit Mode

Table B-1 lists the integer instructions, showing operand size in 64-bit mode and the state of the high 32 bits of destination registers when 32-bit operands are used. Opcodes, such as byte-operand versions of several instructions, that do not appear in Table B-1 are covered by the general rules described in "General Rules for 64-Bit Mode" on page 555.

Table B-1. Operations and Operands in 64-Bit Mode

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
AAA - ASCII Adjust after Addition 37	INVALID IN 64-BIT MODE (invalid-opcode exception)			
AAD - ASCII Adjust AX before Division D5	INVALID IN 64-BIT MODE (invalid-opcode exception)			
AAM - ASCII Adjust AX after Multiply D4	INVALID IN 64-BIT MODE (invalid-opcode exception)			
AAS - ASCII Adjust AL after Subtraction 3F	INVALID IN 64-BIT MODE (invalid-opcode exception)			

Notes:

1. *See "General Rules for 64-Bit Mode" on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See "General Rules for 64-Bit Mode" on page 555 for definitions of "Promoted to 64 bits" and related topics.*
3. *If "Type of Operation" is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
ADC—Add with Carry 11 13 15 81 /2 83 /2	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
ADD—Signed or Unsigned Add 01 03 05 81 /0 83 /0	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
AND—Logical AND 21 23 25 81 /4 83 /4	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
ARPL - Adjust Requestor Privilege Level 63	OPCODE USED as MOVSXD in 64-BIT MODE			
BOUND - Check Array Against Bounds 62	INVALID IN 64-BIT MODE (invalid-opcode exception)			
BSF—Bit Scan Forward 0F BC	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
BSR—Bit Scan Reverse 0F BD	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
BSWAP—Byte Swap 0F C8 through 0F CF	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Swap all 8 bytes of a 64-bit GPR.
BT—Bit Test 0F A3 0F BA /4	Promoted to 64 bits.	32 bits	No GPR register results.	
BTC—Bit Test and Complement 0F BB 0F BA /7	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
BTR—Bit Test and Reset 0F B3 0F BA /6	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
BTS—Bit Test and Set 0F AB 0F BA /5	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
CALL—Procedure Call Near	See “Near Branches in 64-Bit Mode” in Volume 1.			
E8	Promoted to 64 bits.	64 bits	Can’t encode.[6]	RIP = RIP + 32-bit displacement sign-extended to 64 bits.
FF /2	Promoted to 64 bits.	64 bits	Can’t encode.[6]	RIP = 64-bit offset from register or memory.

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
CALL—Procedure Call Far	See “Branches to 64-Bit Offsets” in Volume 1.			
9A	INVALID IN 64-BIT MODE (invalid-opcode exception)			
FF /3	Promoted to 64 bits.	32 bits	If selector points to a gate, then RIP = 64-bit offset from gate, else RIP = zero-extended 32-bit offset from far pointer referenced in instruction.	
CBW, CWDE, CDQE—Convert Byte to Word, Convert Word to Doubleword, Convert Doubleword to Quadword 98	Promoted to 64 bits.	32 bits (size of destination register)	CWDE: Converts word to doubleword. Zero-extends EAX to RAX.	CDQE (new mnemonic): Converts doubleword to quadword. RAX = sign-extended EAX.
CDQ	see **CWD, CDQ, CQO**			
CDQE (new mnemonic)	see **CBW, CWDE, CDQE**			
CDWE	see **CBW, CWDE, CDQE**			
CLC—Clear Carry Flag F8	Same as legacy mode.	Not relevant.	No GPR register results.	
CLD—Clear Direction Flag FC	Same as legacy mode.	Not relevant.	No GPR register results.	
CLFLUSH—Cache Line Invalidate 0F AE /7	Same as legacy mode.	Not relevant.	No GPR register results.	
CLGI—Clear Global Interrupt 0F 01 DD	Same as legacy mode	Not relevant	No GPR register results.	
CLI—Clear Interrupt Flag FA	Same as legacy mode.	Not relevant.	No GPR register results.	

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
CLTS—Clear Task-Switched Flag in CR0 0F 06	Same as legacy mode.	Not relevant.	No GPR register results.	
CMC—Complement Carry Flag F5	Same as legacy mode.	Not relevant.	No GPR register results.	
CMOV*cc*—Conditional Move 0F 40 through 0F 4F	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits. This occurs even if the condition is false.	
CMP—Compare 39 3B 3D 81 /7 83 /7	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
CMPS, **CMPSW**, **CMPSD, CMPSQ**—Compare Strings A7	Promoted to 64 bits.	32 bits	CMPSD: Compare String Doublewords. See footnote[5]	CMPSQ (new mnemonic): Compare String Quadwords See footnote[5]
CMPXCHG—Compare and Exchange 0F B1	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	

Notes:

1. *See "General Rules for 64-Bit Mode" on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See "General Rules for 64-Bit Mode" on page 555 for definitions of "Promoted to 64 bits" and related topics.*
3. *If "Type of Operation" is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
CMPXCHG8B—Compare and Exchange Eight Bytes 0F C7 /1	Same as legacy mode.	32 bits.	Zero-extends EDX and EAX to 64 bits.	CMPXCHG16B (new mnemonic): Compare and Exchange 16 Bytes.
CPUID—Processor Identification 0F A2	Same as legacy mode.	Operand size fixed at 32 bits.	Zero-extends 32-bit register results to 64 bits.	
CQO (new mnemonic)	see **CWD, CDQ, CQO**			
CWD, CDQ, CQO—Convert Word to Doubleword, Convert Doubleword to Quadword, Convert Quadword to Double Quadword 99	Promoted to 64 bits.	32 bits (size of destination register)	CDQ: Converts doubleword to quadword. Sign-extends EAX to EDX. Zero-extends EDX to RDX. RAX is unchanged.	CQO (new mnemonic): Converts quadword to double quadword. Sign-extends RAX to RDX. RAX is unchanged.
DAA - Decimal Adjust AL after Addition 27	INVALID IN 64-BIT MODE (invalid-opcode exception)			
DAS - Decimal Adjust AL after Subtraction 2F	INVALID IN 64-BIT MODE (invalid-opcode exception)			

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
DEC—Decrement by 1 FF /1	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
48 through 4F	OPCODE USED as REX PREFIX in 64-BIT MODE			
DIV—Unsigned Divide F7 /6	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	RDX:RAX contain a 64-bit quotient (RAX) and 64-bit remainder (RDX).
ENTER—Create Procedure Stack Frame C8	Promoted to 64 bits.	64 bits	Can't encode[6]	
HLT—Halt F4	Same as legacy mode.	Not relevant.	No GPR register results.	
IDIV—Signed Divide F7 /7	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	RDX:RAX contain a 64-bit quotient (RAX) and 64-bit remainder (RDX).

Notes:

1. *See "General Rules for 64-Bit Mode" on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See "General Rules for 64-Bit Mode" on page 555 for definitions of "Promoted to 64 bits" and related topics.*
3. *If "Type of Operation" is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
IMUL - Signed Multiply F7 /5	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	RDX:RAX = RAX * reg/mem64 (i.e., 128-bit result)
0F AF				reg64 = reg64 * reg/mem64
69				reg64 = reg/mem64 * imm32
6B				reg64 = reg/mem64 * imm8
IN—Input From Port E5 ED	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
INC—Increment by 1 FF /0	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
40 through 47	OPCODE USED as REX PREFIX in 64-BIT MODE			
INS, INSW, INSD—Input String 6D	Same as legacy mode.	32 bits	INSD: Input String Doublewords. No GPR register results. See footnote[5]	
INT n—Interrupt to Vector CD	Promoted to 64 bits.	Not relevant.	See “Long-Mode Interrupt Control Transfers” in Volume 2.	
INT3—Interrupt to Debug Vector CC				
INTO - Interrupt to Overflow Vector CE	INVALID IN 64-BIT MODE (invalid-opcode exception)			

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
INVD—Invalidate Internal Caches 0F 08	Same as legacy mode.	Not relevant.	No GPR register results.	
INVLPG—Invalidate TLB Entry 0F 01 /7	Promoted to 64 bits.	Not relevant.	No GPR register results.	
INVLPGA—Invalidate TLB Entry in a Specified ASID	Same as legacy mode.	Not relevant.	No GPR register results.	
IRET, **IRETD, IRETQ**—Interrupt Return CF	Promoted to 64 bits.	32 bits	IRETD: Interrupt Return Doubleword. See “Long-Mode Interrupt Control Transfers” in Volume 2.	IRETQ (new mnemonic): Interrupt Return Quadword. See “Long-Mode Interrupt Control Transfers” in Volume 2.
Jcc—Jump Conditional	See “Near Branches in 64-Bit Mode” in Volume 1.			
70 through 7F	Promoted to 64 bits.	64 bits	Can’t encode.[6]	RIP = RIP + 8-bit displacement sign-extended to 64 bits.
0F 80 through 0F 8F				RIP = RIP + 32-bit displacement sign-extended to 64 bits.
JCXZ, **JECXZ, JRCXZ**—Jump on CX/ECX/RCX Zero E3	Promoted to 64 bits.	64 bits	Can’t encode.[6]	RIP = RIP + 8-bit displacement sign-extended to 64 bits. See footnote[5]

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
JMP—Jump Near	See “Near Branches in 64-Bit Mode” in Volume 1.			
EB	Promoted to 64 bits.	64 bits	Can’t encode.[6]	RIP = RIP + 8-bit displacement sign-extended to 64 bits.
E9				RIP = RIP + 32-bit displacement sign-extended to 64 bits.
FF /4				RIP = 64-bit offset from register or memory.
JMP—Jump Far	See “Branches to 64-Bit Offsets” in Volume 1.			
EA	INVALID IN 64-BIT MODE (invalid-opcode exception)			
FF /5	Promoted to 64 bits.	32 bits	If selector points to a gate, then RIP = 64-bit offset from gate, else RIP = zero-extended 32-bit offset from far pointer referenced in instruction.	
LAHF - Load Status Flags into AH Register 9F	Same as legacy mode.	Not relevant.		
LAR—Load Access Rights Byte 0F 02	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
LDS - Load DS Far Pointer C5	INVALID IN 64-BIT MODE (invalid-opcode exception)			

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
LEA—Load Effective Address 8D	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
LEAVE—Delete Procedure Stack Frame C9	Promoted to 64 bits.	64 bits	Can't encode[6]	
LES - Load ES Far Pointer C4	INVALID IN 64-BIT MODE (invalid-opcode exception)			
LFENCE—Load Fence 0F AE /5	Same as legacy mode.	Not relevant.	No GPR register results.	
LFS—Load FS Far Pointer 0F B4	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
LGDT—Load Global Descriptor Table Register 0F 01 /2	Promoted to 64 bits.	Operand size fixed at 64 bits.	No GPR register results. Loads 8-byte base and 2-byte limit.	
LGS—Load GS Far Pointer 0F B5	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
LIDT—Load Interrupt Descriptor Table Register 0F 01 /3	Promoted to 64 bits.	Operand size fixed at 64 bits.	No GPR register results. Loads 8-byte base and 2-byte limit.	
LLDT—Load Local Descriptor Table Register 0F 00 /2	Promoted to 64 bits.	Operand size fixed at 16 bits.	No GPR register results. References 16-byte descriptor to load 64-bit base.	
LMSW—Load Machine Status Word 0F 01 /6	Same as legacy mode.	Operand size fixed at 16 bits.	No GPR register results.	

Notes:

1. *See "General Rules for 64-Bit Mode" on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See "General Rules for 64-Bit Mode" on page 555 for definitions of "Promoted to 64 bits" and related topics.*
3. *If "Type of Operation" is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
LODS, **LODSW**, **LODSD, LODSQ**—Load String AD	Promoted to 64 bits.	32 bits	LODSD: Load String Doublewords. Zero-extends 32-bit register results to 64 bits. See footnote[5]	LODSQ (new mnemonic): Load String Quadwords. See footnote[5]
LOOP—Loop E2	Promoted to 64 bits.	64 bits	Can't encode.[6]	RIP = RIP + 8-bit displacement sign-extended to 64 bits. See footnote[5]
LOOPZ, **LOOPE**—Loop if Zero/Equal E1				
LOOPNZ, **LOOPNE**—Loop if Not Zero/Equal E0				
LSL—Load Segment Limit 0F 03	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
LSS —Load SS Segment Register 0F B2	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
LTR—Load Task Register 0F 00 /3	Promoted to 64 bits.	Operand size fixed at 16 bits.	No GPR register results. References 16-byte descriptor to load 64-bit base.	
LZCNT—Count Leading Zeros F3 0F BD	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
MFENCE—Memory Fence 0F AE /6	Same as legacy mode.	Not relevant.	No GPR register results.	
MONITOR—Setup Monitor Address 0F 01 C8	Same as legacy mode.	Operand size fixed at 32 bits.	No GPR register results.	

Notes:

1. *See "General Rules for 64-Bit Mode" on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See "General Rules for 64-Bit Mode" on page 555 for definitions of "Promoted to 64 bits" and related topics.*
3. *If "Type of Operation" is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
MOV—Move 89 8B C7 B8 through BF A1 (moffset) A3 (moffset)	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
				32-bit immediate is sign-extended to 64 bits.
				64-bit immediate.
			Zero-extends 32-bit register results to 64 bits. Memory offsets are address-sized and default to 64 bits.	Memory offsets are address-sized and default to 64 bits.
MOV—Move to/from Segment Registers 8C	Same as legacy mode.	32 bits	Zero-extends 32-bit register results to 64 bits.	
8E		Operand size fixed at 16 bits.	No GPR register results.	
MOV(CR*n*)—Move to/from Control Registers 0F 22 0F 20	Promoted to 64 bits.	Operand size fixed at 64 bits.	The high 32 bits of control registers differ in their writability and reserved status. See “System Resources” in Volume 2 for details.	
MOV(DR*n*)—Move to/from Debug Registers 0F 21 0F 23	Promoted to 64 bits.	Operand size fixed at 64 bits.	The high 32 bits of debug registers differ in their writability and reserved status. See “Debug and Performance Resources” in Volume 2 for details.	

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
MOVD—Move Doubleword or Quadword 0F 6E 0F 7E	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
66 0F 6E 66 0F 7E			Zero-extends 32-bit register results to 128 bits.	Zero-extends 64-bit register results to 128 bits.
MOVNTI—Move Non-Temporal Doubleword 0F C3	Promoted to 64 bits.	32 bits	No GPR register results.	
MOVS, **MOVSW**, **MOVSD, MOVSQ**—Move String A5	Promoted to 64 bits.	32 bits	MOVSD: Move String Doublewords. See footnote[5]	MOVSQ (new mnemonic): Move String Quadwords. See footnote[5]
MOVSX—Move with Sign-Extend 0F BE	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Sign-extends byte to quadword.
0F BF				Sign-extends word to quadword.

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
MOVSXD—Move with Sign-Extend Doubleword 63	New instruction, available only in 64-bit mode. (In other modes, this opcode is ARPL instruction.)	32 bits	Zero-extends 32-bit register results to 64 bits.	Sign-extends doubleword to quadword.
MOVZX—Move with Zero-Extend 0F B6	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Zero-extends byte to quadword.
0F B7				Zero-extends word to quadword.
MUL—Multiply Unsigned F7 /4	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	RDX:RAX=RAX * quadword in register or memory.
MWAIT—Monitor Wait 0F 01 C9	Same as legacy mode.	Operand size fixed at 32 bits.	No GPR register results.	
NEG—Negate Two's Complement F7 /3	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
NOP—No Operation 90	Same as legacy mode.	Not relevant.	No GPR register results.	
NOT—Negate One's Complement F7 /2	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	

Notes:

1. *See "General Rules for 64-Bit Mode" on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See "General Rules for 64-Bit Mode" on page 555 for definitions of "Promoted to 64 bits" and related topics.*
3. *If "Type of Operation" is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
OR—Logical OR 09 0B 0D 81 /1 83 /1	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
OUT—Output to Port E7 EF	Same as legacy mode.	32 bits	No GPR register results.	
OUTS, OUTSW, OUTSD—Output String 6F	Same as legacy mode.	32 bits	Writes doubleword to I/O port. No GPR register results. See footnote[5]	
PAUSE—Pause F3 90	Same as legacy mode.	Not relevant.	No GPR register results.	
POP—Pop Stack 8F /0 58 through 5F	Promoted to 64 bits.	64 bits	Cannot encode[6]	No GPR register results.
POP—Pop (segment register from) Stack 0F A1 (POP FS) 0F A9 (POP GS)	Same as legacy mode.	64 bits	Cannot encode[6]	No GPR register results.
1F (POP DS) 07 (POP ES) 17 (POP SS)	INVALID IN 64-BIT MODE (invalid-opcode exception)			

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
POPA, POPAD—Pop All to GPR Words or Doublewords 61	INVALID IN 64-BIT MODE (invalid-opcode exception)			
POPCNT—Bit Population Count F3 0F B8	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
POPF, **POPFD, POPFQ**—Pop to rFLAGS Word, Doublword, or Quadword 9D	Promoted to 64 bits.	64 bits	Cannot encode[6]	POPFQ (new mnemonic): Pops 64 bits off stack, writes low 32 bits into EFLAGS and zero-extends the high 32 bits of RFLAGS.
PREFETCH—Prefetch L1 Data-Cache Line 0F 0D /0	Same as legacy mode.	Not relevant.	No GPR register results.	
PREFETCH*level*—Prefetch Data to Cache Level *level* 0F 18 /0-3	Same as legacy mode.	Not relevant.	No GPR register results.	
PREFETCHW—Prefetch L1 Data-Cache Line for Write 0F 0D /1	Same as legacy mode.	Not relevant.	No GPR register results.	
PUSH—Push onto Stack FF /6 50 through 57 6A 68	Promoted to 64 bits.	64 bits	Cannot encode[6]	

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
PUSH—Push (segment register) onto Stack 0F A0 (PUSH FS) 0F A8 (PUSH GS)	Promoted to 64 bits.	64 bits	Cannot encode[6]	
0E (PUSH CS) 1E (PUSH DS) 06 (PUSH ES) 16 (PUSH SS)	INVALID IN 64-BIT MODE (invalid-opcode exception)			
PUSHA, PUSHAD - Push All to GPR Words or Doublewords 60	INVALID IN 64-BIT MODE (invalid-opcode exception)			
PUSHF, **PUSHFD, PUSHFQ**—Push rFLAGS Word, Doubleword, or Quadword onto Stack 9C	Promoted to 64 bits.	64 bits	Cannot encode[6]	PUSHFQ (new mnemonic): Pushes the 64-bit RFLAGS register.
RCL—Rotate Through Carry Left D1 /2 D3 /2 C1 /2	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
RCR—Rotate Through Carry Right D1 /3 D3 /3 C1 /3	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
RDMSR—Read Model-Specific Register 0F 32	Same as legacy mode.	Not relevant.	RDX[31:0] contains MSR[63:32], RAX[31:0] contains MSR[31:0]. Zero-extends 32-bit register results to 64 bits.	

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
RDPMC—Read Performance-Monitoring Counters 0F 33	Same as legacy mode.	Not relevant.	RDX[31:0] contains PMC[63:32], RAX[31:0] contains PMC[31:0]. Zero-extends 32-bit register results to 64 bits.	
RDTSC—Read Time-Stamp Counter 0F 31	Same as legacy mode.	Not relevant.	RDX[31:0] contains TSC[63:32], RAX[31:0] contains TSC[31:0]. Zero-extends 32-bit register results to 64 bits.	
RDTSCP—Read Time-Stamp Counter and Processor ID 0F 01 F9	Same as legacy mode.	Not relevant.	RDX[31:0] contains TSC[63:32], RAX[31:0] contains TSC[31:0]. RCX[31:0] contains the TSC_AUX MSR C000_0103h[31:0]. Zero-extends 32-bit register results to 64 bits.	
REP INS—Repeat Input String F3 6D	Same as legacy mode.	32 bits	Reads doubleword I/O port. See footnote[5]	
REP LODS—Repeat Load String F3 AD	Promoted to 64 bits.	32 bits	Zero-extends EAX to 64 bits. See footnote[5]	See footnote[5]
REP MOVS—Repeat Move String F3 A5	Promoted to 64 bits.	32 bits	No GPR register results. See footnote[5]	
REP OUTS—Repeat Output String to Port F3 6F	Same as legacy mode.	32 bits	Writes doubleword to I/O port. No GPR register results. See footnote[5]	
REP STOS—Repeat Store String F3 AB	Promoted to 64 bits.	32 bits	No GPR register results. See footnote[5]	
REPx CMPS —Repeat Compare String F3 A7	Promoted to 64 bits.	32 bits	No GPR register results. See footnote[5]	

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
REPx SCAS —Repeat Scan String F3 AF	Promoted to 64 bits.	32 bits	No GPR register results. See footnote[5]	
RET—Return from Call Near C2 C3	See “Near Branches in 64-Bit Mode” in Volume 1.			
	Promoted to 64 bits.	64 bits	Cannot encode.[6]	No GPR register results.
RET—Return from Call Far CB CA	Promoted to 64 bits.	32 bits	See “Control Transfers” in Volume 1 and “Control-Transfer Privilege Checks” in Volume 2.	
ROL—Rotate Left D1 /0 D3 /0 C1 /0	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
ROR—Rotate Right D1 /1 D3 /1 C1 /1	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
RSM—Resume from System Management Mode 0F AA	New SMM state-save area.	Not relevant.	See “System-Management Mode” in Volume 2.	
SAHF—Store AH into Flags 9E	Same as legacy mode.	Not relevant.	No GPR register results.	
SAL—Shift Arithmetic Left D1 /4 D3 /4 C1 /4	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
SAR—Shift Arithmetic Right D1 /7 D3 /7 C1 /7	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
SBB—Subtract with Borrow 19 1B 1D 81 /3 83 /3	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
SCAS, **SCASW**, **SCASD, SCASQ**—Scan String AF	Promoted to 64 bits.	32 bits	SCASD: Scan String Doublewords. Zero-extends 32-bit register results to 64 bits. See footnote[5]	SCASQ (new mnemonic): Scan String Quadwords. See footnote[5]
SFENCE—Store Fence 0F AE /7	Same as legacy mode.	Not relevant.	No GPR register results.	
SGDT—Store Global Descriptor Table Register 0F 01 /0	Promoted to 64 bits.	Operand size fixed at 64 bits.	No GPR register results. Stores 8-byte base and 2-byte limit.	
SHL—Shift Left D1 /4 D3 /4 C1 /4	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
SHLD—Shift Left Double 0F A4 0F A5	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
SHR—Shift Right D1 /5 D3 /5 C1 /5	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
SHRD—Shift Right Double 0F AC 0F AD	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	Uses 6-bit count.
SIDT—Store Interrupt Descriptor Table Register 0F 01 /1	Promoted to 64 bits.	Operand size fixed at 64 bits.	No GPR register results. Stores 8-byte base and 2-byte limit.	
SKINIT—Secure Init and Jump with Attestation 0F 01 DE	Same as legacy mode.	Not relevant	Zero-extends 32-bit register results to 64 bits.	
SLDT—Store Local Descriptor Table Register 0F 00 /0	Same as legacy mode.	32	Zero-extends 2-byte LDT selector to 64 bits.	
SMSW—Store Machine Status Word 0F 01 /4	Same as legacy mode.	32	Zero-extends 32-bit register results to 64 bits.	Stores 64-bit machine status word (CR0).
STC—Set Carry Flag F9	Same as legacy mode.	Not relevant.	No GPR register results.	
STD—Set Direction Flag FD	Same as legacy mode.	Not relevant.	No GPR register results.	

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
STGI—Set Global Interrupt Flag 0F 01 DC	Same as legacy mode.	Not relevant.	No GPR register results.	
STI - Set Interrupt Flag FB	Same as legacy mode.	Not relevant.	No GPR register results.	
STOS, **STOSW**, **STOSD, STOSQ**- Store String AB	Promoted to 64 bits.	32 bits	STOSD: Store String Doublewords. See footnote[5]	STOSQ (new mnemonic): Store String Quadwords. See footnote[5]
STR—Store Task Register 0F 00 /1	Same as legacy mode.	32	Zero-extends 2-byte TR selector to 64 bits.	
SUB—Subtract 29 2B 2D 81 /5 83 /5	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
SWAPGS—Swap GS Register with KernelGSbase MSR 0F 01 /7	New instruction, available only in 64-bit mode. (In other modes, this opcode is invalid.)	Not relevant.	See “SWAPGS Instruction” in Volume 2.	
SYSCALL—Fast System Call 0F 05	Promoted to 64 bits.	Not relevant.	See “SYSCALL and SYSRET Instructions” in Volume 2 for details.	

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
SYSENTER—System Call 0F 34	INVALID IN LONG MODE (invalid-opcode exception)			
SYSEXIT—System Return 0F 35	INVALID IN LONG MODE (invalid-opcode exception)			
SYSRET—Fast System Return 0F 07	Promoted to 64 bits.	32 bits	See “SYSCALL and SYSRET Instructions” in Volume 2 for details.	
TEST—Test Bits 85 A9 F7 /0	Promoted to 64 bits.	32 bits	No GPR register results.	
UD2—Undefined Operation 0F 0B	Same as legacy mode.	Not relevant.	No GPR register results.	
VERR—Verify Segment for Reads 0F 00 /4	Same as legacy mode.	Operand size fixed at 16 bits	No GPR register results.	
VERW—Verify Segment for Writes 0F 00 /5	Same as legacy mode.	Operand size fixed at 16 bits	No GPR register results.	
VMLOAD—Load State from VMCB 0F 01 DA	Same as legacy mode.	Not relevant.	No GPR register results.	
VMMCALL—Call VMM 0F 01 D9	Same as legacy mode.	Not relevant.	No GPR register results.	
VMRUN—Run Virtual Machine 0F 01 D8	Same as legacy mode.	Not relevant.	No GPR register results.	
VMSAVE—Save State to VMCB 0F 01 DB	Same as legacy mode.	Not relevant.	No GPR register results.	

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

Table B-1. Operations and Operands in 64-Bit Mode (continued)

Instruction and Opcode (hex)[1]	Type of Operation[2]	Default Operand Size[3]	For 32-Bit Operand Size[4]	For 64-Bit Operand Size[4]
WAIT—Wait for Interrupt 9B	Same as legacy mode.	Not relevant.	No GPR register results.	
WBINVD—Writeback and Invalidate All Caches 0F 09	Same as legacy mode.	Not relevant.	No GPR register results.	
WRMSR—Write to Model-Specific Register 0F 30	Same as legacy mode.	Not relevant.	No GPR register results. MSR[63:32] = RDX[31:0] MSR[31:0] = RAX[31:0]	
XADD—Exchange and Add 0F C1	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
XCHG—Exchange Register/Memory with Register 87 90	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	
XOR—Logical Exclusive OR 31 33 35 81 /6 83 /6	Promoted to 64 bits.	32 bits	Zero-extends 32-bit register results to 64 bits.	

Notes:

1. *See “General Rules for 64-Bit Mode” on page 555, for opcodes that do not appear in this table.*
2. *The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 555 for definitions of “Promoted to 64 bits” and related topics.*
3. *If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.*
4. *Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.*
5. *Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.*
6. *The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.*

B.3 Invalid and Reassigned Instructions in 64-Bit Mode

Table B-2 lists instructions that are illegal in 64-bit mode. Attempted use of these instructions generates an invalid-opcode exception (#UD).

Table B-2. Invalid Instructions in 64-Bit Mode

Mnemonic	Opcode (hex)	Description
AAA	37	ASCII Adjust After Addition
AAD	D5	ASCII Adjust Before Division
AAM	D4	ASCII Adjust After Multiply
AAS	3F	ASCII Adjust After Subtraction
BOUND	62	Check Array Bounds
CALL (far)	9A	Procedure Call Far (far absolute)
DAA	27	Decimal Adjust after Addition
DAS	2F	Decimal Adjust after Subtraction
INTO	CE	Interrupt to Overflow Vector
JMP (far)	EA	Jump Far (absolute)
LDS	C5	Load DS Far Pointer
LES	C4	Load ES Far Pointer
POP DS	1F	Pop Stack into DS Segment
POP ES	07	Pop Stack into ES Segment
POP SS	17	Pop Stack into SS Segment
POPA, POPAD	61	Pop All to GPR Words or Doublewords
PUSH CS	0E	Push CS Segment Selector onto Stack
PUSH DS	1E	Push DS Segment Selector onto Stack
PUSH ES	06	Push ES Segment Selector onto Stack
PUSH SS	16	Push SS Segment Selector onto Stack
PUSHA, PUSHAD	60	Push All to GPR Words or Doublewords
Redundant Grp1	82 /2	Redundant encoding of group1 Eb,Ib opcodes
SALC	D6	Set AL According to CF

Table B-3 lists instructions that are reassigned to different functions in 64-bit mode. Attempted use of these instructions generates the reassigned function.

Table B-3. Reassigned Instructions in 64-Bit Mode

Mnemonic	Opcode (hex)	Description
ARPL	63	Opcode for MOVSXD instruction in 64-bit mode. In all other modes, this is the Adjust Requestor Privilege Level instruction opcode.
DEC and INC	40-4F	REX prefixes in 64-bit mode. In all other modes, decrement by 1 and increment by 1.
LDS	C5	VEX Prefix. Introduces the VEX two-byte instruction encoding escape sequence.
LES	C4	VEX Prefix. Introduces the VEX three-byte instruction encoding escape sequence.

Table B-4 lists instructions that are illegal in long mode. Attempted use of these instructions generates an invalid-opcode exception (#UD).

Table B-4. Invalid Instructions in Long Mode

Mnemonic	Opcode (hex)	Description
SYSENTER	0F 34	System Call
SYSEXIT	0F 35	System Return

B.4 Instructions with 64-Bit Default Operand Size

In 64-bit mode, two groups of instructions default to 64-bit operand size without the need for a REX prefix:

- *Near branches* —CALL, J*cc*, JrCX, JMP, LOOP, and RET.
- *All instructions, except far branches, that implicitly reference the RSP*—CALL, ENTER, LEAVE, POP, PUSH, and RET (CALL and RET are in both groups of instructions).

Table B-5 lists these instructions.

Table B-5. Instructions Defaulting to 64-Bit Operand Size

Mnemonic	Opcode (hex)	Implicitly Reference RSP	Description
CALL	E8, FF /2	yes	Call Procedure Near
ENTER	C8	yes	Create Procedure Stack Frame
J*cc*	many	no	Jump Conditional Near
JMP	E9, EB, FF /4	no	Jump Near
LEAVE	C9	yes	Delete Procedure Stack Frame
LOOP	E2	no	Loop
LOOP*cc*	E0, E1	no	Loop Conditional
POP reg/mem	8F /0	yes	Pop Stack (register or memory)
POP reg	58-5F	yes	Pop Stack (register)
POP FS	0F A1	yes	Pop Stack into FS Segment Register
POP GS	0F A9	yes	Pop Stack into GS Segment Register
POPF, POPFD, POPFQ	9D	yes	Pop to rFLAGS Word, Doubleword, or Quadword
PUSH imm8	6A	yes	Push onto Stack (sign-extended byte)
PUSH imm32	68	yes	Push onto Stack (sign-extended doubleword)
PUSH reg/mem	FF /6	yes	Push onto Stack (register or memory)
PUSH reg	50-57	yes	Push onto Stack (register)
PUSH FS	0F A0	yes	Push FS Segment Register onto Stack
PUSH GS	0F A8	yes	Push GS Segment Register onto Stack
PUSHF, PUSHFD, PUSHFQ	9C	yes	Push rFLAGS Word, Doubleword, or Quadword onto Stack
RET	C2, C3	yes	Return From Call (near)

The 64-bit default operand size can be overridden to 16 bits using the 66h operand-size override. However, it is not possible to override the operand size to 32 bits because there is no 32-bit operand-size override prefix for 64-bit mode. See “Operand-Size Override Prefix” on page 7 for details.

B.5 Single-Byte INC and DEC Instructions in 64-Bit Mode

In 64-bit mode, the legacy encodings for the 16 single-byte INC and DEC instructions (one for each of the eight GPRs) are used to encode the REX prefix values, as described in “REX Prefix” on page 14. Therefore, these single-byte opcodes for INC and DEC are not available in 64-bit mode, although they are available in legacy and compatibility modes. The functionality of these INC and DEC instructions is still available in 64-bit mode, however, using the ModRM forms of those instructions (opcodes FF/0 and FF/1).

B.6 NOP in 64-Bit Mode

Programs written for the legacy x86 architecture commonly use opcode 90h (the XCHG EAX, EAX instruction) as a one-byte NOP. In 64-bit mode, the processor treats opcode 90h specially in order to preserve this legacy NOP use. Without special handling in 64-bit mode, the instruction would not be a true no-operation. Therefore, in 64-bit mode the processor treats XCHG EAX, EAX as a true NOP, regardless of operand size.

This special handling does not apply to the two-byte ModRM form of the XCHG instruction. Unless a 64-bit operand size is specified using a REX prefix byte, using the two byte form of XCHG to exchange a register with itself will not result in a no-operation because the default operation size is 32 bits in 64-bit mode.

B.7 Segment Override Prefixes in 64-Bit Mode

In 64-bit mode, the CS, DS, ES, SS segment-override prefixes have no effect. These four prefixes are no longer treated as segment-override prefixes in the context of multiple-prefix rules. Instead, they are treated as null prefixes.

The FS and GS segment-override prefixes are treated as true segment-override prefixes in 64-bit mode. Use of the FS and GS prefixes cause their respective segment bases to be added to the effective address calculation. See “FS and GS Registers in 64-Bit Mode” in Volume 2 for details.

Appendix C Differences Between Long Mode and Legacy Mode

Table C-1 summarizes the major differences between 64-bit mode and legacy protected mode. The third column indicates differences between 64-bit mode and legacy mode. The fourth column indicates whether that difference also applies to compatibility mode.

Table C-1. Differences Between Long Mode and Legacy Mode

Type	Subject	64-Bit Mode Difference	Applies To Compatibility Mode?
Application Programming	Addressing	RIP-relative addressing available	no
	Data and Address Sizes	Default data size is 32 bits	
		REX Prefix toggles data size to 64 bits	
		Default address size is 64 bits	
		Address size prefix toggles address size to 32 bits	
	Instruction Differences	Various opcodes are invalid or changed in 64-bit mode (see Table B-2 on page 581 and Table B-3 on page 582)	
		Various opcodes are invalid in long mode (see Table B-4 on page 582)	yes
		MOV reg,imm32 becomes MOV reg,imm64 (with REX operand size prefix)	no
		REX is always enabled	
		Direct-offset forms of MOV to or from accumulator become 64-bit offsets	
		MOVD extended to MOV 64 bits between MMX registers and long GPRs (with REX operand-size prefix)	

Table C-1. Differences Between Long Mode and Legacy Mode (continued)

Type	Subject	64-Bit Mode Difference	Applies To Compatibility Mode?
System Programming	x86 Modes	Real and virtual-8086 modes not supported	yes
	Task Switching	Task switching not supported	yes
	Addressing	64-bit virtual addresses	yes
		4-level paging structures	
		PAE must always be enabled	
	Segmentation	CS, DS, ES, SS segment bases are ignored	no
		CS, DS, ES, FS, GS, SS segment limits are ignored	
		CS, DS, ES, SS Segment prefixes are ignored	
	Exception and Interrupt Handling	All pushes are 8 bytes	yes
		16-bit interrupt and trap gates are illegal	
		32-bit interrupt and trap gates are redefined as 64-bit gates and are expanded to 16 bytes	
		SS is set to null on stack switch	
		SS:RSP is pushed unconditionally	
	Call Gates	All pushes are 8 bytes	yes
		16-bit call gates are illegal	
		32-bit call gate type is redefined as 64-bit call gate and is expanded to 16 bytes.	
		SS is set to null on stack switch	
	System-Descriptor Registers	GDT, IDT, LDT, TR base registers expanded to 64 bits	yes
	System-Descriptor Table Entries and Pseudo-descriptors	LGDT and LIDT use expanded 10-byte pseudo-descriptors.	no
		LLDT and LTR use expanded 16-byte table entries.	

Appendix D Instruction Subsets and CPUID Feature Flags

This appendix provides information that can be used to determine if a specific instruction within the AMD64 instruction-set architecture (ISA) is supported on a processor.

Originally the x86 ISA was composed of a set of instructions from the general-purpose and system instruction groups. This set forms the base of the AMD64 ISA. As the ISA expanded over time, new instructions were added. Each addition constituted either a single instruction or a set of instructions and each addition was assigned a specific processor feature flag.

Although most current processor products support the entire ISA, support for each added instruction or instruction subset is optional and must be confirmed by testing the corresponding feature flag. The presence of a particular instruction or subset is indicated by the corresponding feature flag being set. A feature flag is a single bit value located at a specific bit position within the 32-bit value returned in a register as a result of executing the CPUID instruction.

For more information on using the CPUID instruction, see the instruction reference page for CPUID on page 165. For a comprehensive list of processor feature flags accessed using the CPUID instruction, see Appendix E, “Obtaining Processor Information Via the CPUID Instruction” on page 593.

D.1 Instruction Set Overview

The AMD64 ISA can be organized into five instruction groups:

1. General-purpose instructions

   These instructions operate on the general-purpose registers (GP registers) and can be used at all privilege levels. This group includes instructions to load and store the contents of a GP register to and from memory, move values between the GP registers, and perform arithmetic and logical operations on the contents of the registers.

2. System instructions

   These instructions provide the means to manipulate the processor operating mode, access processor resources, handle program and system errors, and manage system memory. Many of these instructions require privilege level 0 to execute.

3. x87 instructions

   These instructions are available at all privilege levels and include legacy floating-point instructions that use the ST(0)–ST(7) stack registers (FPR0–FPR7 physical registers) and internally use extended precision (80-bit) binary floating-point representation and operations.

4. 64-bit media Instructions

   These instructions are available at all privilege levels and perform vector operations on packed integer and floating-point values held in the 64-bit MMX™ registers. The MMX register set overlays the FPR0–FPR7 physical registers. This group is composed of the MMX and 3DNow!™ instruction subsets and was subsequently expanded by the MMX and 3DNow! extensions subsets.

5. SSE instructions

   The SSE instructions operate on packed integer and floating-point values held in the XMM / YMM registers. SSE includes the original Streaming SIMD Extensions, all the subsequent named SSE subsets, and the AVX, XOP, and AES instructions.

Figure D-1 on page 589 represents the relationship between the five major instruction groups and the named instruction subsets. Circles represent the instruction subsets. These include the base instruction set labeled “Base Instructions” in the diagram and the named subsets. The diagram omits individual optional instructions and some of the minor named instruction subsets. Dashed-line polygons represent the instruction groups.

Note that the 128-bit and 256-bit media instructions are referred to collectively as the Streaming SIMD Extensions (SSE). This is also the name of the original SSE subset. In the diagram the original SSE subset is labeled “SSE1 Instructions.” Collectively the 64-bit media and the SSE instructions make up the single instruction / multiple data (SIMD) group (labeled “SIMD Instructions” in the diagram).

The overlapping of the SSE and 64-bit media instruction subsets indicates that these subsets share some common mnemonics. However, these common mnemonics either have distinct opcodes for each subset or they take operands in both the MMX and XMM register sets.

The horizontal axis of Figure D-1 shows how the subsets have evolved over time.

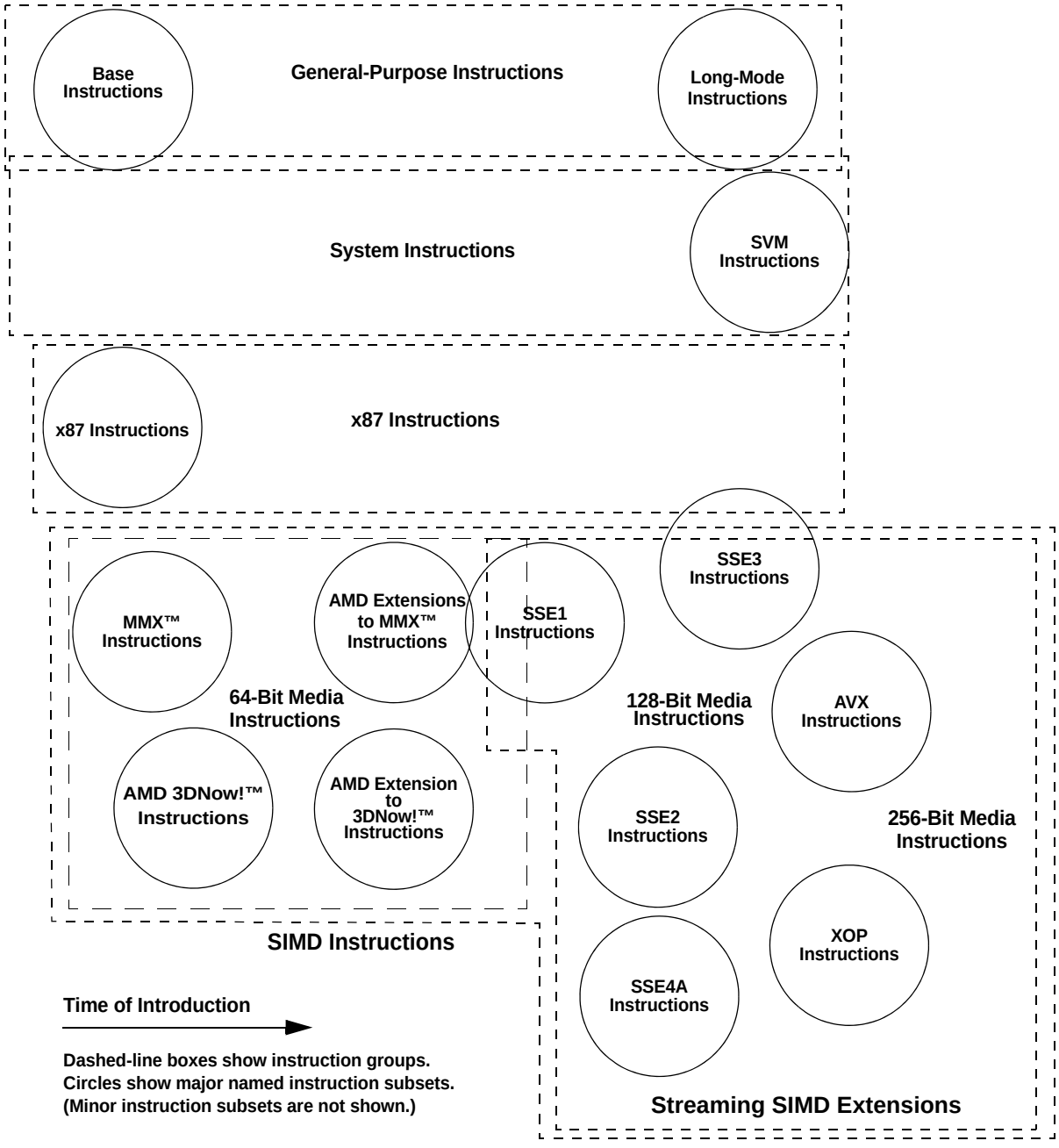


Figure D-1. AMD64 ISA Instruction Subsets

D.2 CPUID Feature Flags Related to Instruction Support

Only a subset of the CPUID feature flags provides information related to instruction support.

The feature flags related to supported instruction subsets are accessed via the *standard* function number 0000_0001h, the *extended* function number 8000_0001h, and the *structured extended* function number 0000_0007h.

The following table lists all flags related to instruction support. Entries for each flag provide the instruction or instruction subset corresponding to the flag, the CPUID function that must be executed to access the flag, and the bit position of the flag in the return value.

Table D-1. Feature Flags for Instruction / Instruction Subset Support

Feature Flag	Instruction or Subset	CPUID Function[1]	Feature Flag Bit Position[2]
3DNow	3DNow!	extended	EDX[31]
3DNowExt	3DNow! Extensions	extended	EDX[30]
3DNowPrefetch	PREFETCH / PREFETCHW	extended	ECX[8], EDX[29], or EDX[31]
ABM	LZCNT	extended	ECX[5]
ADX	ADCX, ADOX	0000_0007_0	EBX[19]
AES	AES	standard	ECX[25]
AVX	AVX	standard	ECX[28]
AVX2	AVX2	0000_0007_0	EBX[5]
BASE	Base Instruction set	—	—
BMI1	Bit Manipulation, group 1	0000_0007_0	EBX[3]
BMI2	Bit Manipulation, group 2	0000_0007_0	EBX[8]
CET_SS	Shadow Stack, CLRSSBSY, INCSSP, RDSSP, RSTORSSP, SAVEPREVSSP, SETSSBSY, WRSS, WRUSS	0000_0007_0	ECX[7]
CLFLOPT	CLFLUSHOPT	0000_0007_0	EBX[23]
CLFSH	CLFLUSH, CLWB	standard	EDX[19]
CLWB	CLWB	0000_0007_0	EBX[24]
CLZERO	CLZERO	8000_0008	EBX[0]
CMPXCHG8B	CMPXCHG8B	both	EDX[8]
CMPXCHG16B	CMPXCHG16B	standard	ECX[13]
CMOV	CMOV*cc*	both	EDX[15]

Notes:

1. *standard* = Fn0000_0001h; *extended* = Fn 8000_0001h; *both* means that both standard and extended CPUID functions return the same feature flag in the same bit position of the return value. For functions of the form xxxx-_xxxx_x, the trailing digit is the value required in ECX.
2. Register and bit position of the return value that corresponds to the feature flag.
3. FCMOVcc instruction is supported if x87 and CMOVcc instructions are both supported.
4. XSAVE (and related) instructions require separate enablement.

Table D-1. Feature Flags for Instruction / Instruction Subset Support (continued)

Feature Flag	Instruction or Subset	CPUID Function[1]	Feature Flag Bit Position[2]
F16C	16-bit floating-point conversion	standard	ECX[29]
FMA	FMA	standard	ECX[12]
FMA4	FMA4	extended	ECX[16]
FPU	x87	both	EDX[0]
FSGSBASE	FS and GS base read and write	0000_0007_0	EBX[0]
FXSR	FXSAVE / FXRSTOR	both	EDX[24]
INVLPGB	INVLPGB, TLBSYNC	8000_0008	EBX[3]
INVPCID	INVPCID	0000_0007_0	EBX[10]
LahfSahf	LAHF / SAHF	extended	ECX[0]
LM	Long Mode	extended	EDX[29]
MCOMMIT	MCOMMIT	8000_0008	EBX[8]
MMX	MMX	both	EDX[23]
MmxExt	MMX Extensions	extended	EDX[22]
MONITOR	MONITOR / MWAIT	standard	ECX[3]
MONITORX	MONITORX / MWAITX	extended	ECX[29]
MOVBE	MOVBE	standard	ECX[22]
MSR	RDMSR / WRMSR	both	EDX[5]
OSPKE	RDPKRU, WRPKRU	0000_0007_0	ECX[4]
PCLMULQDQ	PCLMULQDQ	standard	ECX[1]
POPCNT	POPCNT	standard	ECX[23]
RDPID	RDPID	0000_0007_0	ECX[22]
RDPRU	RDPRU	8000_0008	EBX[4]
RDRAND	RDRAND	standard	ECX[30]
RDTSCP	RDTSCP	extended	EDX[27]
RDSEED	RDSEED	0000_0007_0	EBX[18]
SevEs	VMGEXIT	8000_001F	EAX[3]
SHA	SHA	0000_0007_0	EBX[29]
SKINIT	SKINIT / STGI	extended	ECX[12]
SMAP	CLAC, STAC	0000_0007_0	EBX[20]
SNP	PSMASH, PVALIDATE, RMPADJUST, RMPUPDATE	8000_001F	EAX[4]
SSE	SSE1	standard	EDX[25]
SSE2	SSE2	standard	EDX[26]

Notes:

1. *standard* = Fn0000_0001h; *extended* = Fn 8000_0001h; *both* means that both standard and extended CPUID functions return the same feature flag in the same bit position of the return value. For functions of the form xxxx-_xxxx_x, the trailing digit is the value required in ECX.
2. Register and bit position of the return value that corresponds to the feature flag.
3. FCMOVcc instruction is supported if x87 and CMOVcc instructions are both supported.
4. XSAVE (and related) instructions require separate enablement.

Table D-1. Feature Flags for Instruction / Instruction Subset Support (continued)

Feature Flag	Instruction or Subset	CPUID Function[1]	Feature Flag Bit Position[2]
SSE3	SSE3	standard	ECX[0]
SSSE3	SSSE3	standard	ECX[9]
SSE4A	SSE4A	extended	ECX[6]
SSE41	SSE4.1	standard	ECX[19]
SSE42	SSE4.2	standard	ECX[20]
SVM	Secure Virtual Machine	extended	ECX[2]
SysCallSysRet	SYSCALL / SYSRET	extended	EDX[11]
SysEnterSysExit	SYSENTER / SYSEXIT	standard	EDX[11]
TBM	Trailing bit manipulation	extended	ECX[21]
TSC	RDTSC	both	EDX[4]
VAES	VAES 256-bit instructions	0000_0007_0	ECX[9]
VPCMULQDQ	VPCMULQDQ 256-bit instructions	0000_0007_0	ECX[10]
WBNOINVD	WBNOINVD	8000_0008	EBX[9]
x87 && CMOV	FCMOV*cc*[3]	both	EDX[0] && EDX[15]
XGETBV w/ ECX=1	XGETBV w/ ECX=1	0000_000D_1	EAX[2]
XOP	XOP	extended	ECX[11]
XSAVE	XSAVE / XRSTOR[4]	standard	ECX[26]
XSAVEC	XSAVEC	0000_000D_1	EAX[1]
XSAVEOPT	XSAVEOPT	0000_000D_1	EAX[0]
XSAVES/XRSTORS	XSAVES / XRSTORS	0000_000D_1	EAX[3]

Notes:

1. *standard* = Fn0000_0001h; *extended* = Fn 8000_0001h; *both* means that both standard and extended CPUID functions return the same feature flag in the same bit position of the return value. For functions of the form xxxx-_xxxx_x, the trailing digit is the value required in ECX.
2. Register and bit position of the return value that corresponds to the feature flag.
3. FCMOVcc instruction is supported if x87 and CMOVcc instructions are both supported.
4. XSAVE (and related) instructions require separate enablement.

Appendix E Obtaining Processor Information Via the CPUID Instruction

This appendix specifies the information that software can obtain about the processor on which it is running by executing the CPUID instruction. The information in this appendix supersedes the contents of the *CPUID Specification*, order #25481, which is now obsolete.

The CPUID instruction is described on page 165. This appendix does not replace the CPUID instruction reference information presented there.

The CPUID instruction behaves much like a function call. Parameters are passed to the instruction via registers and on execution the instruction loads specific registers with return values. These return values can be interpreted by software based on the field definitions and their assigned meanings.

The first input parameter is the *function number* which is passed to the instruction via the EAX register. Some functions also accept a second input parameter passed via the ECX register. Values are returned via the EAX, EBX, ECX, and EDX registers. Software should not assume that any values written to these registers prior to the execution of CPUID instruction will be retained after the instruction executes (even those that are marked reserved).

The description of each return value breaks the value down into one or more named *fields* which represent a bit position or contiguous range of bits. All bit positions that are not defined as fields are reserved. The value of bits within reserved ranges cannot be relied upon to be zero. Software must mask off all reserved bits in the return value prior to making any value comparisons of represented information.

This appendix applies to all AMD processors with a family designation of 0Fh or greater.

E.1 Special Notational Conventions

The following special notation conventions are used in this appendix:

- The notation (standard throughout this APM) for representing the function number, optional input parmeter, and the information returned is as follows:

  CPUID Fn*XXXX_XXXX_RRR*[*FieldName*]_x*YYY*.

  Where:

  - *XXXX_XXXX* is the function number represented in hexadecimal (passed to the instruction in EAX).
  - *RRR* is one of {EDX, ECX, EBX, EAX} and represents a register holding a return value.
  - *YYY* represents the optional input parameter passed in the ECX register expressed as a hexadecimal number. If this parameter is not used, the characters represented by _x*YYY* are ommitted from the notation.

  - *FieldName* identifies a specific named element of processor information represented by a specific bit range (1 or more bits wide) within the *RRR* register.
- The notation CPUID Fn*XXXX_XXXX _RRR* is used when refering to one of the registers that holds information returned by the instruction.
- The notation CPUID Fn*XXXX_XXXX* or Fn*XXXX_XXXX* is used to refer to a specific function number.
- Most one-bit fields indicate support or non-support of a specific processor feature. By convention, (unless otherwise noted) a value of 1 means that the feature is supported by the processor and a value of 0 means that the feature is not supported by the processor.

E.2 Standard and Extended Function Numbers

The CPUID instruction supports two sets or ranges of function numbers: standard and extended.

- The smallest function number of the standard function range is Fn0000_0000. The largest function number of the standard function range, for a particular implementation, is returned in CPUID Fn0000_0000_EAX.
- The smallest function number of the extended function range is Fn8000_0000. The largest function number of the extended function range, for a particular implementation, is returned in CPUID Fn8000_0000_EAX.

E.3 Standard Feature Function Numbers

This section describes each of the defined CPUID functions in the standard range.

E.3.1 Function 0h—Maximum Standard Function Number and Vendor String

This function number provides information about the maximum standard function number supported on this processor and a string that identifies the vendor of the product.

CPUID Fn0000_0000_EAX Largest Standard Function Number

The value returned in EAX provides the largest standard function number supported by this processor.

Bits	Field Name	Description
31:0	LFuncStd	Largest standard function. The largest CPUID standard function input value supported by the processor implementation.

CPUID Fn0000_0000_E[D,C,B]X Processor Vendor

The values returned in EBX, EDX, and ECX together provide a 12-character string identifying the vendor of this processor. Each register supplies 4 characters. The leftmost character of each substring

is stored in the least significant bit position in the register. The string is the concatenation of the contents of EBX, EDX, and ECX in left to right order. No null terminator is included in the string.

CPUID Fn8000_0000_E[D,C,B]X return the same values as this function.

Bits	Field Name	Description
31:0	Vendor	Four characters of the 12-byte character string (encoded in ASCII) “AuthenticAMD”. See Table E-1 below.

Table E-1. CPUID Fn0000_0000_E[D,C,B]X values

Register	Value	Description
CPUID Fn0000_0000_EBX	6874_7541h	The ASCII characters “h t u A”.
CPUID Fn0000_0000_ECX	444D_4163h	The ASCII characters “D M A c”.
CPUID Fn0000_0000_EDX	6974_6E65h	The ASCII characters “i t n e”.

E.3.2 Function 1h—Processor and Processor Feature Identifiers

This function number identifies the processor family, model, and stepping and provides feature support information.

CPUID Fn0000_0001_EAX Family, Model, Stepping Identifiers

The value returned in EAX provides the family, model, and stepping identifiers. Three values are used by software to identify a processor: Family, Model, and Stepping.

Bits	Field Name	Description
31:28	—	Reserved.
27:20	ExtFamily	Processor extended family. See above for definition of Family[7:0].
19:16	ExtModel	Processor extended model. See above for definition of Model[7:0].
15:12	—	Reserved.
11:8	BaseFamily	Base processor family. See above for definition of Family[7:0].
7:4	BaseModel	Base processor model. See above for definition of Model[7:0].
3:0	Stepping	Processor stepping. Processor stepping (revision) for a specific model.

The processor *Family* identifies one or more processors as belonging to a group that possesses some common definition for software or hardware purposes. The *Model* specifies one instance of a processor family. The *Stepping* identifies a particular version of a specific model. Therefore, Family, Model and Stepping, when taken together, form a unique identification or signature for a processor.

The **Family** is an 8-bit value and is defined as: **Family[7:0] =** ({0000b,BaseFamily[3:0]} + ExtFamily[7:0]). For example, if BaseFamily[3:0] = Fh and ExtFamily[7:0] = 01h, then Family[7:0] =

10h. If BaseFamily[3:0] is less than Fh, then ExtFamily is reserved and Family is equal to BaseFamily[3:0].

Model is an 8-bit value and is defined as: **Model[7:0]** = {ExtModel[3:0],BaseModel[3:0]}. For example, if ExtModel[3:0] = Eh and BaseModel[3:0] = 8h, then Model[7:0] = E8h. If BaseFamily[3:0] is less than 0Fh, then ExtModel is reserved and Model is equal to BaseModel[3:0].

The value returned by CPUID Fn8000_0001_EAX is equivalent to CPUID Fn0000_0001_EAX.

CPUID Fn0000_0001_EBX LocalApicId, LogicalProcessorCount, CLFlush

The value returned in EBX provides miscellaneous information regarding the processor brand, the number of logical threads per processor socket, the CLFLUSH instruction, and APIC.

Bits	Field Name	Description
31:24	LocalApicId	Initial local APIC physical ID. The 8-bit value assigned to the local APIC physical ID register at power-up. Some of the bits of LocalApicId represent the core within a processor and other bits represent the processor ID. See the APIC20 “APIC ID” register in the processor BKDG or PPR for details.
23:16	LogicalProcessor Count	Logical processor count. If CPUID Fn0000_0001_EDX[HTT] = 1 then LogicalProcessorCount is the number of logic processors per package. If CPUID Fn0000_0001_EDX[HTT] = 0 then LogicalProcessorCount is reserved. See E.5.1 [Legacy Method].
15:8	CLFlush	CLFLUSH size. Specifies the size of a cache line in quadwords flushed by the CLFLUSH instruction. See “CLFLUSH” in APM3.
7:0	8BitBrandId	8-bit brand ID. This field, in conjunction with CPUID Fn8000_0001_EBX[BrandId], is used by the system firmware to generate the processor name string. See the appropriate processor revision guide for how to program the processor name string.

CPUID Fn0000_0001_ECX Feature Identifiers

The value returned in ECX contains the following miscellaneous feature identifiers:

Bits	Field Name	Description
31	—	RAZ. Reserved for use by hypervisor to indicate guest status.
30	RDRAND	RDRAND instruction support.
29	F16C	Half-precision convert instruction support. See "Half-Precision Floating-Point Conversion" in APM1 and listings for individual F16C instructions in APM5.
28	AVX	AVX instruction support. See APM4.
27	OSXSAVE	XSAVE (and related) instructions are enabled. See “OSXSAVE” in APM2. .
26	XSAVE	XSAVE (and related) instructions are supported by hardware. See “XSAVE/XRSTOR Instructions” in APM2.

Bits	Field Name	Description
25	AES	AES instruction support. See “AES Instructions” in APM4.
24	—	Reserved.
23	POPCNT	POPCNT instruction. See “POPCNT” in APM3.
22		MOVBE: MOVBE instruction support.
21	—	Reserved.
20	SSE42	SSE4.2 instruction support. "Determining Media and x87 Feature Support" in APM2 and individual SSE4.2 instruction listings in APM4.
19	SSE41	SSE4.1 instruction support. See individual instruction listings in APM4. .
18:14	—	Reserved.
13	CMPXCHG16B	CMPXCHG16B instruction support. See “CMPXCHG16B” in APM3.
12	FMA	FMA instruction support.
11:10	—	Reserved.
9	SSSE3	Supplemental SSE3 instruction support.
8:4	—	Reserved.
3	MONITOR	MONITOR/MWAIT instructions. See “MONITOR” and “MWAIT” in APM3.
2	—	Reserved.
1	PCLMULQDQ	PCLMULQDQ instruction support. See instruction reference page for the PCLMULQDQ / VPCLMULQDQ instruction in APM4.
0	SSE3	SSE3 instruction support. See Appendix D “Instruction Subsets and CPUID Feature Sets” in APM3 for the list of instructions covered by the SSE3 feature bit. See APM4 for the definition of the SSE3 instructions.

CPUID Fn0000_0001_EDX Feature Identifiers

The value returned in EDX contains the following miscellaneous feature identifiers:

Bits	Field Name	Description
31:29	—	Reserved.
28	HTT	Hyper-threading technology. Indicates either that there is more than one thread per core or more than one core per compute unit.See “Legacy Method” on page 633.
27	—	Reserved.
26	SSE2	SSE2 instruction support. See Appendix D “CPUID Feature Sets” in APM3.
25	SSE	SSE instruction support. See Appendix D “CPUID Feature Sets” in APM3 appendix and “64-Bit Media Programming” in APM1.
24	FXSR	FXSAVE and FXRSTOR instructions. See “FXSAVE” and “FXRSTOR” in APM5.
23	MMX	MMX™ instructions. See Appendix D “CPUID Feature Sets” in APM3 and “128-Bit Media and Scientific Programming” in APM1.
22:20	—	Reserved.
19	CLFSH	CLFLUSH instruction support. See “CLFLUSH” in APM3.

Bits	Field Name	Description
18	—	Reserved.
17	PSE36	Page-size extensions. The PDE[20:13] supplies physical address [39:32]. See “Page Translation and Protection” in APM2.
16	PAT	Page attribute table. See “Page-Attribute Table Mechanism” in APM2.
15	CMOV	Conditional move instructions. See “CMOV”, “FCMOV” in APM3.
14	MCA	Machine check architecture. See “Machine Check Mechanism” in APM2.
13	PGE	Page global extension. See “Page Translation and Protection” in APM2.
12	MTRR	Memory-type range registers. See “Page Translation and Protection” in APM2.
11	SysEnterSysExit	SYSENTER and SYSEXIT instructions. See “SYSENTER”, “SYSEXIT“ in APM3.
10	—	Reserved.
9	APIC	Avanced programmable interrupt controller. Indicates APIC exists and is enabled. See “Exceptions and Interrupts” in APM2.
8	CMPXCHG8B	CMPXCHG8B instruction. See “CMPXCHG8B” in APM3.
7	MCE	Machine check exception. See “Machine Check Mechanism” in APM2.
6	PAE	Physical-address extensions. Indicates support for physical addresses [3] 32b. Number of physical address bits above 32b is implementation specific. See “Page Translation and Protection” in APM2.
5	MSR	AMD model-specific registers. Indicates support for AMD model-specific registers (MSRs), with RDMSR and WRMSR instructions. See “Model Specific Registers” in APM2.
4	TSC	Time stamp counter. RDTSC and RDTSCP instruction support. See “Debug and Performance Resources” in APM2.
3	PSE	Page-size extensions. See “Page Translation and Protection” in APM2.
2	DE	Debugging extensions. See “Debug and Performance Resources” in APM2.
1	VME	Virtual-mode enhancements. CR4.VME, CR4.PVI, software interrupt indirection, expansion of the TSS with the software, indirection bitmap, EFLAGS.VIF, EFLAGS.VIP. See “System Resources” in APM2.
0	FPU	x87 floating point unit on-chip. See “x87 Floating Point Programming” in APM1.

E.3.3 Functions 2h–4h—Reserved

CPUID Fn0000_000[4:2] Reserved

These function numbers are reserved.

E.3.4 Function 5h—Monitor and MWait Features

This function provides feature identifiers for the MONITOR and MWAIT instructions. For more information see the description of the MONITOR instruction on page 414 and the MWAIT instruction on page 420.

CPUID Fn0000_0005_EAX Monitor/MWait

The value returned in EAX provides the following information:

Bits	Field Name	Description
31:16	—	Reserved.
15:0	MonLineSizeMin	Smallest monitor-line size in bytes.

CPUID Fn0000_0005_EBX Monitor/MWait

The value returned in EBX provides the following information:

Bits	Field Name	Description
31:16	—	Reserved.
15:0	MonLineSizeMax	Largest monitor-line size in bytes.

CPUID Fn0000_0005_ECX Monitor/MWait

The value returned in ECX provides the following information:

Bits	Field Name	Description
31:2	—	Reserved.
1	IBE	Interrupt break-event. Indicates MWAIT can use ECX bit 0 to allow interrupts to cause an exit from the monitor event pending state, even if EFLAGS.IF=0.
0	EMX	Enumerate MONITOR/MWAIT extensions: Indicates enumeration MONITOR/MWAIT extensions are supported.

CPUID Fn0000_0005_EDX Monitor/MWait

The value returned in EDX is undefined and is reserved.

E.3.5 Function 6h—Power Management Related Features

This function provides information about the local APIC timer timebase and the effective frequency interface for the processor.

CPUID Fn0000_0006_EAX Local APIC Timer Invariance

The value returned in EAX is undefined and is reserved.

Bits	Field Name	Description
31:3	—	Reserved.
2	ARAT	If set, indicates that the timebase for the local APIC timer is not affected by processor p-state.
1:0	—	Reserved.

CPUID Fn0000_0006_EBX Reserved

The value returned in EBX is undefined and is reserved.

CPUID Fn0000_0006_ECX Effective Processor Frequency Interface

The value returned in ECX indicates support of the processor effective frequency interface. For more information on this feature, see "Determining Processor Effective Frequency" in APM2.

Bits	Field Name	Description
31:1	—	Reserved.
0	EffFreq	Effective frequency interface support. If set, indicates presence of MSR0000_00E7 (MPERF) and MSR0000_00E8 (APERF).

CPUID Fn0000_0006_EDX Reserved

The value returned in EDX is undefined and is reserved.

E.3.6 Function 7h—Structured Extended Feature Identifiers

CPUID Fn0000_0007_EAX_x0 Structured Extended Feature Identifiers (ECX=0)

Bits	Field Name	Description
31:0	MaxSubFn	Returns the number of subfunctions supported.

CPUID Fn0000_0007_EBX_x0 Structured Extended Feature Identifiers (ECX=0)

Bits	Field Name	Description
31:30	—	Reserved.
29	SHA	Secure Hash Algorithm instruction extension.
28:25	—	Reserved.
24	CLWB	CLWB instruction support.
23	CLFLUSHOPT	CLFLUSHOPT instruction support.
22	RDPID	RDPID instruction and TSC_AUX MSR support.
21	—	Reserved.

Bits	Field Name	Description
20	SMAP	Supervisor mode access prevention.
19	ADX	ADCX, ADOX instruction support.
18	RDSEED	RDSEED instruction support.
17:11	—	Reserved.
10	INVPCID	INVPCID instruction support.
9	—	Reserved.
8	BMI2	Bit manipulation group 2 instruction support.
7	SMEP	Supervisor mode execution prevention.
6	—	Reserved.
5	AVX2	AVX2 instruction subset support.
4	—	Reserved.
3	BMI1	Bit manipulation group 1 instruction support.
2:1	—	Reserved.
0	FSGSBASE	FS and GS base read/write instruction support.

CPUID Fn0000_0007_ECX_x0 Structured Extended Feature Identifiers (ECX=0)

Bits	Field Name	Description
31:11	—	Reserved.
10	VPCMULQDQ	Support for VPCLMULQDQ 256-bit instruction.
9	VAES	Support for VAES 256-bit instructions.
8	—	Reserved.
7	CET_SS	Shadow Stacks supported.
6:5	—	Reserved.
4	OSPKE	OS has enabled Memory Protection Keys and use of the RDPKRU/WRPKRU instructions by setting CR4.PKE=1.
3	PKU	Memory Protection Keys supported.
2	UMIP	User mode instruction prevention support.
1:0	—	Reserved.

CPUID Fn0000_0007_EDX_x0 Structured Extended Feature Identifiers (ECX=0)

Bits	Field Name	Description
31:0	—	Reserved.

E.3.7 Functions 8h–Ah—Reserved

E.3.8 Function Bh — Extended Topology Enumeration

CPUID Fn0000_000B enumerates each level in the processor's topological hierarchy. The level number is specified by the input value passed in the ECX register.

If this function is executed with an unimplemented level (passed in ECX), the instruction returns all zeros in the EAX register.

Subfunction 0 of Fn0000_000B - Thread Level

Subfunction 0 provides information about the thread-level topology.

CPUID Fn0000_000B_EAX_x0 Extended Topology Enumeration (ECX=0)

Bits	Field Name	Description
31:5	—	Reserved.
4:0	ThreadMaskWidth	Number of bits to shift x2APIC_ID right to get to the topology ID of the next level

CPUID Fn0000_000B_EBX_x0 Extended Topology Enumeration (ECX=0)

Bits	Field Name	Description
31:16	—	Reserved.
15:0		Number of threads in a core

CPUID Fn0000_000B_ECX_x0 Extended Topology Enumeration (ECX=0)

Bits	Field Name	Description
31:16	—	Reserved.
15:8	level number	returns '1' indicating thread level
7:0	ECX input value	returns '0'

CPUID Fn0000_000B_EDX_x0 Extended Topology Enumeration (ECX=0)

Bits	Field Name	Description
31:0	x2APIC_ID	32-bit Extended APIC_ID

Subfunction 1 of Fn0000_000B - Core Level

Subfunction 1 provides information about the core-level topology.

CPUID Fn0000_000B_EAX_x1 Extended Topology Enumeration (ECX=1)

Bits	Field Name	Description
31:5	—	Reserved.
4:0	CoreMaskWidth	Number of bits to shift x2APIC_ID right to get to the topology ID of the next level

CPUID Fn0000_000B_EBX_x1 Extended Topology Enumeration (ECX=1)

Bits	Field Name	Description
31:16	—	Reserved.
15:0		Number of logical cores in socket

CPUID Fn0000_000B_ECX_x1 Extended Topology Enumeration (ECX=1)

Bits	Field Name	Description
31:16	—	Reserved.
15:8	level numbers	returns ‘2’, indicating core-level
7:0	ECX input value	returns ‘1’

CPUID Fn0000_000B_EDX_x1 Extended Topology Enumeration (ECX=1)

Bits	Field Name	Description
31:0	x2APIC_ID	32-bit Extended APIC_ID

E.3.9 Function Ch—Reserved

E.3.10 Function Dh—Processor Extended State Enumeration

The XSAVE / XRSTOR instructions are used to save and restore x87/MMX FPU and SSE processor state. These instructions allow processor state associated with specific architected features to be selectively saved and restored. This function provides information about extended state support and save area size requirements.

The function has a number of subfunctions specified by the input value passed to the CPUID instruction in the ECX register. If CPUID Fn0000_000D is executed with an unimplemented subfunction (passed in ECX), the instruction returns all zeros in the EAX, EBX, ECX, and EDX registers.

Subfunction 0 of Fn0000_000D

Subfunction 0 provides information about features within the extended processor state management architecture that are supported by the processor.

CPUID Fn0000_000D_EAX_x0 Processor Extended State Enumeration (ECX=0)

The value returned in EAX provides a bit mask specifying which of the features defined by the extended processor state architecture are supported by the processor.

Bits	Field Name	Description
31:0	XFeatureSupportedMask[31:0]	Reports the valid bit positions for the lower 32 bits of the XFeatureEnabledMask register. If a bit is set, the corresponding feature is supported. See “XSAVE/XRSTOR Instructions” in APM2.

CPUID Fn0000_000D_EBX_x0 Processor Extended State Enumeration (ECX=0)

The value returned in EBX gives the save area size requirement in bytes based on the features currently enabled in the XFEATURE_ENABLED_MASK (XCR0).

Bits	Field Name	Description
31:0	XFeatureEnabledSizeMax	Size in bytes of XSAVE/XRSTOR area for the currently enabled features in XCR0.

CPUID Fn0000_000D_ECX_x0 Processor Extended State Enumeration (ECX=0)

The value returned in ECX gives the save area size requirement in bytes for all extended state management features supported by the processor (whether enabled or not).

Bits	Field Name	Description
31:0	XFeatureSupportedSizeMax	Size in bytes of XSAVE/XRSTOR area for all features that the logical processor supports. See XFeatureEnabledSizeMax.

CPUID Fn0000_000D_EDX_x0 Processor Extended State Enumeration (ECX=0)

The value returned in EDX provides a bit mask specifying which of the features defined by the extended processor state architecture are supported by the processor.

Bits	Field Name	Description
31:0	XFeatureSupportedMask[63:32]	Reports the valid bit positions for the upper 32 bits of the XFeatureEnabledMask register. If a bit is set, the corresponding feature is supported.

See “XSAVE/XRSTOR Instructions” in APM2 and reference pages for the individual instructions in APM4.

Subfunction 1 of Fn0000_000D

Subfunction 1 provides additional information about features within the extended processor state management architecture that are supported by the processor.

CPUID Fn0000_000D_EAX_x1 Processor Extended State Enumeration (ECX=1)

Bits	Field Name	Description
31:4		Reserved.
3	XSAVES	XSAVES, XRSTOR, and XSS are supported.
2	XGETBV	XGETBV with ECX = 1 supported.
1	XSAVEC	XSAVEC and compact XRSTOR supported.
0	XSAVEOPT	XSAVEOPT is available.

CPUID Fn0000_000D_EBX_x1 Processor Extended State Enumeration (ECX=1)

The value returned on EBX represents the fixed size of the save area (240h) plus the state size of each enabled extended feature:

EBX = 0240h
+ ((XCR0[AVX] == 1) ? 0000_0100h : 0)
+ ((XCR0[MPK] == 1) ? 0000_0008h : 0)
+ ((XSS[CET_U] == 1) ? 0000_0010h : 0)
+ ((XSS[CET_S] == 1) ? 0000_0018h : 0)

CPUID Fn0000_000D_ECX_x1 Processor Extended State Enumeration (ECX=1)

The value returned on ECX returns a "1" for each bit that is settable in the XSS MSR. The following bits are defined:

Bits	Field Name	Description
31:13	—	Reserved.
12	CET_S	CET supervisor.
11	CET_U	CET user state.
10:0	—	Reserved

CPUID Fn0000_000D_EDX_x1 Processor Extended State Enumeration (ECX=1)

The value returned in EDX for subfunction 1 is undefined and reserved.

Subfunction 2 of Fn0000_000D

Subfunction 2 provides information about the size and offset of the 256-bit SSE vector floating point processor unit state save area.

CPUID Fn0000_000D_EAX_x2 Processor Extended State Enumeration (ECX=2)

The value returned in EAX provides information about the size of the 256-bit SSE vector floating point processor unit state save area.

Bits	Field Name	Description
31:0	YmmSaveStateSize	YMM state save size. The state save area size in bytes for The YMM registers.

CPUID Fn0000_000D_EBX_x2 Processor Extended State Enumeration (ECX=2)

The value returned in EBX provides information about the offset of the 256-bit SSE vector floating point processor unit state save area from the base of the extended state (XSAVE/XRSTOR) save area.

Bits	Field Name	Description
31:0	YmmSaveStateOffset	YMM state save offset. The offset in bytes from the base of the extended state save area of the YMM register state save area.

CPUID Fn0000_000D_E[D,C]X_x2 Processor Extended State Enumeration (ECX=2)

The values returned in ECX and EDX for subfunction 2 are undefined and are reserved.

Subfunction 11 of Fn0000_000D

Subfunction 11 provides information about the CET user state save area.

CPUID Fn0000_000D_E[A, B, C, D]X_x11 Processor Extended State Emulation (ECX=11)

The value returned in EAX, EBX, ECX and EDX provides information about the CET user state save area.

Register	Bits	Field Name	Description
EAX	31:0	CetUserSize	CET user state save size in bytes
EBX	31:0	CetUserOffset	CET user state offset from the base of the extended state save area
ECX	0	U/S	Set to '1', indicating a supervisor state component
ECX	31:0	—	Cleared to 0
EDX	31:0	—	Unused, cleared to 0

Subfunction 12 of Fn0000_000D

Subfunction 12 provides information about the CET supervisor state save area.

CPUID Fn0000_000D_E[A, B, C, D]X_x12 Processor Extended State Emulation (ECX=12)

The value returned in EAX, EBX, ECX and EDX provides information about the CET supervisor state save area.

Register	Bits	Field Name	Description
EAX	31:0	CetSupervisorSize	CET supervisor state save size in bytes
EBX	31:0	CetSupervisorOffset	CET supervisor state offset from the base of the extended state save area
ECX	0	U/S	Set to '1', indicating a supervisor state component
ECX	31:0	—	Cleared to 0
EDX	31:0	—	Unused, cleared to 0

Subfunction 3Eh of Fn0000_000D

Subfunction 3Eh provides information about the size and offset of the Lightweight Profiling (LWP) unit state save area.

CPUID Fn0000_000D_EAX_x3E Processor Extended State Enumeration (ECX=62)

The value returned in EAX provides the size of the Lightweight Profiling (LWP) unit state save area.

Bits	Field Name	Description
31:0	LwpSaveStateSize	LWP state save area size. The size of the save area for LWP state in bytes. See “Lightweight Profiling” in APM2.

CPUID Fn0000_000D_EBX_x3E Processor Extended State Enumeration (ECX=62)

The value returned in EBX provides the offset of the Lightweight Profiling (LWP) unit state save area from the base of the extended state (XSAVE/XRSTOR) save area.

Bits	Field Name	Description
31:0	LwpSaveStateOffset	LWP state save byte offset. The offset in bytes from the base of the extended state save area of the state save area for LWP. See “Lightweight Profiling” in APM2.

CPUID Fn0000_000D_E[D,C]X_x3E Processor Extended State Enumeration (ECX=62)

The values returned in ECX and EDX for subfunction 3Eh are undefined and are reserved.

Subfunctions of Fn0000_000D greater than 3Eh

For CPUID Fn0000_000D, if the subfunction (specified by contents of ECX) passed as input to the instruction is greater than 3Eh, the instruction returns zero in the EAX, EBX, ECX, and EDX registers.

E.3.11 Functions 4000_0000h–4000_FFh—Reserved for Hypervisor Use

CPUID Fn4000_00[FF:00] Reserved

These function numbers are reserved for use by the virtual machine monitor.

E.4 Extended Feature Function Numbers

This section describes each of the defined CPUID functions in the extended range.

E.4.1 Function 8000_0000h—Maximum Extended Function Number and Vendor String

This function provides information about the maximum extended function number supported on this processor and a string that identifies the vendor of the product.

CPUID Fn8000_0000_EAX Largest Extended Function Number

The value returned in EAX provides the largest extended function number supported by the processor.

Bits	Field Name	Description
31:0	LFuncExt	Largest extended function. The largest CPUID extended function input value supported by the processor implementation.

CPUID Fn8000_0000_E[D,C,B]X Processor Vendor

The values returned in EBX, ECX, and EDX together provide a 12-character string identifying the vendor of this processor. The output string is the same as the one returned by Fn0000_0000. See CPUID Fn0000_0000_E[D,C,B]X on page 594 for more details.

Bits	Field Name	Description
31:0	Vendor	Four characters of the 12-byte character string (encoded in ASCII) “AuthenticAMD”. See Table E-2 below.

Table E-2. CPUID Fn8000_0000_E[D,C,B]X values

Register	Value	Description
CPUID Fn8000_0000_EBX	6874_7541h	The ASCII characters “h t u A”.
CPUID Fn8000_0000_ECX	444D_4163h	The ASCII characters “D M A c”.
CPUID Fn8000_0000_EDX	6974_6E65h	The ASCII characters “i t n e”.

E.4.2 Function 8000_0001h—Extended Processor and Processor Feature Identifiers

CPUID Fn8000_0001_EAX AMD Family, Model, Stepping

The value returned in EAX provides the family, model, and stepping identifiers. Three values are used by software to identify a processor: Family, Model, and Stepping. The value returned in EAX is the same as the value returned in EAX for Fn0000_0001. See CPUID Fn0000_0001_EAX on page 595 for more details on the field definitions.

Bits	Field Names	Description
31:0	Family, Model, Stepping	See: CPUID Fn0000_0001_EAX.

CPUID Fn8000_0001_EBX BrandId Identifier

The value returned in EBX provides package type and a 16-bit processor name string identifiers.

Bits	Field Name	Description
31:28	PkgType	Package type. If (Family[7:0] >= 10h), this field is valid. If (Family[7:0]<10h), this field is reserved.
27:16	—	Reserved.
15:0	BrandId	Brand ID. This field, in conjunction with CPUID Fn0000_0001_EBX[8BitBrandId], is used by system firmware to generate the processor name string. See your processor revision guide for how to program the processor name string.

For processor families 10h and greater, PkgType is described in the *BIOS and Kernel Developer’s Guide* for the product.

CPUID Fn8000_0001_ECX Feature Identifiers

This function contains the following miscellaneous feature identifiers:

Bits	Field Name	Description
31	—	Reserved.
30	AddrMaskExt	Breakpoint Addressing masking extended to bit 31.
29	MONITORX	Support for MWAITX and MONITORX instructions.
28	PerfCtrExtLLC	Support for L3 performance counter extension.
27	PerfTsc	Performance time-stamp counter. Indicates support for MSRC001_0280 [Performance Time Stamp Counter].
26	DataBkptExt	Data access breakpoint extension. Indicates support for MSRC001_1027 and MSRC001_101[B:9].
25	—	Reserved
24	PerfCtrExtNB	NB performance counter extensions support. Indicates support for MSRC001_024[6,4,2,0] and MSRC001_024[7,5,3,1].
23	PerfCtrExtCore	Processor performance counter extensions support. Indicates support for MSRC001_020[A,8,6,4,2,0] and MSRC001_020[B,9,7,5,3,1].
22	TopologyExtensions	Topology extensions support. Indicates support for CPUID Fn8000_001D_EAX_x[N:0]-CPUID Fn8000_001E_EDX.
21	TBM	Trailing bit manipulation instruction support.
20	—	Reserved.
19	—	Reserved.
18	—	Reserved.
17	TCE	Translation Cache Extension support.
16	FMA4	Four-operand FMA instruction support.
15	LWP	Lightweight profiling support. See “Lightweight Profiling” in APM2 and reference pages for individual LWP instructions in APM3.
14	—	Reserved.
13	WDT	Watchdog timer support. See APM2 and APM3. Indicates support for MSRC001_0074.
12	SKINIT	SKINIT and STGI are supported. Indicates support for SKINIT and STGI, independent of the value of MSRC000_0080[SVME]. See APM2 and APM3.
11	XOP	Extended operation support.
10	IBS	Instruction based sampling. See “Instruction Based Sampling” in APM2.
9	OSVW	OS visible workaround. Indicates OS-visible workaround support. See “OS Visible Work-around (OSVW) Information” in APM2.
8	3DNowPrefetch	PREFETCH and PREFETCHW instruction support. See “PREFETCH” and “PREFETCHW” in APM3.
7	MisAlignSse	Misaligned SSE mode. See “Misaligned Access Support Added for SSE Instructions” in APM1.
6	SSE4A	EXTRQ, INSERTQ, MOVNTSS, and MOVNTSD instruction support. See “EXTRQ”, “INSERTQ”, “MOVNTSS”, and “MOVNTSD” in APM4.
5	ABM	Advanced bit manipulation. LZCNT instruction support. See “LZCNT” in APM3.
4	AltMovCr8	LOCK MOV CR0 means MOV CR8. See “MOV(CRn)” in APM3.

Bits	Field Name	Description
3	ExtApicSpace	Extended APIC space. This bit indicates the presence of extended APIC register space starting at offset 400h from the “APIC Base Address Register,” as specified in the BKDG.
2	SVM	Secure virtual machine. See “Secure Virtual Machine” in APM2.
1	CmpLegacy	Core multi-processing legacy mode. See “Legacy Method” on page 633.
0	LahfSahf	LAHF and SAHF instruction support in 64-bit mode. See “LAHF” and “SAHF” in APM3.

CPUID Fn8000_0001_EDX Feature Identifiers

This function contains the following miscellaneous feature identifiers:

Bits	Field Name	Description
31	3DNow	3DNow!™ instructions. See Appendix D “Instruction Subsets and CPUID Feature Sets” in APM3.
30	3DNowExt	AMD extensions to 3DNow! instructions. See Appendix D “Instruction Subsets and CPUID Feature Sets” in APM3.
29	LM	Long mode. See “Processor Initialization and Long-Mode Activation” in APM2.
28	—	Reserved.
27	RDTSCP	RDTSCP instruction. See “RDTSCP” in APM3.
26	Page1GB	1-GB large page support. See “1-GB Paging Support” in APM2.
25	FFXSR	FXSAVE and FXRSTOR instruction optimizations. See “FXSAVE” and “FXRSTOR” in APM5.
24	FXSR	FXSAVE and FXRSTOR instructions. Same as CPUID Fn0000_0001_EDX[FXSR].
23	MMX	MMX™ instructions. Same as CPUID Fn0000_0001_EDX[MMX].
22	MmxExt	AMD extensions to MMX instructions. See Appendix D “Instruction Subsets and CPUID Feature Sets” in APM3 and “128-Bit Media and Scientific Programming” in APM1.
21	—	Reserved.
20	NX	No-execute page protection. See “Page Translation and Protection” in APM2.
19:18	—	Reserved.
17	PSE36	Page-size extensions. Same as CPUID Fn0000_0001_EDX[PSE36].
16	PAT	Page attribute table. Same as CPUID Fn0000_0001_EDX[PAT].
15	CMOV	Conditional move instructions. Same as CPUID Fn0000_0001_EDX[CMOV].
14	MCA	Machine check architecture. Same as CPUID Fn0000_0001_EDX[MCA].
13	PGE	Page global extension. Same as CPUID Fn0000_0001_EDX[PGE].
12	MTRR	Memory-type range registers. Same as CPUID Fn0000_0001_EDX[MTRR].
11	SysCallSysRet	SYSCALL and SYSRET instructions. See “SYSCALL” and “SYSRET” in APM3.
10	—	Reserved.

Bits	Field Name	Description
9	APIC	Advanced programmable interrupt controller. Same as CPUID Fn0000_0001_EDX[APIC].
8	CMPXCHG8B	CMPXCHG8B instruction. Same as CPUID Fn0000_0001_EDX[CMPXCHG8B].
7	MCE	Machine check exception. Same as CPUID Fn0000_0001_EDX[MCE].
6	PAE	Physical-address extensions. Same as CPUID Fn0000_0001_EDX[PAE].
5	MSR	AMD model-specific registers. Same as CPUID Fn0000_0001_EDX[MSR].
4	TSC	Time stamp counter. Same as CPUID Fn0000_0001_EDX[TSC].
3	PSE	Page-size extensions. Same as CPUID Fn0000_0001_EDX[PSE].
2	DE	Debugging extensions. Same as CPUID Fn0000_0001_EDX[DE].
1	VME	Virtual-mode enhancements. Same as CPUID Fn0000_0001_EDX[VME].
0	FPU	x87 floating-point unit on-chip. Same as CPUID Fn0000_0001_EDX[FPU].

E.4.3 Functions 8000_0002h–8000_0004h—Extended Processor Name String

CPUID Fn8000_000[4:2]_E[D,C,B,A]X Processor Name String Identifier

The three extended functions from Fn8000_0002 to Fn8000_0004 are programmed to return a null terminated ASCII string up to 48 characters in length corresponding to the processor name.

Bits	Field Name	Description
31:0	ProcName	Four characters of the extended processor name string.

The 48 character maximum includes the terminating null character. The 48 character string is ordered first to last (left to right) as follows:

Fn8000_0002[EAX[7:0],..., EAX[31:24], EBX[7:0],..., EBX[31:24], ECX[7:0],..., ECX[31:24],EDX[7:0],..., EDX[31:24]],
Fn8000_0003[EAX[7:0],..., EAX[31:24], EBX[7:0],..., EBX[31:24], ECX[7:0],..., ECX[31:24], EDX[7:0],..., EDX[31:24]],
Fn8000_0004[EAX[7:0],..., EAX[31:24], EBX[7:0],..., EBX[31:24], ECX[7:0],..., ECX[31:24], EDX[7:0],..., EDX[31:24]].

The extended processor name string is programmed by system firmware. See your processor revision guide for information about how to display the extended processor name string.

E.4.4 Function 8000_0005h—L1 Cache and TLB Information

This function provides first level cache TLB characteristics for the processor that executes the instruction.

CPUID Fn8000_0005_EAX L1 TLB 2M/4M Information

The value returned in EAX provides information about the L1 TLB for 2-MB and 4-MB pages.

Bits	Field Name	Description
31:24	L1DTlb2and4MAssoc	Data TLB associativity for 2-MB and 4-MB pages. Encoding is per Table E-3 below.
23:16	L1DTlb2and4MSize	Data TLB number of entries for 2-MB and 4-MB pages. The value returned is for the number of entries available for the 2-MB page size; 4-MB pages require two 2-MB entries, so the number of entries available for the 4-MB page size is one-half the returned value.
15:8	L1ITlb2and4MAssoc	Instruction TLB associativity for 2-MB and 4-MB pages. Encoding is per Table E-3 below.
7:0	L1ITlb2and4MSize	Instruction TLB number of entries for 2-MB and 4-MB pages. The value returned is for the number of entries available for the 2-MB page size; 4-MB pages require two 2-MB entries, so the number of entries available for the 4-MB page size is one-half the returned value.

The associativity fields (L1DTlb2and4MAssoc and L1ITlb2and4MAssoc) are encoded as follows:

Table E-3. L1 Cache and TLB Associativity Field Encodings

Associativity [7:0]	Definition
00h	Reserved
01h	1 way (direct mapped)
02h–FEh	*n*-way associative. (field encodes *n*)
FFh	Fully associative

CPUID Fn8000_0005_EBX L1 TLB 4K Information

The value returned in EBX provides information about the L1 TLB for 4-KB pages.

Bits	Field Name	Description
31:24	L1DTlb4KAssoc	Data TLB associativity for 4 KB pages. Encoding is per Table E-3 above.
23:16	L1DTlb4KSize	Data TLB number of entries for 4 KB pages.
15:8	L1ITlb4KAssoc	Instruction TLB associativity for 4 KB pages. Encoding is per Table E-3 above.
7:0	L1ITlb4KSize	Instruction TLB number of entries for 4 KB pages.

The associativity fields (L1DTlb4KAssoc and L1ITlb4KAssoc) are encoded as specified in Table E-3 on page 613.

CPUID Fn8000_0005_ECX L1 Data Cache Information

The value returned in ECX provides information about the first level data cache.

Bits	Field Name	Description
31:24	L1DcSize	L1 data cache size in KB.
23:16	L1DcAssoc	L1 data cache associativity. Encoding is per Table E-3.
15:8	L1DcLinesPerTag	L1 data cache lines per tag.
7:0	L1DcLineSize	L1 data cache line size in bytes.

The associativity field (L1DcAssoc) is encoded as specified in Table E-3 on page 613.

CPUID Fn8000_0005_EDX L1 Instruction Cache Information

The value returned in EDX provides information about the first level instruction cache.

Bits	Field Name	Description
31:24	L1IcSize	L1 instruction cache size KB.
23:16	L1IcAssoc	L1 instruction cache associativity. Encoding is per Table E-3.
15:8	L1IcLinesPerTag	L1 instruction cache lines per tag.
7:0	L1IcLineSize	L1 instruction cache line size in bytes.

The associativity field (L1IcAssoc) is encoded as specified in Table E-3 on page 613.

E.4.5 Function 8000_0006h—L2 Cache and TLB and L3 Cache Information

This function provides the second level cache and TLB characteristics for the logical processor that executes the instruction. The EDX register returns the processor's third level cache characteristics that are shared by all logical processors in the package.

CPUID Fn8000_0006_EAX L2 TLB 2M/4M Information

The value returned in EAX provides information about the L2 TLB for 2-MB and 4-MB pages.

Bits	Field Name	Description
31:28	L2DTlb2and4MAssoc	L2 data TLB associativity for 2-MB and 4-MB pages. Encoding is per Table E-4 below.
27:16	L2DTlb2and4MSize	L2 data TLB number of entries for 2-MB and 4-MB pages. The value returned is for the number of entries available for the 2 MB page size; 4 MB pages require two 2 MB entries, so the number of entries available for the 4 MB page size is one-half the returned value.

15:12	L2ITlb2and4MAssoc	L2 instruction TLB associativity for 2-MB and 4-MB pages. Encoding is per Table E-4 below.
11:0	L2ITlb2and4MSize	L2 instruction TLB number of entries for 2-MB and 4-MB pages. The value returned is for the number of entries available for the 2 MB page size; 4 MB pages require two 2 MB entries, so the number of entries available for the 4 MB page size is one-half the returned value.

The associativity fields (L2DTlb2and4MAssoc and L2ITlb2and4MAssoc) are encoded as follows:

Table E-4. L2/L3 Cache and TLB Associativity Field Encoding

Associativity [3:0]	Definition
0h	L2/L3 cache or TLB is disabled.
1h	Direct mapped.
2h	2-way associative.
3h	3-way associative.
4h	4-way associative.
5h	6-way associative.
6h	8-way associative.
8h	16-way associative.
9h	Value for all fields should be determined from Fn8000_001D
Ah	32-way associative.
Bh	48-way associative.
Ch	64-way associative.
Dh	96-way associative.
Eh	128-way associative.
Fh	Fully associative.
All other encodings are reserved.	

CPUID Fn8000_0006_EBX L2 TLB 4K Information

The value returned in EBX provides information about the L2 TLB for 4-KB pages.

Bits	Field Name	Description
31:28	L2DTlb4KAssoc	L2 data TLB associativity for 4-KB pages. Encoding is per Table E-4 above.
27:16	L2DTlb4KSize	L2 data TLB number of entries for 4-KB pages.
15:12	L2ITlb4KAssoc	L2 instruction TLB associativity for 4-KB pages. Encoding is per Table E-4 above.
11:0	L2ITlb4KSize	L2 instruction TLB number of entries for 4-KB pages.

The associativity fields (L2DTlb4KAssoc and L2ITlb4KAssoc) are encoded per Table E-4 above.

CPUID Fn8000_0006_ECX L2 Cache Information

The value returned in ECX provides information about the L2 cache.

Bits	Field Name	Description
31:16	L2Size	L2 cache size in KB.
15:12	L2Assoc	L2 cache associativity. Encoding is per Table E-4 on page 615.
11:8	L2LinesPerTag	L2 cache lines per tag.
7:0	L2LineSize	L2 cache line size in bytes.

The associativity field (L2Assoc) is encoded per Table E-4 on page 615.

CPUID Fn8000_0006_EDX L3 Cache Information

The value returned in EDX provides the third level cache characteristics shared by all logical processors in the package.

Bits	Field Name	Description
31:18	L3Size	Specifies the L3 cache size range: (L3Size[31:18] * 512KB) ≤ L3 cache size < ((L3Size[31:18]+1) * 512KB).
17:16	—	Reserved.
15:12	L3Assoc	L3 cache associativity. Encoded per Table E-4 on page 615.
11:8	L3LinesPerTag	L3 cache lines per tag.
7:0	L3LineSize	L3 cache line size in bytes.

The associativity field (L3Assoc) is encoded per Table E-4 on page 615.

E.4.6 Function 8000_0007h—Processor Power Management and RAS Capabilities

This function provides information about the power management, power reporting, and RAS capabilities of the processor that executes the instruction.There may be other processor-specific features and reporting capabilities not covered here. Refer to the *BIOS and Kernel Developer's Guide* for your specific product to otain more information.

CPUID Fn8000_0007_EAX Reserved

Bits	Field Name	Description
31:0	—	Reserved.

CPUID Fn8000_0007_EBX RAS Capabilities

The value returned in EBX provides information about RAS features that allow system software to detect specific hardware errors.

Bits	Field Name	Description
31:4	—	Reserved.
3	ScalableMca	0=MCAX is not supported. 1=MCAX is supported; the MCAX MSR addresses are supported; MCA_CONFIG[Mcax] is present in all MCA banks.
2	HWA	Hardware assert supported. Indicates support for MSRC001_10[DF:C0].
1	SUCCOR	Software uncorrectable error containment and recovery capability. The processor supports software containment of uncorrectable errors through context synchronizing data poisoning and deferred error interrupts; see APM2, Chapter 9, “Determining Machine-Check Architecture Support.”
0	McaOverflowRecov	MCA overflow recovery support. If set, indicates that MCA overflow conditions (MCi_STATUS[Overflow]=1) are not fatal; software may safely ignore such conditions. If clear, MCA overflow conditions require software to shut down the system. See APM2, Chapter 9, “Handling Machine Check Exceptions.”

CPUID Fn8000_0007_ECX Processor Power Monitoring Interface

The value returned in ECX provides information about the implementation of the processor power monitoring interface.

Bits	Field Name	Description
31:0	CpuPwrSampleTimeRatio	Specifies the ratio of the compute unit power accumulator sample period to the TSC counter period. Returns a value of 0 if not applicable for the system.

CPUID Fn8000_0007_EDX Advanced Power Management Features

The value returned in EDX provides information about the advanced power management and power reporting features available. Refer to the *BIOS and Kernel Developer’s Guide* for your specific product for a detailed description of the definition of each power management feature.

Bits	Field Name	Description
31:13	—	Reserved.
12	ProcPowerReporting	Processor power reporting interface supported.
11	ProcFeedbackInterface	Processor feedback interface. Value: 1. 1=Indicates support for processor feedback interface. **Note:** This feature is deprecated.
10	EffFreqRO	Read-only effective frequency interface. 1=Indicates presence of MSRC000_00E7 [Read-Only Max Performance Frequency Clock Count (MPerfReadOnly)] and MSRC000_00E8 [Read-Only Actual Performance Frequency Clock Count (APerfReadOnly)].
9	CPB	Core performance boost.

8	TscInvariant	TSC invariant. The TSC rate is ensured to be invariant across all P-States, C-States, and stop grant transitions (such as STPCLK Throttling); therefore the TSC is suitable for use as a source of time. 0 = No such guarantee is made and software should avoid attempting to use the TSC as a source of time.
7	HwPstate	Hardware P-state control. MSRC001_0061 [P-state Current Limit], MSRC001_0062 [P-state Control] and MSRC001_0063 [P-state Status] exist.
6	100MHzSteps	100 MHz multiplier Control.
5	—	Reserved.
4	TM	Hardware thermal control (HTC).
3	TTP	THERMTRIP.
2	VID	Voltage ID control. Function replaced by HwPstate.
1	FID	Frequency ID control. Function replaced by HwPstate.
0	TS	Temperature sensor.

E.4.7 Function 8000_0008h—Processor Capacity Parameters and Extended Feature Identification

This function provides the size or capacity of various architectural parameters that vary by implementation, as well as an extension to the Fn8000_0001 feature identifiers.

CPUID Fn8000_0008_EAX Long Mode Size Identifiers

The value returned in EAX provides information about the maximum host and guest physical and linear address width (in bits) supported by the processor.

Bits	Field Name	Description
31:24	—	Reserved.
23:16	GuestPhysAddrSize	Maximum guest physical address size in bits. This number applies only to guests using nested paging. When this field is zero, refer to the PhysAddrSize field for the maximum guest physical address size. See “Secure Virtual Machine” in APM2.
15:8	LinAddrSize	Maximum linear address size in bits.
7:0	PhysAddrSize	Maximum physical address size in bits. When GuestPhysAddrSize is zero, this field also indicates the maximum guest physical address size.

The address width reported is the maximum supported in any mode. For long mode capable processors, the size reported is independent of whether long mode is enabled. See “Processor Initialization and Long-Mode Activation” in APM2.

CPUID Fn8000_0008_EBX Extended Feature Identifiers

The value returned in EBX is an extension to the Fn8000_0001 feature flags and indicates the presence of various ISA extensions.

Bit	Field Name	Description
31:29	—	Reserved
28	PSFD	Predictive Store Forward Disable
27	—	Reserved
26	SsbdNotRequired	SSBD not needed on this processor
25	SsbdVirtSpecCtrl	Use VIRT_SPEC_CTL for SSBD
24	SSBD	Speculative Store Bypass Disable
23:22	—	Reserved
21	INVLPGBnestedPages	INVLPGB support for invalidating guest nested translations
20	EferLmsleUnsupported	EFER.LMSLE is unsupported.
19	IbrsSameMode	IBRS provides same mode speculation limits
18	IbrsPreferred	IBRS is preferred over software solution
17	StibpAlwaysOn	Processor prefers that STIBP be left on
16	IbrsAlwaysOn	Processor prefers that IBRS be left on
15	STIBP	Single Thread Indirect Branch Prediction mode
14	IBRS	Indirect Branch Restricted Speculation
13	INT_WBINVD	WBINVD/WBNOINVD are interruptible.
12	IBPB	Indirect Branch Prediction Barrier
11:10	—	Reserved
9	WBNOINVD	WBNOINVD instruction supported
8	MCOMMIT	MCOMMIT instruction supported
7:5	—	Reserved
4	RDPRU	RDPRU instruction supported
3	INVLPGB	INVLPGB and TLBSYNC instruction supported
2	RstrFpErrPtrs	FP Error Pointers Restored by XRSTOR
1	InstRetCntMsr	Instruction Retired Counter MSR available
0	CLZERO	CLZERO instruction supported

CPUID Fn8000_0008_ECX Size Identifiers

The value returned in ECX provides information about the number of cores supported by the processor, the width of the APIC ID, and the width of the performance time-stamp counter.

Bits	Field Name	Description
31:18	—	Reserved.
17:16	PerfTscSize	Performance time-stamp counter size. Indicates the size of MSRC001_0280[PTSC]. Bits Description 00b 40 bits 01b 48 bits 10b 56 bits 11b 64 bits

15:12	ApicIdSize	APIC ID size. The number of bits in the initial APIC20[ApicId] value that indicate logical processor ID within a package. The size of this field determines the maximum number of logical processors (MNLP) that the package could theoretically support, and not the actual number of logical processors that are implemented or enabled in the package, as indicated by CPUID Fn8000_0008_ECX[NC]. A value of zero indicates that legacy methods must be used to determine the maximum number of logical processors, as indicated by CPUID Fn8000_0008_ECX[NC].
		
		if (ApicIdSize[3:0] == 0) {
		// Used by legacy dual-core/single-core processors
		MNLP = CPUID Fn8000_0008_ECX[NC] + 1;
		} else {
		// use ApicIdSize[3:0] field
		MNLP = (2 raised to the power of ApicIdSize[3:0]);
		}

11:8	—	Reserved.
7:0	NT	Number of physical threads - 1. The number of threads in the processor is NT+1 (e.g., if NT = 0, then there is one thread). See “Legacy Method” on page 633.

CPUID Fn8000_0008_EDX RDPRU Register Identifier Range

The value returned in EDX identifies the maximum recognized register identifier for the RDPRU instruction.

Bits	Field Name	Description
63:32	—	Reserved.
31:16	MaxRdpruID	The maximum ECX value recognized by RDPRU.
15:0	InvlpgbCountMax	Maximum page count for INVLPGB instruction.

E.4.8 Function 8000_0009h—Reserved

CPUID Fn8000_0009 Reserved

This function is reserved.

E.4.9 Function 8000_000Ah—SVM Features

This function provides information about the SVM features that the processory supports. If SVM is not supported (CPUID Fn8000_0001_ECX[SVM] = 0), this function is reserved.

CPUID Fn8000_000A_EAX SVM Revision and Feature Identification

The value returned in EAX provides the SVM revision number. I

Bits	Field Name	Description
31:8	—	Reserved.
7:0	SvmRev	SVM revision number.

CPUID Fn8000_000A_EBX SVM Revision and Feature Identification

The value returned in EBX provides the number of address space identifiers (ASIDs) that the processor supports.

Bits	Field Name	Description
31:0	NASID	Number of available address space identifiers (ASID).

CPUID Fn8000_000A_ECX Reserved

The value returned in ECX for this function is undefined and is reserved.

CPUID Fn8000_000A_EDX SVM Feature Identification

The value returned in EDX provides Secure Virtual Machine architecture feature information. All cross references in the table below are to sections within the *Secure Virtual Machine* chapter of APM2.

Bits	Field Name	Description
31:25	—	Reserved.
24	TlbiCtl	Support for INVLPGB/TLBSYNC hypervisor enable in VMCB and TLBSYNC intercept.
23	HOST_MCE_OVERRIDE	When host CR4.MCE=1 and guest CR4.MCE=0, machine check exceptions (#MC) in a guest do not cause shutdown and are always intercepted.
22:21	—	Reserved.

Bits	Field Name	Description
20	SpecCtrl	SPEC_CTRL virtualization.
19	SSSCheck	SVM supervisor shadow stack restrictions. See “Supervisor Shadow Stack Restrictions” in Volume 2.
18	—	Reserved.
17	GMET	Guest Mode Execution Trap.
16	VGIF	Virtualize the Global Interrupt Flag. See "Nested Virtualization"
15	VMSAVEvirt	VMSAVE and VMLOAD virtualization. See "Nested Virtualization"
14	—	Reserved.
13	AVIC	Support for the AMD advanced virtual interrupt controller. See “Advanced Virtual Interrupt Controller.”
12	PauseFilterThreshold	PAUSE filter threshold. Indicates support for the PAUSE filter cycle count threshold. See "Pause Intercept Filtering” in Volume 2.
11	—	Reserved.
10	PauseFilter	Pause intercept filter. Indicates support for the pause intercept filter. See “Pause Intercept Filtering.”
9:8	—	Reserved.
7	DecodeAssists	Decode assists. Indicates support for the decode assists. See “Decode Assists.”
6	FlushByAsid	Flush by ASID. Indicates that TLB flush events, including CR3 writes and CR4.PGE toggles, flush only the current ASID's TLB entries. Also indicates support for the extended VMCB TLB_Control. See “TLB Control.”
5	VmcbClean	VMCB clean bits. Indicates support for VMCB clean bits. See “VMCB Clean Bits.”
4	TscRateMsr	MSR based TSC rate control. Indicates support for MSR TSC ratio MSRC000_0104. See “TSC Ratio MSR (C000_0104h).”
3	NRIPS	NRIP save. Indicates support for NRIP save on #VMEXIT. See “State Saved on Exit.”
2	SVML	SVM lock. Indicates support for SVM-Lock. See “Enabling SVM.”
1	LbrVirt	LBR virtualization. Indicates support for LBR Virtualization. See “Enabling LBR Virtualization.”
0	NP	Nested paging. Indicates support for nested paging. See “Nested Paging.”

E.4.10 Functions 8000_000Bh–8000_0018h—Reserved

CPUID Fn8000_00[18:0B] Reserved

These functions are reserved.

E.4.11 Function 8000_0019h—TLB Characteristics for 1GB pages

This function provides information about the TLB for 1 GB pages for the processor that executes the instruction.

CPUID Fn8000_0019_EAX L1 TLB 1G Information

The value returned in EAX provides information about the L1 TLB for 1 GB pages.

Bits	Field Name	Description
31:28	L1DTlb1GAssoc	L1 data TLB associativity for 1 GB pages. See Table E-4 on page 615.
27:16	L1DTlb1GSize	L1 data TLB number of entries for 1 GB pages.
15:12	L1ITlb1GAssoc	L1 instruction TLB associativity for 1 GB pages. See Table E-4 on page 615.
11:0	L1ITlb1GSize	L1 instruction TLB number of entries for 1 GB pages.

CPUID Fn8000_0019_EBX L2 TLB 1G Information

The value returned in EBX provides information about the L2 TLB for 1 GB pages.

Bits	Field Name	Description
31:28	L2DTlb1GAssoc	L2 data TLB associativity for 1 GB pages. See Table E-4 on page 615.
27:16	L2DTlb1GSize	L2 data TLB number of entries for 1 GB pages.
15:12	L2ITlb1GAssoc	L2 instruction TLB associativity for 1 GB pages. See Table E-4 on page 615.
11:0	L2ITlb1GSize	L2 instruction TLB number of entries for 1 GB pages.

CPUID Fn8000_0019_E[D,C]X Reserved

The values returned in ECX and EDX for this function are undefined and reserved for future use.

E.4.12 Function 8000_001Ah—Instruction Optimizations

CPUID Fn8000_001A_EAX Performance Optimization Identifiers

This function returns performance related information. For more details on how to use these bits to optimize software, see the *Software Optimization Guide* applicable to your product.

Bits	Field Name	Description
31:3	—	Reserved.
2	FP256	The internal FP/SIMD execution datapath is 256 bits wide.
1	MOVU	MOVU SSE nstructions are more efficient and should be preferred to SSE MOVL/MOVH. MOVUPS is more efficient than MOVLPS/MOVHPS. MOVUPD is more efficient than MOVLPD/MOVHPD.
0	FP128	The internal FP/SIMD execution datapath is 128 bits wide.

CPUID Fn8000_001A_E[D,C,B]X Reserved

The values returned in EBX, ECX, and EDX are undefined for this function and are reserved.

E.4.13 Function 8000_001Bh—Instruction-Based Sampling Capabilities

If instruction-based sampling (IBS) is supported (CPUID Fn8000_0001_ECX[IBS] = 1), this CPUID function can be used to obtain IBS feature information. If IBS is not supported (CPUID Fn8000_0001_ECX[IBS] = 0), this function number is reserved. For more information on using IBS, see “Instruction-Based Sampling” in APM2.

CPUID Fn8000_001B_EAX Instruction-Based Sampling Feature Indicators

The value returned in EAX provides the following information about the specific features of IBS that the processor supports:

Bits	Field Name	Description
31:9		Reserved.
8	OpBrnFuse	Fused branch micro-op indication supported.
7	RipInvalidChk	Invalid RIP indication supported.
6	OpCntExt	IbsOpCurCnt and IbsOpMaxCnt extend by 7 bits.
5	BrnTrgt	Branch target address reporting supported.
4	OpCnt	Op counting mode supported.
3	RdWrOpCnt	Read write of op counter supported.
2	OpSam	IBS execution sampling supported.
1	FetchSam	IBS fetch sampling supported.
0	IBSFFV	IBS feature flags valid.

CPUID Fn8000_001B_E[D,C,B]X Reserved

The values returned in EBX, ECX, and EDX are undefined and are reserved.

E.4.14 Function 8000_001Ch—Lightweight Profiling Capabilities

If lightweight profilling (LWP) is supported (CPUID Fn8000_0001_ECX[LWP] = 1), this CPUID function can be used to obtain information about LWP features supported by the processor. If LWP is not supported (CPUID Fn8000_0001_ECX[LWP] = 0), this function number is reserved. For more information on using LWP, see “Lightweight Profiling” in APM2.

CPUID Fn8000_001C_EAX Lightweight Profiling Capabilities 0

The value returned in EAX provides the following information about LWP capabilities supported by the processor:

Bits	Field Name	Description
31	LwpInt	Interrupt on threshold overflow available.
30	LwpPTSC	Performance time stamp counter in event record is available.
29	LwpCont	Sampling in continuous mode is available.
28:7	—	Reserved.
6	LwpRNH	Core reference clocks not halted event available.
5	LwpCNH	Core clocks not halted event available.
4	LwpDME	DC miss event available.
3	LwpBRE	Branch retired event available.
2	LwpIRE	Instructions retired event available.
1	LwpVAL	LWPVAL instruction available.
0	LwpAvail	The LWP feature is available.

CPUID Fn8000_001C_EBX Lightweight Profiling Capabilities 0

The value returned in EBX provides the following additional information about LWP capabilities supported by the processor:

Bits	Field Name	Description
31:24	LwpEventOffset	Offset in bytes from the start of the LWPCB to the EventInterval1 field.
23:16	LwpMaxEvents	Maximum EventId value supported.
15:8	LwpEventSize	Event record size. Size in bytes of an event record in the LWP event ring buffer.
7:0	LwpCbSize	Control block size. Size in quadwords of the LWPCB.

CPUID Fn8000_001C_ECX Lightweight Profiling Capabilities 0

The value returned in ECX provides the following additional information about LWP capabilities supported by the processor:

Bits	Field Name	Description
31	LwpCacheLatency	Cache latency filtering supported. Cache-related events can be filtered by latency.
30	LwpCacheLevels	Cache level filtering supported. Cache-related events can be filtered by the cache level that returned the data.
29	LwpIpFiltering	IP filtering supported.
28	LwpBranchPrediction	Branch prediction filtering supported. Branches Retired events can be filtered based on whether the branch was predicted properly.

Bits	Field Name	Description
27:24	—	Reserved.
23:16	LwpMinBufferSize	Event ring buffer size. Minimum size of the LWP event ring buffer, in units of 32 event records.
15:9	LwpVersion	Version of LWP implementation.
8:6	LwpLatencyRnd	Amount by which cache latency is rounded.
5	LwpDataAddress	Data cache miss address valid. Address is valid for cache miss event records.
4:0	LwpLatencyMax	Latency counter size. Size in bits of the cache latency counters.

CPUID Fn8000_001C_EDX Lightweight Profiling Capabilities 0

The value returned in EDX provides the following additional information about LWP capabilities supported by the processor:

Bits	Field Name	Description
31	LwpInt	Interrupt on threshold overflow supported.
30	LwpPTSC	Performance time stamp counter in event record is supported.
29	LwpCont	Sampling in continuous mode is supported.
28:7	—	Reserved.
6	LwpRNH	Core reference clocks not halted event is supported.
5	LwpCNH	Core clocks not halted event is supported.
4	LwpDME	DC miss event is supported.
3	LwpBRE	Branch retired event is supported.
2	LwpIRE	Instructions retired event is supported.
1	LwpVAL	LWPVAL instruction is supported.
0	LwpAvail	Lightweight profiling is supported.

E.4.15 Function 8000_001Dh—Cache Topology Information

CPUID Fn8000_001D_E[D,C,B,A]X reports cache topology information for the cache enumerated by the value passed to the instruction in ECX, referred to as Cache *n* in the following description. To gather information for all cache levels, software must repeatedly execute CPUID with 8000_001Dh in EAX and ECX set to increasing values beginning with 0 until a value of 00h is returned in the field CacheType (EAX[4:0]) indicating no more cache descriptions are available for this processor.

If CPUID Fn8000_0001_ECX[TopologyExtensions] = 0, then CPUID Fn8000_001Dh is reserved. Any value in ECX which does not select an existing cache will return a Null cache type in EAX[4:0].

CPUID Fn8000_001D_EAX_x[N:0] Cache Properties

Bits	Field Name	Description
31:26	—	Reserved.
25:14	NumSharingCache	Specifies the number of logical processors sharing the cache enumerated by *N*, the value passed to the instruction in ECX. The number of logical processors sharing this cache is the value of this field incremented by 1. To determine which logical processors are sharing a cache, determine a Share Id for each processor as follows: ShareId = LocalApicId >> log2(NumSharingCache+1) Logical processors with the same ShareId then share a cache. If NumSharingCache+1 is not a power of two, round it up to the next power of two.
13:10	—	Reserved.
9	FullyAssociative	Fully associative cache. When set, indicates that the cache is fully associative. If 0 is returned in this field, the cache is set associative.
8	SelfInitialization	Self-initializing cache. When set, indicates that the cache is self initializing; software initialization not required. If 0 is returned in this field, hardware does not initialize this cache.
7:5	CacheLevel	Cache level. Identifies the level of this cache. Note that the enumeration value is not necessarily equal to the cache level. Bits — Description 000b — Reserved. 001b — Level 1 010b — Level 2 011b — Level 3 111b-100b — Reserved.
4:0	CacheType	Cache type. Identifies the type of cache. Bits — Description 00h — Null; no more caches. 01h — Data cache 02h — Instruction cache 03h — Unified cache 1Fh-04h — Reserved.

CPUID Fn8000_001D_EBX_x[N:0] Cache Properties

See CPUID Fn8000_001D_EAX_x[N:0].

Bits	Field Name	Description
31:22	CacheNumWays	Number of ways for this cache. The number of ways is the value returned in this field incremented by 1.
21:12	CachePhysPartitions	Number of physical line partitions. The number of physical line partitions is the value returned in this field incremented by 1.
11:0	CacheLineSize	Cache line size. The cache line size in bytes is the value returned in this field incremented by 1.

CPUID Fn8000_001D_ECX_x[N:0] Cache Properties

See CPUID Fn8000_001D_EAX_x[N:0].

Bits	Field Name	Description
31:0	CacheNumSets	Number of ways for set associative cache. Number of ways is the value returned in this field incremented by 1. Only valid for caches that are not fully associative (Fn8000_001D_EAX_x*n*[FullyAssociative] = 0).

CPUID Fn8000_001D_EDX_x[N:0] Cache Properties

See CPUID Fn8000_001D_EAX_x[N:0].

Bits	Field Name	Description
31:2	—	Reserved.
1	CacheInclusive	Cache inclusivity. A value of 0 indicates that this cache is not inclusive of lower cache levels. A value of 1 indicates that the cache is inclusive of lower cache levels.
0	WBINVD	Write-Back Invalidate/Invalidate execution scope. A value of 0 returned in this field indicates that the WBINVD/INVD instruction invalidates all lower level caches of non-originating logical processors sharing this cache. When set, this field indicates that the WBINVD/INVD instruction is not guaranteed to invalidate all lower level caches of non-originating logical processors sharing this cache.

E.4.16 Function 8000_001Eh—Processor Topology Information

CPUID Fn8000_001E_EAX Extended APIC ID

If CPUID Fn8000_0001_ECX[TopologyExtensions] = 0, this function number is reserved.

Bits	Field Name	Description
31:0	ExtendedApicId	Extended APIC ID. If MSR0000_001B[ApicEn] = 0, this field is reserved..

CPUID Fn8000_001E_EBX Compute Unit Identifiers

See CPUID Fn8000_001E_EAX.

Bits	Field Name	Description
31:16	—	Reserved.
15:8	ThreadsPerComputeUnit	Threads per compute unit (zero-based count). The actual number of threads per compute unit is the value of this field + 1. To determine which logical processors (threads) belong to a given Compute Unit, determine a ShareId for each processor as follows: ShareId = LocalApicId >> log2(ThreadsPerComputeUnit+1) Logical processors with the same ShareId then belong to the same Compute Unit. (If ThreadsPerComputeUnit+1 is not a power of two, round it up to the next power of two).
7:0	ComputeUnitId	Compute unit ID. Identifies a Compute Unit, which may be one or more physical cores that each implement one or more logical processors.

CPUID Fn8000_001E_ECX Node Identifiers

See CPUID Fn8000_001E_EAX.

Bits	Field Name	Description
31:0	—	Reserved.
10:8	NodesPerProcessor	Specifies the number of nodes in the package/socket in which this logical processor resides. Node in this context corresponds to a processor die. Encoding is N-1, where N is the number of nodes present in the socket.
7:0	NodeId	Specifies the ID of the node containing the current logical processor. NodeId values are unique across the system..

CPUID Fn8000_001E_EDX Reserved

The value returned in EDX is undefined and is reserved.

E.4.17 Function 8000_001Fh—Encrypted Memory Capabilities

CPUID Fn8000_001F_EAX Secure Encryption

Bits	Field Name	Description
31:25	—	Reserved.
24	VmsaRegProt	VMSA Register Protection supported
23:17	—	Reserved.
16	VTE	Virtual Transparent Encryption supported
15	PreventHostIbs	Disallowing IBS use by the host supported

Bits	Field Name	Description
14	DebugSwap	Full debug state swap supported for SEV-ES guests
13	AlternateInjection	Alternate Injection supported
12	RestrictedInjection	Restricted Injection supported
11	64BitHost	SEV guest execution only allowed from a 64-bit host
10	HwEnfCacheCoh	Hardware cache coherency across encryption domains enforced
9	—	Reserved.
8	SecureTsc	Secure TSC supported
7:6	—	Reserved.
5	VMPL	VM Permission Levels supported
4	SEV-SNP	SEV Secure Nested Paging supported
3	SEV-ES	SEV Encrypted State supported
2	PageFlushMsr	Page Flush MSR available
1	SEV	Secure Encrypted Virtualization supported
0	SME	Secure Memory Encryption supported

CPUID Fn8000_001F_EBX Secure Encryption

Bits	Field Name	Description
31:16	—	Reserved.
15:12	NumVMPL	Number of VM Permission Levels supported
11:6	PhysAddrReduction	Physical Address bit reduction
5:0	CbitPosition	C-bit location in page table entry

CPUID Fn8000_001F_ECX Secure Encryption

Bits	Field Name	Description
31:0	NumEncryptedGuests	Number of encrypted guests supported simultaneously

CPUID Fn8000_001F_EDX Minimum ASID

Bits	Field Name	Description
31:0	MinSevNoEsAsid	Minimum ASID value for an SEV enabled, SEV-ES disabled guest

E.4.18 Function 8000_0020—Reserved

E.4.19 Function 8000_0021—Extended Feature Identification 2

CPUID Fn8000_0021_EAX Extended Feature 2

Bits	Field Name	Description
31:14	—	Reserved
13	PrefetchCtlMsr	Prefetch control MSR supported. See Core::X86::Msr::PrefetchControl in BKDG or PPR for details
12:10	—	Reserved
9	NoSmmCtlMSR	SMM_CTL MSR (C001_0116h) is not supported.
8	—	Reserved
7	UpperAddressIgnore	Upper Address Ignore is supported.
6	NullSelectClearsBase	Null segment selector loads also clear the destination segment register base and limit.
5:4	—	Reserved
3	SmmPgCfgLock	SMM paging configuration lock supported.
2	LFenceAlwaysSerializing	LFENCE is always dispatch serializing.
1	—	Reserved
0	NoNestedDataBp	Processor ignores nested data breakpoints

CPUID Fn8000_0021_EBX Extended Feature 2

Bits	Field Name	Description
31:12	—	Reserved.
11:0	MicrocodePatchSize	The size of the Microcode patch in 16-byte multiples. If 0, the size of the patch is at most 5568 (15C0h) bytes.

CPUID Fn8000_0021_E[C,D]X Reserved

The values returned in ECX and EDX are undefined and are reserved.

CPUID Fn8000_0022_EAX Reserved

Bits	Field Name	Description
31:0	—	Reserved

CPUID Fn8000_0022_EBX Extended Performance Monitoring and Debug

Bits	Field Name	Description
31:16	—	Reserved
15:10	NumDfPmc	Number of available Northbridge Performance Monitor Counters
9:4	—	Reserved
3:0	NumCorePmc	Number of Core Performance Counters

CPUID Fn8000_0022_E[C,D]X Reserved

The values returned in ECX and EDX are undefined and are reserved.

E.5 Multiple Processor Calculation

Operating systems may use one of two possible methods to calculate the actual number of logical processors per package (NC), and the maximum possible number of logical processors per package (MNLP). The extended method is recommended, but a legacy method is also available.

E.5.1 Legacy Method

The CPUID identification of total number of logical processors per package is derived from information returned by the following fields:

- CPUID Fn0000_0001_EBX[LogicalProcessorCount]
- CPUID Fn0000_0001_EDX[HTT] (Hyper-Threading Technology)
- CPUID Fn8000_0001_ECX[CmpLegacy]
- CPUID Fn8000_0008_ECX[NC]

Table E-5 defines LogicalProcessorCount, HTT, CmpLegacy, and NC as a function of the number of logical processors per package (n).

When HTT = 0, LogicalProcessorCount is reserved and the package contains one logical processor.

When HTT = 1 and CmpLegacy = 1, LogicalProcessorCount represents the number of logical processors per package (n).

Table E-5. LogicalProcessorCount, CmpLegacy, HTT, and NC

Logical Processors per package	CmpLegacy	HTT	LogicalProcessorCount	NC
1	0	0	Reserved	0
2 or more	1	1	n	n-1

The use of CmpLegacy and LogicalProcessorCount for determining the number of logical processors is deprecated. Instead, use NC to determine the number of logical processors per package.

E.5.2 Extended Method (Recommended)

The CPUID identification of total number of logical processors per package is derived from information returned by the CPUID Fn8000_0008_ECX[ApicIdSize[3:0]]. This field indicates the number of least significant bits in the CPUID Fn0000_0001_EBX[LocalApicId] that indicates logical processor ID within the package. The size of this field determines the maximum number of logical processors (MNLP) that the package could theoretically support, and not the actual number of logical processors that are implemented or enabled in the package, as indicated by CPUID Fn8000_0008_ECX[NC].

A value of zero for ApicIdSize[3:0] indicates that the legacy method (section E5.1) should be used to derive the maximum number of logical processors:

MNLP = CPUID Fn8000_0008_ECX[NC] + 1.

And for non-zero values of ApicIdSize[3:0]:

MNLP = 2 raised to the power of ApicIdSize[3:0]

Appendix F Instruction Effects on RFLAGS

The flags in the RFLAGS register are described in “Flags Register” in Volume 1 and “RFLAGS Register” in Volume 2. Table F-1 summarizes the effect that instructions have on these flags. The table includes all instructions that affect the flags. Instructions not shown have no effect on RFLAGS.

The following codes are used within the table:

- 0—The flag is always cleared to 0.
- 1—The flag is always set to 1.
- AH—The flag is loaded with value from AH register.
- Mod—The flag is modified, depending on the results of the instruction.
- Pop—The flag is loaded with value popped off of the stack.
- Tst—The flag is tested.
- U—The effect on the flag is undefined.
- Gray shaded cells indicate that the flag is not affected by the instruction.

Table F-1. Instruction Effects on RFLAGS

Instruction Mnemonic	RFLAGS Mnemonic and Bit Number																
	ID 21	VIP 20	VIF 19	AC 18	VM 17	RF 16	NT 14	IOPL 13:12	OF 11	DF 10	IF 9	TF 8	SF 7	ZF 6	AF 4	PF 2	CF 0
AAA AAS									U				U	U	Tst Mod	U	Mod
AAD AAM									U				Mod	Mod	U	Mod	U
ADC									Mod				Mod	Mod	Mod	Mod	Tst Mod
ADD									Mod				Mod	Mod	Mod	Mod	Mod
AND									0				Mod	Mod	U	Mod	0
ARPL														Mod			
BSF BSR									U				U	Mod	U	U	U
BT BTC BTR BTS									U				U	U	U	U	Mod
BZHI									0				Mod	Mod	U	U	Mod
CLC																	0
CLD										0							
CLI			Mod					TST			Mod						
CMC																	Mod
CMOV*cc*									Tst				Tst	Tst		Tst	Tst
CMP									Mod				Mod	Mod	Mod	Mod	Mod

Table F-1. Instruction Effects on RFLAGS (continued)

Instruction Mnemonic	RFLAGS Mnemonic and Bit Number																
	ID 21	VIP 20	VIF 19	AC 18	VM 17	RF 16	NT 14	IOPL 13:12	OF 11	DF 10	IF 9	TF 8	SF 7	ZF 6	AF 4	PF 2	CF 0
CMPS*x*									Mod	Tst			Mod	Mod	Mod	Mod	Mod
CMPXCHG									Mod				Mod	Mod	Mod	Mod	Mod
CMPXCHG8B														Mod			
CMPXCHG16B														Mod			
COMISD COMISS									0				0	Mod	0	Mod	Mod
DAA DAS									U				Mod	Mod	Tst Mod	Mod	Tst Mod
DEC									Mod				Mod	Mod	Mod	Mod	
DIV									U				U	U	U	U	U
FCMOV*cc*														Tst		Tst	Tst
FCOMI FCOMIP FUCOMI FUCOMIP														Mod		Mod	Mod
IDIV									U				U	U	U	U	U
IMUL									Mod				U	U	U	U	Mod
INC									Mod				Mod	Mod	Mod	Mod	
IN								Tst									
INS*x*								Tst		Tst							
INT INT 3			Mod	Mod	Tst Mod	0	Mod	Tst			Mod	0					
INTO				Mod	Tst Mod	0	Mod	Tst	Tst		Mod	Mod					
IRET*x*	Pop	Pop	Pop	Pop	Tst Pop	Pop	Tst Pop	Tst Pop	Pop	Pop	Pop	Pop	Pop	Pop	Pop	Pop	Pop
Jcc									Tst				Tst	Tst		Tst	Tst
LAR														Mod			
LODS*x*										Tst							
LOOPE LOOPNE														Tst			
LSL														Mod			
LZCNT									U				U	Mod	U	U	Mod
MOVS*x*										Tst							
MUL									Mod				U	U	U	U	Mod
NEG									Mod				Mod	Mod	Mod	Mod	Mod
OR									0				Mod	Mod	U	Mod	0
OUT								Tst									
OUTS*x*								Tst		Tst							
PSMASH									Mod				Mod	Mod	Mod	Mod	
PVALIDATE									Mod				Mod	Mod	Mod	Mod	Mod
POPCNT									0				0	Mod	0	0	0

Table F-1. Instruction Effects on RFLAGS (continued)

Instruction Mnemonic	RFLAGS Mnemonic and Bit Number																
	ID 21	VIP 20	VIF 19	AC 18	VM 17	RF 16	NT 14	IOPL 13:12	OF 11	DF 10	IF 9	TF 8	SF 7	ZF 6	AF 4	PF 2	CF 0
POPF*x*	Pop	Tst	Mod	Pop	Tst	0	Pop	Tst Pop	Pop	Pop	Pop	Pop	Pop	Pop	Pop	Pop	Pop
RCL 1									Mod								Tst Mod
RCL *count*									U								Tst Mod
RCR 1									Mod								Tst Mod
RCR *count*									U								Tst Mod
RMPADJUST									Mod				Mod	Mod	Mod	Mod	
RMPUPDATE									Mod				Mod	Mod	Mod	Mod	
ROL 1									Mod								Mod
ROL *count*									U								Mod
ROR 1									Mod								Mod
ROR *count*									U								Mod
RSM	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod
SAHF													AH	AH	AH	AH	AH
SHL/SAL 1									Mod				Mod	Mod	U	Mod	Mod
SHL/SAL *count*									U				Mod	Mod	U	Mod	Mod
SAR 1									Mod				Mod	Mod	U	Mod	Mod
SAR *count*									U				Mod	Mod	U	Mod	Mod
SBB									Mod				Mod	Mod	Mod	Mod	Tst Mod
SCAS*x*									Mod	Tst			Mod	Mod	Mod	Mod	Mod
SETcc									Tst				Tst	Tst		Tst	Tst
SHLD 1 SHRD 1									Mod				Mod	Mod	U	Mod	Mod
SHLD count SHRD count									U				Mod	Mod	U	Mod	Mod
SHR 1									Mod				Mod	Mod	U	Mod	Mod
SHR count									U				Mod	Mod	U	Mod	Mod
STC																	1
STD										1							
STI			Mod					Tst			Mod						
STOS*x*										Tst							
SUB									Mod				Mod	Mod	Mod	Mod	Mod
SYSCALL	Mod	Mod	Mod	Mod	0	0	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod
SYSENTER					0	0					0						
SYSRET	Mod	Mod	Mod	Mod		0	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod	Mod
TEST									0				Mod	Mod	U	Mod	0

Table F-1. Instruction Effects on RFLAGS (continued)

Instruction Mnemonic	RFLAGS Mnemonic and Bit Number																
	ID 21	VIP 20	VIF 19	AC 18	VM 17	RF 16	NT 14	IOPL 13:12	OF 11	DF 10	IF 9	TF 8	SF 7	ZF 6	AF 4	PF 2	CF 0
UCOMISD UCOMISS									0				0	Mod	0	Mod	Mod
VERR VERW														Mod			
XADD									Mod				Mod	Mod	Mod	Mod	Mod
XOR									0				Mod	Mod	U	Mod	0

Index

S

T

V

X

Z

AMD64 Technology

AMD64 Architecture Programmer's Manual

Volume 4: 128-Bit and 256-Bit Media Instructions

Publication No.	Revision	Date
26568	3.25	November 2021

Advanced Micro Devices

Trademarks

Contents

Figures

Tables

Revision History

Date	Revision	Description
November 2021	3.25	Corrections to XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, XGETBV, and XSETBV descriptions.
May 2020	3.24	Chapter 2: Sections: Updated VAESDEC, VAESDECLAST, VAESENC, VAESENCLAST, VCMPPS, VPCLMULQDQ, and VPCMPGTQ.
January 2019	3.23	Updated the Exceptions table for MOVNTDAQA and VMOVNTDAQA. Corrections to VPMACSSDD and VPMACSSWW. Corrected scr1 to src1 throughout the document.
May 2018	3.22	Update Packed String Compare Algorithm Fixed a number of erroneous references to double precision that should be single precision Separate out MOVQ from MOVD
December 2017	3.21	Clarifications to XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV instructions.
March 2017	3.20	Corrections to ROUNDPD, VROUNDPD, ROUNDPS, VROUNDPS, ROUNDSD, VROUNDSD, ROUNDSS, VROUNDSS, VPERMD, VPERMPD, VPERMPS, VPERMQ, VTESTPD, VTESTPS, XGETBV, XSETBV, XSAVE, and AVX instruction descriptions. Added SHA1RNDS4, SHA1NEXTE, SHA1MSG1, SHA1MSG2, SHA256RNDS2, SHA256MSG1, SHA256MSG2, XRSTOR, XRSTORS and XSAVEC instructions.
June 2015	3.19	Corrections to the MOVLPD, PHSUBW, PHSUBSW instruction descriptions.
October 2013	3.18	Added AVX2 Instructions. Added “Instruction Support” subsection to each instruction reference page that lists CPUID feature bit information in a table.
May 2013	3.17	Removed all references to the CPUID specification which has been superseded by Volume 3, Appendix E, "Obtaining Processor Information Via the CPUID Instruction." Corrected exceptions table for the explicitly-aligned load/store instructions. General protection exception does not depend on state of MXCSR.MM bit.

Date	Revision	Description
September 2012	3.16	Corrected REX.W bit encoding for the MOVD instruction. (See page 186.) Corrected L bit encoding for the VMOVQ (D6h opcode) instruction. (See page 222.) Corrected statement about zero extension for third encoding (11h opcode) of MOVSS instruction. (See page 230.)
March 2012	3.15	Corrected instruction encoding for VPCOMUB, VPCOMUD, VPCOMUQ, VPCOMUW, and VPHSUBDQ instructions. Other minor corrections.
December 2011	3.14	Reworked Section 1.5, "String Compare Instructions" on page 10. Revised descriptions of the string compare instructions in instruction reference. Moved AES overview to Appendix A. Clarified trap and exception behavior for elements not selected for writing. See MASKMOVDQU VMASKMOVDQU on page 160. Additional minor corrections and clarifications.
September 2011	3.13	Moved discussion of extended instruction encoding; VEX and XOP prefixes to Volume 3. Added FMA instructions. Described on the corresponding FMA4 reference page. Moved BMI and TBM instructions to Volume 3. Added XSAVEOPT instruction. Corrected descriptions of VSQRTSD and VSQRTSS.
May 2011	3.12	Added F16C, BMI, and TBM instructions.
December 2010	3.11	Complete revision and reformat accommodating 128-bit and 256-bit media instructions. Includes revised definitions of legacy SSE, SSE2, SSE3, SSE4.1, SSE4.2, and SSSE3 instructions, as well as new definitions of extended AES, AVX, CLMUL, FMA4, and XOP instructions. Introduction includes supplemental information concerning encoding of extended instructions, enhanced processor state management provided by the XSAVE/XRSTOR instructions, cryptographic capabilities of the AES instructions, and functionality of extended string comparison instructions.
September 2007	3.10	Added minor clarifications and corrected typographical and formatting errors.

Date	Revision	Description
July 2007	3.09	Added the following instructions: EXTRQ, INSERTQ, MOVNTSD, and MOVNTSS. Added misaligned exception mask (MXCSR.MM) information. Added imm8 values with corresponding mnemonics to (V)CMPPD, (V)CMPPS, (V)CMPSD, and (V)CMPSS. Reworded CPUID information in condition tables. Added minor clarifications and corrected typographical and formatting errors.
September 2006	3.08	Made minor corrections.
December 2005	3.07	Made minor editorial and formatting changes.
January 2005	3.06	Added documentation on SSE3 instructions. Corrected numerous minor factual errors and typos.
September 2003	3.05	Made numerous small factual corrections.
April 2003	3.04	Made minor corrections.

Preface

About This Book

This book is part of a multivolume work entitled the *AMD64 Architecture Programmer's Manual*. The complete set includes the following volumes.

Title	Order No.
Volume 1: Application Programming	24592
Volume 2: System Programming	24593
Volume 3: General-Purpose and System Instructions	24594
Volume 4: 128-Bit and 256-Bit Media Instructions	26568
Volume 5: 64-Bit Media and x87 Floating-Point Instructions	26569

Audience

This volume is intended for programmers who develop application or system software.

Organization

Volumes 3, 4, and 5 describe the AMD64 instruction set in detail, providing mnemonic syntax, instruction encoding, functions, affected flags, and possible exceptions.

The AMD64 instruction set is divided into five subsets:

- General-purpose instructions
- System instructions
- Streaming SIMD Extensions (includes 128-bit and 256-bit media instructions)
- 64-bit media instructions (MMX™)
- x87 floating-point instructions

Several instructions belong to, and are described identically in, multiple instruction subsets.

This volume describes the Streaming SIMD Extensions (SSE) instruction set which includes 128-bit and 256-bit media instructions. SSE includes both legacy and extended forms. The index at the end cross-references topics within this volume. For other topics relating to the AMD64 architecture, and for information on instructions in other subsets, see the tables of contents and indexes of the other volumes.

Conventions and Definitions

The section which follows, **Notational Conventions,** describes notational conventions used in this volume. The next section, **Definitions,** lists a number of terms used in this volume along with their technical definitions. Some of these definitions assume knowledge of the legacy x86 architecture. See "Related Documents" on page xl for further information about the legacy x86 architecture. Finally, the **Registers** section lists the registers which are a part of the system programming model.

Notational Conventions

Section 1.1, "Syntax and Notation" on page 2 describes notation relating specifically to instruction encoding.

#GP(0)

An instruction exception—in this example, a general-protection exception with error code of 0.

1011b

A binary value, in this example, a 4-bit value.

F0EA_0B40h

A hexadecimal value, in this example a 32-bit value. Underscore characters may be used to improve readability.

128

Numbers without an alpha suffix are decimal unless the context indicates otherwise.

7:4

A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first. Commas may be inserted to indicate gaps.

#GP(0)

A general-protection exception (#GP) with error code of 0.

CPUID Fn*XXXX_XXXX_RRR*[*FieldName*]

Support for optional features or the value of an implementation-specific parameter of a processor can be discovered by executing the CPUID instruction on that processor. To obtain this value, software must execute the CPUID instruction with the function code *XXXX_XXXX*h in EAX and then examine the field *FieldName* returned in register *RRR*. If the "*_RRR*" notation is followed by "_x*YYY*", register ECX must be set to the value *YYY*h before executing CPUID. When *FieldName* is not given, the entire contents of register *RRR* contains the desired value. When determining optional feature support, if the bit identified by *FieldName* is set to a one, the feature is supported on that processor.

CR0–CR4

A register range, from register CR0 through CR4, inclusive, with the low-order register first.

CR4[OSXSAVE], CR4.OSXSAVE

The OSXSAVE bit of the CR4 register.

CR0[PE] = 1, CR0.PE = 1

The PE bit of the CR0 register has a value of 1.

EFER[LME] = 0, EFER.LME = 0

The LME field of the EFER register is cleared (contains a value of 0).

DS:rSI

The content of a memory location whose segment address is in the DS register and whose offset relative to that segment is in the rSI register.

RFLAGS[13:12]

A field within a register identified by its bit range. In this example, corresponding to the IOPL field.

Definitions

128-bit media instruction

Instructions that operate on the various 128-bit vector data types. Supported within both the legacy SSE and extended SSE instruction sets.

256-bit media instruction

Instructions that operate on the various 256-bit vector data types. Supported within the extended SSE instruction set.

64-bit media instructions

Instructions that operate on the 64-bit vector data types. These are primarily a combination of MMX and 3DNow!™ instruction sets and their extensions, with some additional instructions from the SSE1 and SSE2 instruction sets.

16-bit mode

Legacy mode or compatibility mode in which a 16-bit address size is active. See *legacy mode* and *compatibility mode*.

32-bit mode

Legacy mode or compatibility mode in which a 32-bit address size is active. See *legacy mode* and *compatibility mode*.

64-bit mode

A submode of *long mode*. In 64-bit mode, the default address size is 64 bits and new features, such as register extensions, are supported for system and application software.

absolute

A displacement that references the base of a code segment rather than an instruction pointer. See *relative*.

AES

Advance Encryption Standard (AES) algorithm acceleration instructions; part of Streaming SIMD Extensions (SSE).

ASID

Address space identifier.

AVX

Extension of the SSE instruction set supporting 256-bit vector (packed) operands. See Streaming SIMD Extensions.

biased exponent

The sum of a floating-point value's exponent and a constant bias for a particular floating-point data type. The bias makes the range of the biased exponent always positive, which allows reciprocation without overflow.

byte

Eight bits.

clear, cleared

To write the value 0 to a bit or a range of bits. See *set*.

compatibility mode

A submode of *long mode*. In compatibility mode, the default address size is 32 bits, and legacy 16-bit and 32-bit applications run without modification.

commit

To irreversibly write, in program order, an instruction's result to software-visible storage, such as a register (including flags), the data cache, an internal write buffer, or memory.

CPL

Current privilege level.

direct

Referencing a memory address included in the instruction syntax as an immediate operand. The address may be an absolute or relative address. See *indirect*.

displacement

A signed value that is added to the base of a segment (absolute addressing) or an instruction pointer (relative addressing). Same as *offset*.

doubleword

Two words, or four bytes, or 32 bits.

double quadword

Eight words, or 16 bytes, or 128 bits. Also called *octword*.

effective address size

The address size for the current instruction after accounting for the default address size and any address-size override prefix.

effective operand size

The operand size for the current instruction after accounting for the default operand size and any operand-size override prefix.

element

See *vector*.

exception

An abnormal condition that occurs as the result of instruction execution. Processor response to an exception depends on the type of exception. For all exceptions except SSE floating-point exceptions and x87 floating-point exceptions, control is transferred to a handler (or service routine) for that exception as defined by the exception's vector. For floating-point exceptions defined by the IEEE 754 standard, there are both masked and unmasked responses. When unmasked, the exception handler is called, and when masked, a default response is provided instead of calling the handler.

extended SSE instructions

Enhanced set of SIMD instructions supporting 256-bit vector data types and allowing the specification of up to four operands. A subset of the *Streaming SIMD Extensions (SSE)*. Includes the AVX, FMA, FMA4, and XOP instructions. Compare *legacy SSE*.

flush

An often ambiguous term meaning (1) writeback, if modified, and invalidate, as in "flush the cache line," or (2) invalidate, as in "flush the pipeline," or (3) change a value, as in "flush to zero."

FMA4

Fused Multiply Add, four operand. Part of the extended SSE instruction set.

FMA

Fused Multiply Add. Part of the extended SSE instruction set.

GDT

Global descriptor table.

GIF

Global interrupt flag.

IDT

Interrupt descriptor table.

IGN

Ignored. Value written is ignored by hardware. Value returned on a read is indeterminate. See *reserved*.

indirect

Referencing a memory location whose address is in a register or other memory location. The address may be an absolute or relative address. See *direct*.

IRB

The virtual-8086 mode interrupt-redirection bitmap.

IST

The long-mode interrupt-stack table.

IVT

The real-address mode interrupt-vector table.

LDT

Local descriptor table.

legacy x86

The legacy x86 architecture.

legacy mode

An operating mode of the AMD64 architecture in which existing 16-bit and 32-bit applications and operating systems run without modification. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Legacy mode has three submodes, *real mode*, *protected mode*, and *virtual-8086 mode*.

legacy SSE instructions

All Streaming SIMD Extensions instructions prior to AVX, XOP, and FMA4. Legacy SSE instructions primarily utilize operands held in XMM registers. The legacy SSE instructions include the original Streaming SIMD Extensions (SSE1) and the subsequent extensions SSE2, SSE3, SSSE3, SSE4, SSE4A, SSE4.1, and SSE4.2. See *Streaming SIMD instructions*.

long mode

An operating mode unique to the AMD64 architecture. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Long mode has two submodes, *64-bit mode* and *compatibility mode*.

lsb
: Least-significant bit.

LSB
: Least-significant byte.

main memory
: Physical memory, such as RAM and ROM (but not cache memory) that is installed in a particular computer system.

mask
: (1) A control bit that prevents the occurrence of a floating-point exception from invoking an exception-handling routine. (2) A field of bits used for a control purpose.

MBZ
: Must be zero. If software attempts to set an MBZ bit to 1, a general-protection exception (#GP) occurs. See *reserved*.

memory
: Unless otherwise specified, *main memory*.

moffset
: A 16, 32, or 64-bit offset that specifies a memory operand directly, without using a ModRM or SIB byte.

msb
: Most-significant bit.

MSB
: Most-significant byte.

octword
: Same as *double quadword*.

offset
: Same as *displacement*.

overflow
: The condition in which a floating-point number is larger in magnitude than the largest, finite, positive or negative number that can be represented in the data-type format being used.

packed
: See *vector*.

PAE
: Physical-address extensions.

physical memory

Actual memory, consisting of *main memory* and cache.

probe

A check for an address in processor caches or internal buffers. *External probes* originate outside the processor, and *internal probes* originate within the processor.

protected mode

A submode of *legacy mode*.

quadword

Four words, eight bytes, or 64 bits.

RAZ

Read as zero. Value returned on a read is always zero (0) regardless of what was previously written. See *reserved*.

real-address mode, real mode

A short name for *real-address mode,* a submode of *legacy mode*.

relative

Referencing with a *displacement* (*offset*) from an instruction pointer rather than the base of a code segment. See *absolute*.

reserved

Fields marked as reserved may be used at some future time.

To preserve compatibility with future processors, reserved fields require special handling when read or written by software. Software must not depend on the state of a reserved field (unless qualified as RAZ), nor upon the ability of such fields to return a previously written state.

If a field is marked reserved without qualification, software must not change the state of that field; it must reload that field with the same value returned from a prior read.

Reserved fields may be qualified as IGN, MBZ, RAZ, or SBZ (see definitions).

REX

A legacy instruction modifier prefix that specifies 64-bit operand size and provides access to additional registers.

RIP-relative addressing

Addressing relative to the 64-bit relative instruction pointer.

SBZ

Should be zero. An attempt by software to set an SBZ bit to 1 results in undefined behavior. See *reserved*.

scalar

An atomic value existing independently of any specification of location, direction, etc., as opposed to vectors.

set

To write the value 1 to a bit or a range of bits. See *clear*.

SIMD

Single instruction, multiple data. See *vector*.

Streaming SIMD Extensions (SSE)

Instructions that operate on scalar or vector (packed) integer and floating point numbers. The SSE instruction set comprises the legacy SSE and extended SSE instruction sets.

SSE1

Original SSE instruction set. Includes instructions that operate on vector operands in both the MMX and the XMM registers.

SSE2

Extensions to the SSE instruction set.

SSE3

Further extensions to the SSE instruction set.

SSSE3

Further extensions to the SSE instruction set.

SSE4.1

Further extensions to the SSE instruction set.

SSE4.2

Further extensions to the SSE instruction set.

SSE4A

A minor extension to the SSE instruction set adding the instructions EXTRQ, INSERTQ, MOVNTSS, and MOVNTSD.

sticky bit

A bit that is set or cleared by hardware and that remains in that state until explicitly changed by software.

TSS

Task-state segment.

underflow

The condition in which a floating-point number is smaller in magnitude than the smallest nonzero, positive or negative number that can be represented in the data-type format being used.

vector

(1) A set of integer or floating-point values, called *elements*, that are packed into a single operand. Most media instructions use vectors as operands. Also called *packed* or *SIMD* operands.

(2) An *interrupt descriptor table* index, used to access exception handlers. See *exception*.

VEX prefix

Extended instruction encoding escape prefix. Introduces a two- or three-byte encoding escape sequence used in the encoding of AVX instructions. Opens a new extended instruction encoding space. Fields select the opcode map and allow the specification of operand vector length and an additional operand register. See *XOP prefix*.

virtual-8086 mode

A submode of *legacy mode*.

VMCB

Virtual machine control block.

VMM

Virtual machine monitor.

word

Two bytes, or 16 bits.

x86

See *legacy x86*.

XOP instructions

Part of the extended SSE instruction set using the XOP prefix. See Streaming SIMD Extensions.

XOP prefix

Extended instruction encoding escape prefix. Introduces a three-byte escape sequence used in the encoding of XOP instructions. Opens a new extended instruction encoding space distinct from the VEX opcode space. Fields select the opcode map and allow the specification of operand vector length and an additional operand register. See *VEX prefix*.

Registers

In the following list of registers, mnemonics refer either to the register itself or to the register content:

AH–DH

The high 8-bit AH, BH, CH, and DH registers. See *[AL–DL]*.

AL–DL

The low 8-bit AL, BL, CL, and DL registers. See *[AH–DH]*.

AL–r15B

The low 8-bit AL, BL, CL, DL, SIL, DIL, BPL, SPL, and [r8B–r15B] registers, available in 64-bit mode.

BP

Base pointer register.

CR*n*

Control register number *n*.

CS

Code segment register.

eAX–eSP

The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers. See *[rAX–rSP]*.

EFER

Extended features enable register.

eFLAGS

16-bit or 32-bit flags register. See *rFLAGS*.

EFLAGS

32-bit (extended) flags register.

eIP

16-bit or 32-bit instruction-pointer register. See *rIP*.

EIP

32-bit (extended) instruction-pointer register.

FLAGS

16-bit flags register.

GDTR

Global descriptor table register.

GPRs

General-purpose registers. For the 16-bit data size, these are AX, BX, CX, DX, DI, SI, BP, and SP. For the 32-bit data size, these are EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP. For the 64-bit data size, these include RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and R8–R15.

IDTR
: Interrupt descriptor table register.

IP
: 16-bit instruction-pointer register.

LDTR
: Local descriptor table register.

MSR
: Model-specific register.

r8–r15
: The 8-bit R8B–R15B registers, or the 16-bit R8W–R15W registers, or the 32-bit R8D–R15D registers, or the 64-bit R8–R15 registers.

rAX–rSP
: The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers, or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers, or the 64-bit RAX, RBX, RCX, RDX, RDI, RSI, RBP, and RSP registers. Replace the placeholder *r* with nothing for 16-bit size, “E” for 32-bit size, or “R” for 64-bit size.

RAX
: 64-bit version of the EAX register.

RBP
: 64-bit version of the EBP register.

RBX
: 64-bit version of the EBX register.

RCX
: 64-bit version of the ECX register.

RDI
: 64-bit version of the EDI register.

RDX
: 64-bit version of the EDX register.

rFLAGS
: 16-bit, 32-bit, or 64-bit flags register. See *RFLAGS*.

RFLAGS
: 64-bit flags register. See *rFLAGS*.

rIP

16-bit, 32-bit, or 64-bit instruction-pointer register. See *RIP*.

RIP

64-bit instruction-pointer register.

RSI

64-bit version of the ESI register.

RSP

64-bit version of the ESP register.

SP

Stack pointer register.

SS

Stack segment register.

TPR

Task priority register (CR8).

TR

Task register.

YMM/XMM

Set of sixteen (eight accessible in legacy and compatibility modes) 256-bit wide registers that hold scalar and vector operands used by the SSE instructions.

Endian Order

The x86 and AMD64 architectures address memory using little-endian byte-ordering. Multibyte values are stored with the least-significant byte at the lowest byte address, and illustrated with their least significant byte at the right side. Strings are illustrated in reverse order, because the addresses of string bytes increase from right to left.

Related Documents

- Peter Abel, *IBM PC Assembly Language and Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Rakesh Agarwal, *80x86 Architecture & Programming: Volume II*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- AMD, *AMD-K6™ MMX™ Enhanced Processor Multimedia Technology*, Sunnyvale, CA, 2000.
- AMD, *3DNow!™ Technology Manual*, Sunnyvale, CA, 2000.
- AMD, *AMD Extensions to the 3DNow!™ and MMX™ Instruction Sets*, Sunnyvale, CA, 2000.
- Don Anderson and Tom Shanley, *Pentium Processor System Architecture*, Addison-Wesley, New York, 1995.
- Nabajyoti Barkakati and Randall Hyde, *Microsoft Macro Assembler Bible*, Sams, Carmel, Indiana, 1992.
- Barry B. Brey, *8086/8088, 80286, 80386, and 80486 Assembly Language Programming*, Macmillan Publishing Co., New York, 1994.
- Barry B. Brey, *Programming the 80286, 80386, 80486, and Pentium Based Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Ralf Brown and Jim Kyle, *PC Interrupts*, Addison-Wesley, New York, 1994.
- Penn Brumm and Don Brumm, *80386/80486 Assembly Language Programming*, Windcrest McGraw-Hill, 1993.
- Geoff Chappell, *DOS Internals*, Addison-Wesley, New York, 1994.
- Chips and Technologies, Inc. *Super386 DX Programmer's Reference Manual*, Chips and Technologies, Inc., San Jose, 1992.
- John Crawford and Patrick Gelsinger, *Programming the 80386*, Sybex, San Francisco, 1987.
- Cyrix Corporation, *5x86 Processor BIOS Writer's Guide*, Cyrix Corporation, Richardson, TX, 1995.
- Cyrix Corporation, *M1 Processor Data Book*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor MMX Extension Opcode Table*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor Data Book*, Cyrix Corporation, Richardson, TX, 1997.
- Ray Duncan, *Extending DOS: A Programmer's Guide to Protected-Mode DOS*, Addison Wesley, NY, 1991.
- William B. Giles, *Assembly Language Programming for the Intel 80xxx Family*, Macmillan, New York, 1991.
- Frank van Gilluwe, *The Undocumented PC*, Addison-Wesley, New York, 1994.
- John L. Hennessy and David A. Patterson, *Computer Architecture*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.
- Thom Hogan, *The Programmer's PC Sourcebook*, Microsoft Press, Redmond, WA, 1991.

- Hal Katircioglu, *Inside the 486, Pentium, and Pentium Pro*, Peer-to-Peer Communications, Menlo Park, CA, 1997.
- IBM Corporation, *486SLC Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *486SLC2 Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *80486DX2 Processor Floating Point Instructions*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *80486DX2 Processor BIOS Writer's Guide*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *Blue Lightning 486DX2 Data Book*, IBM Corporation, Essex Junction, VT, 1994.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, ANSI/IEEE Std 854-1987.
- Muhammad Ali Mazidi and Janice Gillispie Mazidi, *80X86 IBM PC and Compatible Computers*, Prentice-Hall, Englewood Cliffs, NJ, 1997.
- Hans-Peter Messmer, *The Indispensable Pentium Book,* Addison-Wesley, New York, 1995.
- Karen Miller, *An Assembly Language Introduction to Computer Architecture: Using the Intel Pentium*, Oxford University Press, New York, 1999.
- Stephen Morse, Eric Isaacson, and Douglas Albert, *The 80386/387 Architecture*, John Wiley & Sons, New York, 1987.
- NexGen Inc., *Nx586 Processor Data Book*, NexGen Inc., Milpitas, CA, 1993.
- NexGen Inc., *Nx686 Processor Data Book*, NexGen Inc., Milpitas, CA, 1994.
- Bipin Patwardhan, *Introduction to the Streaming SIMD Extensions in the Pentium III*, www.x86.org/articles/sse_pt1/ simd1.htm, June, 2000.
- Peter Norton, Peter Aitken, and Richard Wilton, *PC Programmer's Bible*, Microsoft Press, Redmond, WA, 1993.
- *PharLap 386|ASM Reference Manual*, Pharlap, Cambridge MA, 1993.
- *PharLap TNT DOS-Extender Reference Manual*, Pharlap, Cambridge MA, 1995.
- Sen-Cuo Ro and Sheau-Chuen Her, *i386/i486 Advanced Programming*, Van Nostrand Reinhold, New York, 1993.
- Jeffrey P. Royer, *Introduction to Protected Mode Programming*, course materials for an onsite class, 1992.
- Tom Shanley, *Protected Mode System Architecture*, Addison Wesley, NY, 1996.
- SGS-Thomson Corporation, *80486DX Processor SMM Programming Manual*, SGS-Thomson Corporation, 1995.

- Walter A. Triebel, *The 80386DX Microprocessor*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- John Wharton, *The Complete x86*, MicroDesign Resources, Sebastopol, California, 1994.
- Web sites and newsgroups:
  - www.amd.com
  - news.comp.arch
  - news.comp.lang.asm.x86
  - news.intel.microprocessors
  - news.microsoft

1 Introduction

Processors capable of performing the same mathematical operation simultaneously on multiple data streams are classified as *single-instruction, multiple-data (SIMD)*. Instructions that utilize this hardware capability are called SIMD instructions.

Software can utilize SIMD instructions to drastically increase the performance of media applications which typically employ algorithms that perform the same mathematical operation on a set of values in parallel. The original SIMD instruction set was called MMX and operated on 64-bit wide vectors of integer and floating-point elements. Subsequently a new SIMD instruction set called the Streaming SIMD Extensions (SSE) was added to the architecture.

The SSE instruction set defines a new programming model with its own array of vector data registers (YMM/XMM registers) and a control and status register (MXCSR). Most SSE instructions pull their operands from one or more YMM/XMM registers and store results in a YMM/XMM register, although some instructions use a GPR as either a source or destination. Most instructions allow one operand to be loaded from memory. The set includes instructions to load a YMM/XMM register from memory (aligned or unaligned) and store the contents of a YMM/XMM register.

An overview of the SSE instruction set is provided in Volume 1, Chapter 4.

This volume provides detailed descriptions of each instruction within the SSE instruction set. The SSE instruction set comprises the *legacy* SSE instructions and the *extended* SSE instructions.

Legacy SSE instructions comprise the following subsets:

- The original Streaming SIMD Extensions (herein referred to as SSE1)
- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- SSE4A
- Advanced Encryption Standard (AES)

Extended SSE instructions comprise the following subsets:

- AVX
- AVX2
- FMA
- FMA4
- XOP

Legacy SSE architecture supports operations involving 128-bit vectors and defines the base programming model including the SSE registers, the Media eXtension Control and Status Register (MXCSR), and the instruction exception behavior.

The Streaming SIMD Extensions (SSE) instruction set is extended to include the AVX, FMA, FMA4, and XOP instruction sets. The AVX instruction set provides an extended form for most legacy SSE instructions and several new instructions. Extensions include providing for the specification of a unique destination register for operations with two or more source operands and support for 256-bit wide vectors. Some AVX instructions also provide enhanced functionality compared to their legacy counterparts.

A significant feature of the extended SSE instruction set architecture is the doubling of the width of the XMM registers. These registers are referred to as the YMM registers. The XMM registers overlay the lower octword (128 bits) of the YMM registers. Registers YMM/XMM0–7 are accessible in legacy and compatibility mode. Registers YMM/XMM8–15 are available in 64-bit mode (a subset of long mode). VEX/XOP instruction prefixes allow instruction encodings to address the additional registers.

The SSE instructions can be used in processor legacy mode or long (64-bit) mode. CPUID Fn8000_0001_EDX[LM] indicates the availability of long mode.

Compilation for execution in 64-bit mode offers the following advantages:

- Access to an additional eight YMM/XMM registers for a total of 16
- Access to an additional eight 64-bit general-purpose registers for a total of 16
- Access to the 64-bit virtual address space and the RIP-relative addressing mode

Hardware support for each of the subsets of SSE instructions listed above is indicated by CPUID feature flags. Refer to Volume 3, Appendix D, “Instruction Subsets and CPUID Feature Flags,” for a complete list of instruction-related feature flags. The CPUID feature flags that pertain to each instruction are also given in the instruction descriptions below. For information on using the CPUID instruction, see the instruction description in Volume 3.

Chapter 2, “Instruction Reference” contains detailed descriptions of each instruction, organized in alphabetic order by mnemonic. For those legacy SSE instructions that have an AVX form, the extended form of the instruction is described together with the legacy instruction in one entry. For these instructions, the instruction reference page is located based on the instruction mnemonic of the legacy SSE and not the extended (AVX) form. Those AVX instructions without a legacy form are listed in order by their AVX mnemonic. The mnemonic for all extended SSE instructions including the FMA and XOP instructions begin with the letter *V*.

1.1 Syntax and Notation

The descriptive synopsis of opcode syntax for legacy SSE instructions follows the conventions described in *Volume 3: General Purpose and System Instructions*. See Chapter 2 and the section entitled “Notation.”

For general information on the programming model and overview descriptions of the SSE instruction set, see:

- “Streaming SIMD Extensions Media and Scientific Programming” in Volume 1.
- “Instruction Encoding” in Volume 3
- “Summary of Registers and Data Types” in Volume 3.

The syntax of the extended instruction sets requires an expanded synopsis. The expanded synopsis includes a mnemonic summary and a summary of prefix sequence fields. Figure 1-1 shows the descriptive synopsis of a typical XOP instruction. The synopsis of VEX-encoded instructions have the same format, differing only in regard to the instruction encoding escape prefix, that is, VEX instead of XOP.

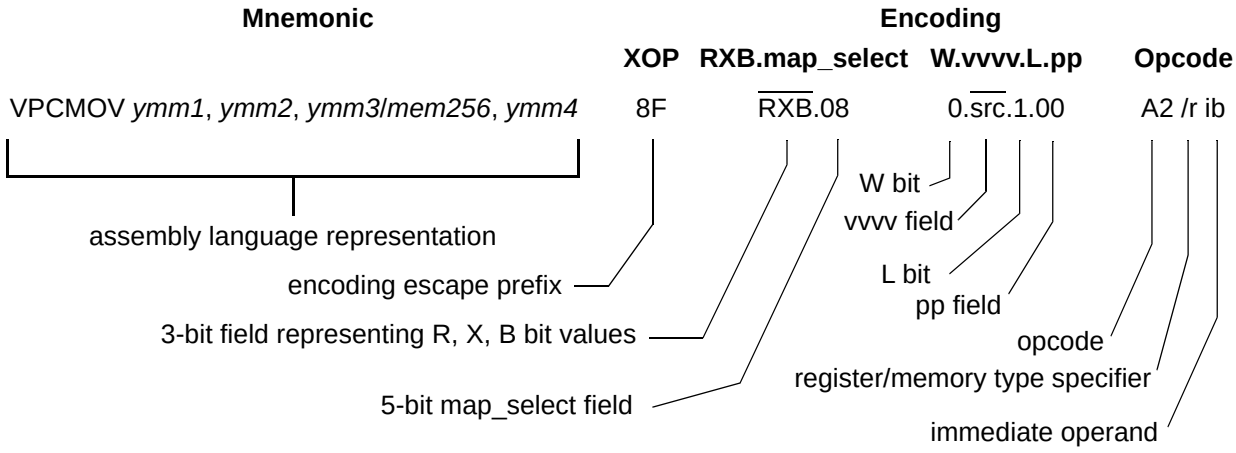


Figure 1-1. Typical Descriptive Synopsis - Extended SSE Instructions

1.2 Extended Instruction Encoding

The legacy SSE instructions are encoded using the legacy encoding syntax and the extended instructions are encoded using an enhanced encoding syntax which is compatible with the legacy syntax. Both are described in detail in Chapter 1 of Volume 3.

As described in Volume 3, the extended instruction encoding syntax utilizes multi-byte escape sequences to both select alternate opcode maps as well as augment the encoding of the instruction. Multi-byte escape sequences are introduced by one of the two VEX prefixes or the XOP prefix.

The AVX and AVX2 instructions utilize either the two-byte (introduced by the VEX C5h prefix) or the three-byte (introduced by the VEX C4h prefix) encoding escape sequence. XOP instructions are encoded using a three-byte encoding escape sequence introduced by the XOP prefix (except for the XOP instructions VPERMIL2PD and VPERMIL2PS which are encoded using the VEX prefix). The XOP prefix is 8Fh. The three-byte encoding escape sequences utilize the map_select field of the second byte to select the opcode map used to interpret the opcode byte.

The two-byte VEX prefix sequence implicitly selects the secondary (“two-byte”) opcode map.

1.2.1 Immediate Byte Usage Unique to the SSE instructions

An *immediate* is a value, typically an operand, explicitly provided within the instruction encoding. Depending on the opcode and the operating mode, the size of an immediate operand can be 1, 2, 4, or 8 bytes. Legacy and extended media instructions typically use an immediate byte operand (*imm8)*.

A one-byte immediate is generally shown in the instruction synopsis as “ib” suffix. For extended SSE instructions with four source operands, the suffix “is4” is used to indicate the presence of the immediate byte used to select the fourth source operand.

The VPERMIL2PD and VPERMIL2PS instructions utilize a fifth 2-bit operand which is encoded along with the fourth register select index in an immediate byte. For this special case the immediate byte will be shown in the instruction synopsis as “is5”.

1.2.2 Instruction Format Examples

The following sections provide examples of two-, three-, and four-operand extended instructions. These instructions generally perform *nondestructive-source operations*, meaning that the result of the operation is written to a separately specified destination register rather than overwriting one of the source operands. This preserves the contents of the source registers. Most legacy SSE instructions perform *destructive-source operations*, in which a single register is both source and destination, so source content is lost.

1.2.2.1 XMM Register Destinations

The following general properties apply to YMM/XMM register destination operands.

- For legacy instructions that use XMM registers as a destination: When a result is written to a destination XMM register, bits [255:128] of the corresponding YMM register are not affected.
- For extended instructions that use XMM registers as a destination: When a result is written to a destination XMM register, bits [255:128] of the corresponding YMM register are cleared.

1.2.2.2 Two Operand Instructions

Two-operand instructions use ModRM-based operand assignment. For most instructions, the first operand is the destination, selected by the ModRM.reg field, and the second operand is either a register or a memory source, selected by the ModRM.r/m field.

VCVTDQ2PD is an example of a two-operand AVX instruction.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTDQ2PD *xmm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.01	0.1111.0.10	E6 /r
VCVTDQ2PD *ymm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.01	0.1111.1.10	E6 /r

The destination register is selected by ModRM.reg. The size of the destination register is determined by VEX.L. The source is either a YMM/XMM register or a memory location specified by ModRM.r/m Because this instruction converts packed doubleword integers to double-precision floating-point values, the source data size is smaller than the destination data size.

VEX.vvvv is not used and must be set to 1111b.

1.2.2.3 Three-Operand Instructions

These extended instructions have two source operands and a destination operand.

VPROTB is an example of a three-operand XOP instruction.

There are versions of the instruction for variable-count rotation and for fixed-count rotation.

VPROTB *dest, src, variable-count*

VPROTB *dest, src, fixed-count*

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPROTB *xmm1*, *xmm2/mem128, xmm3*	8F	$\overline{\text{RXB}}$.09	0.$\overline{src}$.0.00	90 /r
VPROTB *xmm1*, *xmm2, xmm3/mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	90 /r
VPROTB *xmm1*, *xmm2/mem128*, *imm8*	8F	$\overline{\text{RXB}}$.08	0.1111.0.00	90 /r ib

For both versions of the instruction, the destination (*dest*) operand is an XMM register specified by ModRM.reg.

The variable-count version of the instruction rotates each byte of the source as specified by the corresponding byte element *variable-count*.

Selection of *src* and *variable-count* is controlled by XOP.W.

- When XOP.W = 0, *src* is either an XMM register or a 128-bit memory location specified by ModRM.r/m, and *variable-count* is an XMM register specified by XOP.vvvv.
- When XOP.W = 1, *src* is an XMM register specified by XOP.vvvv and *variable-count* is either an XMM register or a 128-bit memory location specified by ModRM.r/m.

Table 1-1 summarizes the effect of the XOP.W bit on operand selection.

Table 1-1. Three-Operand Selection

XOP.W	*dest*	*src*	*variable-count*
0	ModRM.reg	ModRM.r/m	XOP.vvvv
1	ModRM.reg	XOP.vvvv	ModRM.r/m

The fixed-count version of the instruction rotates each byte of *src* as specified by the immediate byte operand *fixed-count*. For this version, $\overline{src}$ is either an XMM register or a 128-bit memory location

specified by ModRM.r/m. Because XOP.vvvv is not used to specify the source register, it must be set to 1111b or execution of the instruction will cause an Invalid Opcode (#UD) exception.

1.2.2.4 Four-Operand Instructions

Some extended instructions have three source operands and a destination operand. This is accomplished by using the VEX/XOP.vvvv field, the ModRM.reg and ModRM.r/m fields, and bits [7:4] of an immediate byte to select the operands. The opcode suffix “is4” is used to identify the immediate byte, and the selected operands are shown in the synopsis.

VFMSUBPD is an example of an four-operand FMA4 instruction.

VFMSUBPD *dest, src1, src2, src3* *dest = src1* src2 - src3*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMSUBPD *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src}$.0.01	6D /r is4
VFMSUBPD *ymm1*, *ymm2*, *ymm3/mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src}$.1.01	6D /r is4
VFMSUBPD *xmm1*, *xmm2, xmm3, xmm4/mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src}$.0.01	6D /r is4
VFMSUBPD *ymm1*, *ymm2, ymm3, ymm4/mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src}$.1.01	6D /r is4

The first operand, the destination (*dest*), is an XMM register or a YMM register (as determined by VEX.L) selected by ModRM.reg. The following three operands (*src1*, *src2*, *src3*) are sources.

The *src1* operand is an XMM or YMM register specified by VEX.vvvv.

VEX.W determines the configuration of the *src2* and *src3* operands.

- When VEX.W = 0, *src2* is either a register or a memory location specified by ModRM.r/m, and *src3* is a register specified by bits [7:4] of the immediate byte.
- When VEX.W = 1, *src2* is a register specified by bits [7:4] of the immediate byte and *src3* is either a register or a memory location specified by ModRM.r/m.

Table 1-1 summarizes the effect of the VEX.W bit on operand selection.

Table 1-2. Four-Operand Selection

VEX.W	*dest*	*src1*	*src2*	*src3*
0	ModRM.reg	VEX.vvvv	ModRM.r/m	is4[7:4]
1	ModRM.reg	VEX.vvvv	is4[7:4]	ModRM.r/m

1.3 VSIB Addressing

Specific AVX2 instructions utilize a vectorized form of indexed register-indirect addressing called vector SIB (VSIB) addressing. In contrast to the standard indexed register-indirect address mode, which generates a single effective address to access a single memory operand, VSIB addressing generates an array of effective addresses which is used to access data from multiple memory locations in a single operation.

VSIB addressing is encoded using three or six bytes following the opcode byte, augmented by the X and B bits from the VEX prefix. The first byte is the ModRM byte with the standard mod, reg, and r/m fields (although allowed values for the mod and r/m fields are restricted). The second is the VSIB byte which replaces the SIB byte in the encoding. The VSIB byte specifies a GPR which serves as a base address register and an XMM/YMM register that contains a packed array of index values. The two-bit scale field specifies a common scaling factor to be applied to all of the index values. A constant displacement value is encoded in the one or four bytes that follow the VSIB byte.

Figure 1-2 shows the format of the VSIB byte.

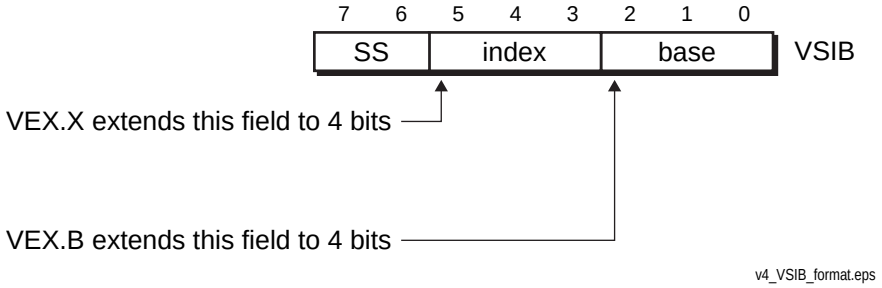


Figure 1-2. VSIB Byte Format

VSIB.SS (Bits [7:6]). The SS field is used to specify the scale factor to be used in the computation of each of the effective addresses. The scale factor *scale* is equal to 2^{SS} (two raised to power of the value of the SS field). Therefore, if SS = 00b, *scale* = 1; if SS = 01b, *scale* = 2; if SS = 10b, *scale* = 4; and if SS = 11b, *scale* = 8.

VSIB.index (Bits [5:3]). This field is concatenated with the complement of the VEX.X bit ({$\overline{X}$, index}) to specify the YMM/XMM register that contains the packed array of index values *index*[*i*] to be used in the computation of the array of effective addresses *effective address*[*i*].

VSIB.base (Bits [5:3]). This field is concatenated with the complement of the VEX.B bit ({$\overline{B}$, base}) to specify the general-purpose register (base GPR) that contains the base address *base* to be used in the computation of each of the effective addresses.

1.3.1 Effective Address Array Computation

Each element *i* of the effective address array is computed using the formula:

effective address[*i*] = *scale* * *index*[*i*] + *base* + *displacement*.

where *index*[*i*] is the *i*th element of the XMM/YMM register specified by {$\overline{X}$,VSIB.index}. An index element is either 32 or 64 bits wide and is treated as a signed integer.

Variants of this mode use either an eight-bit or a 32-bit displacement value. One variant sets the base to zero. The value of the ModRM.mod field specifies the specific variant of VSIB addressing mode, as shown in Table 1. In the table, the notation [XMM*n*/YMM*n*] indicates the XMM/YMM register that contains the packed index array and [base GPR] means the contents of the base GPR selected by {$\overline{B}$, base}.

Table 1: Vectorized Addressing Modes

Index[1]	ModRM.mod		
	00	01	10
0000	scale * [XMM0/YMM0] + Disp32	scale * [XMM0/YMM0] + Disp8 + [base GPR]	scale * [XMM0/YMM0] + Disp32 + [base GPR]
0001	scale * [XMM1/YMM1] + Disp32	scale * [XMM1/YMM1] + Disp8 + [base GPR]	scale * [XMM1/YMM1] + Disp32 + [base GPR]
0010	scale * [XMM2/YMM2] + Disp32	scale * [XMM2/YMM2] + Disp8 + [base GPR]	scale * [XMM2/YMM2] + Disp32 + [base GPR]
0011	scale * [XMM3/YMM3] + Disp32	scale * [XMM3/YMM3] + Disp8 + [base GPR]	scale * [XMM3/YMM3] + Disp32 + [base GPR]
0100	scale * [XMM4/YMM4] + Disp32	scale * [XMM4/YMM4] + Disp8 + [base GPR]	scale * [XMM4/YMM4] + Disp32 + [base GPR]
0101	scale * [XMM5/YMM5] + Disp32	scale * [XMM5/YMM5] + Disp8 + [base GPR]	scale * [XMM5/YMM5] + Disp32 + [base GPR]
0110	scale * [XMM6/YMM6] + Disp32	scale * [XMM6/YMM6] + Disp8 + [base GPR]	scale * [XMM6/YMM6] + Disp32 + [base GPR]
0111	scale * [XMM7/YMM7] + Disp32	scale * [XMM7/YMM7] + Disp8 + [base GPR]	scale * [XMM7/YMM7] + Disp32 + [base GPR]
1000	scale * [XMM8/YMM8] + Disp32	scale * [XMM8/YMM8] + Disp8 + [base GPR]	scale * [XMM8/YMM8] + Disp32 + [base GPR]
1001	scale * [XMM9/YMM9] + Disp32	scale * [XMM9/YMM9] + Disp8 + [base GPR]	scale * [XMM9/YMM9] + Disp32 + [base GPR]
1010	scale * [XMM10/YMM10] + Disp32	scale * [XMM10/YMM10] + Disp8 + [base GPR]	scale * [XMM10/YMM10] + Disp32 + [base GPR]
1011	scale * [XMM11/YMM11] + Disp32	scale * [XMM11/YMM11] + Disp8 + [base GPR]	scale * [XMM11/YMM11] + Disp32 + [base GPR]
1100	scale * [XMM12/YMM12] + Disp32	scale * [XMM12/YMM12] + Disp8 + [base GPR]	scale * [XMM12/YMM12] + Disp32 + [base GPR]
1101	scale * [XMM13/YMM13] + Disp32	scale * [XMM13/YMM13] + Disp8 + [base GPR]	scale * [XMM13/YMM13] + Disp32 + [base GPR]
1110	scale * [XMM14/YMM14] + Disp32	scale * [XMM14/YMM14] + Disp8 + [base GPR]	scale * [XMM14/YMM14] + Disp32 + [base GPR]
1111	scale * [XMM15/YMM15] + Disp32	scale * [XMM15/YMM15] + Disp8 + [base GPR]	scale * [XMM15/YMM15] + Disp32 + [base GPR]

Note *1. Index = {$\overline{VEX.X}$,VSIB.index}. In 32-bit mode, VEX.X = 1.*

1.3.2 Notational Conventions Related to VSIB Addressing Mode

In the instruction descriptions that follow, the notation vm32x indicates a packed array of four 32-bit index values contained in the specified XMM index register and vm32y indicates a packed array of eight 32-bit index values contained in the specified YMM index register. Depending on the instruction, these indices can be used to compute the effective address of up to four (vm32x) or eight (vm32y) memory-based operands.

The notation vm64x indicates a packed array of two 64-bit index values contained in the specified XMM index register and vm64y indicates a packed array of four 64-bit index values contained in the specified YMM index register. Depending on the instruction, these indices can be used to compute the effective address of up to two (vm64x) or four (vm64y) memory-based operands.

In body of the description of the instructions, the notation mem32[vm32x] is used to represent a sparse array of 32-bit memory operands where the packed array of four 32-bit indices used to calculate the effective addresses of the operands is held in an XMM register. The notation mem32[vm32y] refers to a similar array of 32-bit memory operands where the packed array of eight 32-bit indices is held in a YMM register. The notation mem32[vm64x] means a sparse array of 32-bit memory operands where the packed array of two 64-bit indices is held in an XMM register and mem32[vm64y] means a sparse array of 32-bit memory operands where the packed array of four 64-bit indices is held in a YMM register.

The notation mem64[*index_array*], where *index_array* is either vm32x, vm64x, or vm64y, specifies a sparse array of 64-bit memory operands addressed via a packed array of 32-bit or 64-bit indices held in an XMM/YMM register. If an instruction uses either an XMM or a YMM register, depending on operand size, to hold the index array, the notation vm32x/y or vm64x/y is used to represent the array.

In summary, given a maximum operand size of 256-bits, a sparse array of 32-bit memory-based operands can be addressed using a vm32x, vm32y, vm64x, or vm64y index array. A sparse array of 64-bit memory-based operands can be addressed using a vm32x, vm64x, or vm64y index array. Specific instructions may use fewer than the maximum number of memory operands that can be addressed using the specified index array.

VSIB addressing is only valid in 32-bit or 64-bit effective addressing mode and is only supported for instruction encodings using the VEX prefix. The ModRM.mod value of 11b is not valid in VSIB addressing mode and ModRM.r/m must be set to 100b.

1.3.3 Memory Ordering and Exception Behavior

VSIB addressing has some special considerations relative to memory ordering and the signaling of exceptions.

VSIB addressing specifies an array of addresses that allows an instruction to access multiple memory locations. The order in which data is read from or written to memory is not specified. Memory ordering with respect to other instructions follows the memory-ordering model described in Volume 2.

Data may be accessed by the instruction in any order, but access-triggered exceptions are delivered in right-to-left order. That is, if a exception is triggered by the load or store of an element of an XMM/YMM register and delivered, all elements to the right of that element (all the lower indexed elements) have been or will be completed without causing an exception. Elements to the left of the element causing the exception may or may not be completed. If the load or store of a given element triggers multiple exceptions, they are delivered in the conventional order.

Because data can be accessed in any order, elements to the left of the one that triggered the exception may be read or written before the exception is delivered. Although the ordering of accesses is not specified, it is repeatable in a specific processor implementation. Given the same input values and initial architectural state, the same set of elements to the left of the faulting one will be accessed.

VSIB addressing should not be used to access memory mapped I/O as the ordering of the individual loads is implementation-specific and some implementations may access data larger than the data element size or access elements more than once.

1.4 Enabling SSE Instruction Execution

Application software that utilizes the SSE instructions requires support from operating system software.

To enable and support SSE instruction execution, operating system software must:

- enable hardware for supported SSE subsets
- manage the SSE hardware architectural state, saving and restoring it as required during and after task switches
- provide exception handlers for all unmasked SSE exceptions.

See Volume 2, Chapter 11, for details on enabling SSE execution and managing its execution state.

1.5 String Compare Instructions

The legacy SSE instructions PCMPESTRI, PCMPESTRM, PCMPISTRI, and PCMPISTRM and the extended SSE instructions VPCMPESTRI, VPCMPESTRM, VPCMPISTRI, and VPCMPISTRM provide a versatile means of classifying characters of a string by performing one of several different types of comparison operations using a second string as a prototype.

This section describes the operation of the legacy string compare instructions. This discussion applies equally to the extended versions of the instructions. Any difference between the legacy and the extended version of a given instruction is described in the instruction reference entry for the instruction in the following chapter.

A character string is a vector of data elements that is normally used to represent an ordered arrangement of graphemes which may be stored, processed, displayed, or printed. Ordered strings of graphemes are most often used to convey information in a human-readable manner. The string compare instructions, however, do not restrict the use or interpretation of their operands.

The first source operand provides the prototype string and the second operand is the string to be scanned and characterized (referred to herein as the string under test, or SUT). Four string formats and four types of comparisons are supported. The intermediate result of this processing is a bit vector that summarizes the characterization of each character in the SUT. This bit vector is then post-processed based on options specified in the instruction encoding. Instruction variants determine the final result—either an index or a mask.

Instruction execution affects the arithmetic status flags (ZF, CF, SF, OF, AF, PF), but the significance of many of the flags is redefined to provide information tailored to the result of the comparison performed. See Section 1.5.6, “Affect on Flags” on page 19.

The instructions have a defined base function and additional functionality controlled by bit fields in an immediate byte operand (*imm8*). The base function determines whether the source strings have implicitly (PCMPISTRI and PCMPISTRM) or explicitly (PCMPESTRI and PCMPESTRM) defined lengths, and whether the result is an index (PCMPISTRI and PCMPESTRI) or a mask (PCMPISTRM and PCMPESTRM).

PCMPISTRI and PCMPESTRI return their final result (an integer value) via the ECX register, while PCMPISTRM and PCMPESTRM write a bit or character mask, depending on the option selected, to the XMM0 register.

There are a number of different schemes for encoding a set of graphemes, but the most common ones use either an 8-bit code (ASCII) or a 16-bit code (unicode). The string compare instructions support both character sizes.

Bit fields of the immediate operand control the following functions:

- Source data format — character size (byte or word), signed or unsigned values
- Comparison type
- Intermediate result postprocessing
- Output option selection

This overview description covers functions common to all of the string compare instructions and describes some of the differentiated features of specific instructions. Information on instruction encoding and exception behavior are covered in the individual instruction reference pages in the following chapter.

1.5.1 Source Data Format

The character strings that constitute the source operands for the string compare instructions are formatted as either 8-bit or 16-bit integer values packed into a 128-bit data type. The figure below illustrates how a string of byte-wide characters is laid out in memory and how these characters are arranged when loaded into an XMM register.

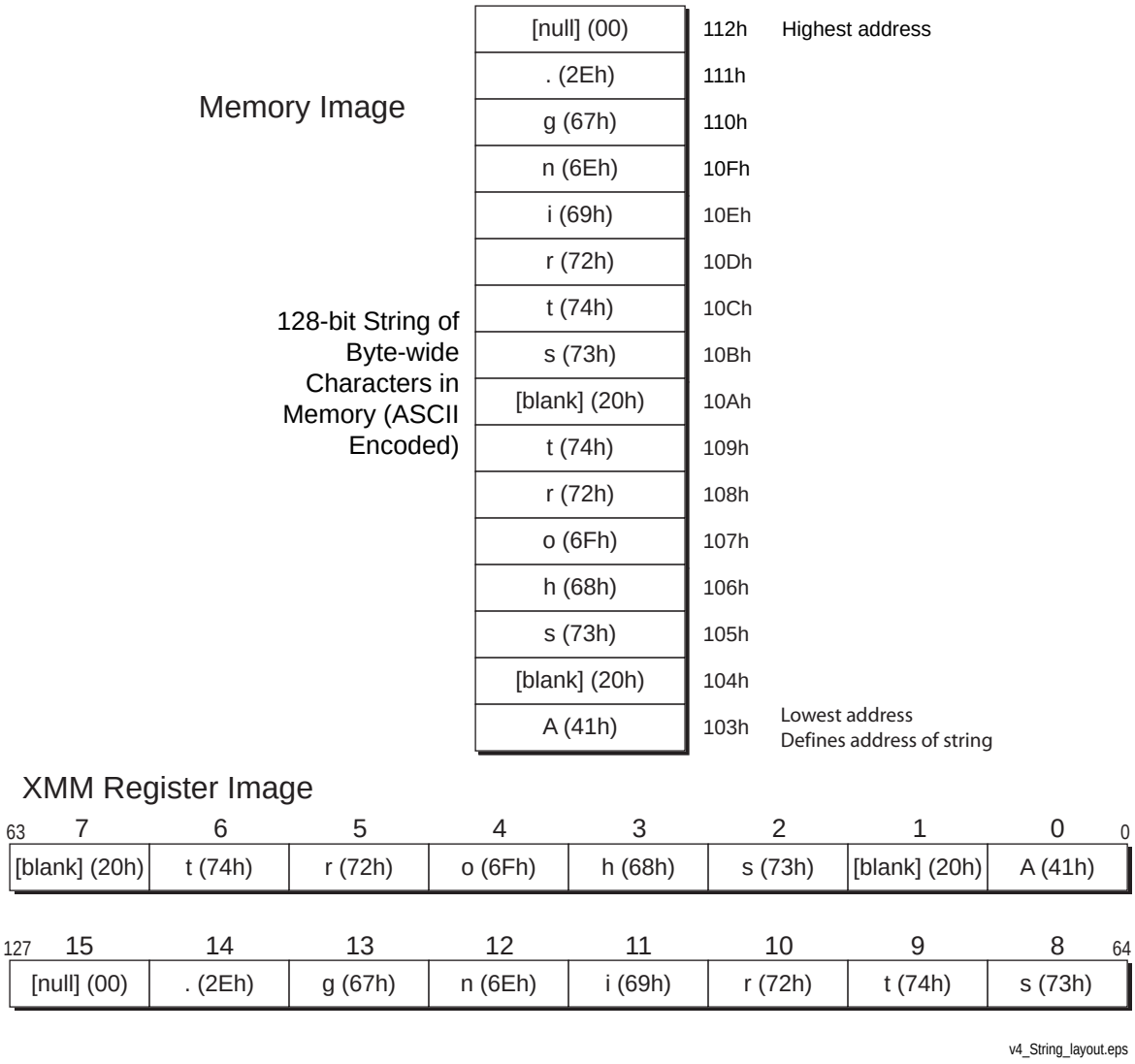


Figure 1-3. Byte-wide Character String – Memory and Register Image

Note from the figure that the longest string that can be packed in a 128-bit data object is either sixteen 8-bit characters (as illustrated) or eight 16-bit characters. When loaded from memory, the character read from the lowest address in memory is placed in the least-significant position of the register and the character read from the highest address is placed in the most-significant position. In other words, for character *i* of width *w*, bits [*w*−1:0] of the character are placed in bits [*iw* + (*w*−1):*iw*] of the register.

Bits [1:0] of the immediate byte operand specify the source string data format, as shown in Table 1-3.

Table 1-3. Source Data Format

Imm8[1:0]	Character Format	Maximum String Length
00b	unsigned bytes	16
01b	unsigned words	8
10b	signed bytes	16
11b	signed words	8

The string compare instructions are defined with the capability of operating on strings of lengths from 0 to the maximum that can be packed into the 128-bit data type as shown in the table above. Because strings being processed may be shorter than the maximum string length, a means is provided to designate the length of each string. As mentioned above, one pair of string compare instructions relies on an explicit method while the other utilizes an implicit method.

For the explicit method, the length of the first operand (the prototype string) is specified by the absolute value of the signed integer contained in rAX and the length of the second operand (the SUT) is specified by the absolute value of the signed integer contained in rDX. If a specified length is greater than the maximum allowed, the maximum value is used. Using the explicit method of length specification, null characters (characters whose numerical value is 0) can be included within a string.

Using the implicit method, a string shorter than the maximum length is terminated by a null character. If no null character is found in the string, its length is implied to be the maximum. For the example illustrated in Figure 1-3 above, the implicit length of the string is 15 because the final character is null. However, using the explicit method, a specified length of 16 would include the null character in the string.

In the following discussion, l_1 is the length of the first operand string (the prototype string), l_2 is the length of the second operand string (the SUT) and m is the maximum string length based on the selected character size.

1.5.2 Comparison Type

Although the string compare instructions can be implemented in many different ways, the instructions are most easily understood as the sequential processing of the SUT using the characters of the prototype string as a template. The template is applied at each character index of SUT, processing the string from the first character (index 0) to the last character (index l_2-1).

The result of each comparison is recorded in successive positions of a summary bit vector *CmprSumm*. When the sequence of comparisons is complete, this bit vector summarizes the results of comparison operations that were performed. The length of the *CmprSumm* bit vector is equal to the maximum input operand string length (*m)*. The rules for the setting of *CmprSumm* bits beyond the end of the SUT (*CmprSumm*[$m-1:l_2$]) are dependent on the comparison type (see Table 1-4 below.)

Bits [3:2] of the immediate byte operand determine the comparison type, as shown in Table 1-4.

Table 1-4. Comparison Type

Imm8[3:2]	Comparison Type	Description
00b	Subset	Tests each character of the SUT to determine if it is within the subset of characters specified by the prototype string. Each set bit of *CmprSumm* indicates that the corresponding character of the SUT is within the subset specified by the prototype. Bits [$m-1:l_2$] are cleared.
01b	Ranges	Tests each character of the SUT to determine if it lies within one or more ranges specified by pairs of values within the prototype string. The ranges are inclusive. Each set bit in *CmprSumm* indicates that the corresponding character of the SUT is within one or more of the inclusive ranges specified. Bits [$m-1:l_2$] are cleared. If the length of the prototype is odd, the last value in the prototype is effectively ignored.
10b	Match	Performs a character-by-character comparison between the SUT and the prototype string. Each set bit of *CmprSumm* indicates that the corresponding characters in the two strings match. If not, the bit is cleared. Bits [$m-1:\max(l_1, l_2)$] of *CmprSumm* are set.
11b	Sub-string	Searches for an exact match between the prototype string and an ordered sequence of characters (a sub-string) in the SUT beginning at the current index *i*. Bit *i* of *CmprSumm* is set for each value of *i* where the sub-string match is made, otherwise the bit is cleared. See discussion below.

In the Sub-string comparison type, any matching sub-string of the SUT must match the prototype string one-for-one, in order, and without gaps. Null characters in the SUT do not match non-null characters in the prototype. If the prototype and the SUT are equal in length and less than the max length, the two strings must be identical for the comparison to be TRUE. In this case, bit 0 of *CmprSumm* is set to one and the remainder are all 0s. If the length of the SUT is less than the prototype string, no match is possible and *CmprSumm* is all 0s.

If the prototype string is shorter than the SUT ($l_1 < l_2$), a sequential search of the SUT is performed. For each *i* from 0 to $l_2 - l_1$, the prototype is compared to characters [$i + l_1 - 1:i$] of the SUT. If the prototype and the sub-string SUT[$i + l_1 - 1:i$] match exactly, then *CmprSumm*[*i*] is set, otherwise the bit is cleared. When the comparison at $i = l_2 - l_1$ is complete, no further testing is required because there are not enough characters remaining in the SUT for a match to be possible. The remaining bits $l_2 - l_1 + 1$ through *m-1* are all set to 0.

For the Match comparison type, the character-by-character comparison is performed on all *m* characters in the 128-bit operand data, which may extend beyond the end of one or both strings. A null character at index *i* within one string is not considered a match when compared with a character beyond the end of the other string. In this case, *CmprSumm*[*i*] is cleared. For index positions beyond the end of both strings, *CmprSumm*[*i*] is set.

The following section provides more detail on the generation of the comparison summary bit vector based on the specified comparison type.

1.5.3 Comparison Summary Bit Vector

The following pseudo code provides more detail on the generation of the comparison summary bit vector *CmprSumm*. The function `CompareStrgs` defined below returns a bit vector of length *m*, the maximum length of the operand data strings.

```
bit vector CompareStrgs(ProtoType, length1, SUT, length2, CmpType, signed, m)
   doubleword vector StrUndTst   // temp vector; holds string under test
   doubleword vector StrProto    // temp vector; holds prototype string
   bit vector[m] Result          // length of vector is m

   StrProto = m{0}         //initialize m elements of StrProto to 0
   StrUndTst = m{0}        //initialize m elements of StrUndTst to 0
   Result = m{0}           //initialize result bit vector

   FOR i = 0 to length1
      StrProto[i] = signed ? SignExtend(ProtoType[i]) : ZeroExtend(ProtoType[i])
   FOR i = 0 to length2
      StrUndTst[i] = signed ? SignExtend(SUT[i]) : ZeroExtend(SUT[i])

   IF CmpType == Subset
      FOR j = 0 to length2 - 1            // j indexes SUT
         FOR i = 0 to length1 - 1         // i indexes prototype
            Result[j] |= (StrProto[i] == StrUndTst[j])

   IF CmpType == Ranges
      FOR j = 0 to length2 - 1                    // j indexes SUT
         FOR i = 0 to length1 - 2, BY 2           // i indexes prototype
            Result[j] |= (StrProto[i] <= StrUndTst[j])
                        && (StrProto[i+1] >= StrUndTst[j])

   IF CmpType == Match
      FOR i = 0 to (min(length1, length2)-1)
         Result[i] = (StrProto[i] == StrUndTst[i])
      FOR i = min(length1, length2) to (max(length1, length2)-1)
         Result[i] = 0
      FOR i = max(length1, length2) to (m-1)
         Result[i] = 1

   IF CmpType == Sub-string
      IF (length2==16)&& (length1==16)
         maxlength=15
      else
          maxlength = length2-length1
   IF length2 >= lenght1
    FOR j = 0 to maxlength             // j indexes result bit vector
         Result[j] = 1
         k = j                         // k scans the SUT
         FOR i = 0 to length1 - 1      // i scans the Prototype
            Result[j] &= (StrProto[i] == StrUndTst[k])// Result[j] is cleared if
any of the comparisons do not match
            k++

Return Result
```

Given the above definition of `CompareStrgs()`, the following pseudo code computes the value of *CmprSumm*:

```
ProtoType = contents of first source operand (xmm1)
SUT = contents of xmm2 or 128-bit value read from the specified memory location
length1 = length of first operand string     //specified implicitly or explicitly
length2 = length of second operand string    //specified implicitly or explicitly
m = Maximum String Length from Table 1-3 above
CmpType = Comparison Type from Table 1-4 above
signed = (imm8[1] == 1) ? TRUE : FALSE
bit vector [m] CmprSumm    // CmprSumm is m bits long

CmprSumm = CompareStrgs(ProtoType, length1, SUT, length2, CmpType, signed, m)
```

The following examples demonstrate the comparison summary bit vector *CmprSumm* for each comparison type. For the sake of illustration, the operand strings are represented as ASCII-encoded strings. Each character value is represented by its ASCII grapheme. Strings are displayed with the lowest indexed character on the left as they would appear when printed or displayed. *CmprSumm* is shown in reverse order with the least significant bit on the left to agree with the string presentation.

Comparison Type = Subset

```
Prototype:  ZCx
SUT:        aCx%xbZreCx
CmprSumm:   0110101001100000
```

Comparison Type = Ranges

```
Prototype:  ACax
SUT:        aCx%xbZreCx
CmprSumm:   1110110111100000
```

Comparison Type = Match

```
Prototype:  ZCx
SUT:        aCx%xbZreCx
CmprSumm:   0110000000011111
```

Comparison Type = Sub-string

```
Prototype:  ZCx
SUT:        aZCx%xCZreZCxCZ
CmprSumm:   0100000000100000
```

1.5.4 Intermediate Result Post-processing

Post-processing of the *CmprSumm* bit vector is controlled by *imm8*[5:4]. The result of this step is designated *pCmprSumm*.

Bit [4] of the immediate operand determines whether a ones' complement (bit-wise inversion) is performed on *CmprSumm*; bit [5] of the immediate operand determines whether the inversion applies to the entire comparison summary bit vector (*CmprSumm*) or just to those bits that correspond to characters within the SUT. See Table 1-5 below for the encoding of the *imm8*[5:4] field.

Table 1-5. Post-processing Options

Imm8[5:4]	Post-processing Applied
x0b	pCmprSumm = CmprSumm
01b	pCmprSumm = NOT CmprSumm
11b	pCmprSumm[*i*] = **!**CmprSumm[*i*] for $i < I_2$, pCmprSumm[*i*] = CmprSumm[*i*], for $I_2 \leq i < m$

1.5.5 Output Option Selection

For PCMPESTRI and PCMPISTRI, *imm8*[6] determines whether the index of the lowest set bit or the highest set bit of *pCmprSumm* is written to ECX, as shown in Table 1-6.

Table 1-6. Indexed Output Option Selection

Imm8[6]	Description
0b	Return the index of the least significant set bit in pCmprSumm.
1b	Return the index of the most significant set bit in pCmprSumm.

For PCMPESTRM and PCMPISTRM, *imm8*[6] specifies whether the output from the instruction is a bit mask or an expanded mask. The bit mask is a copy of *pCmprSumm* zero-extended to 128 bits. The expanded mask is a packed vector of byte or word elements, as determined by the string operand format (as indicated by *imm8*[0]). The expanded mask is generated by copying each bit of *pCmprSumm* to all bits of the element of the same index. Table 1-7 below shows the encoding of *imm8*[6].

Table 1-7. Masked Output Option Selection

Imm8[6]	Description
0b	Return pCmprSumm as the output with zero extension to 128 bits.
1b	Return expanded pCmprSumm byte or word mask.

The PCMPESTRM and PCMPISTRM instructions return their output in register XMM0. For the extended forms of the instructions, bits [127:64] of YMM0 are cleared.

1.5.6 Affect on Flags

The execution of a string compare instruction updates the state of the CF, PF, AF, ZF, SF, and OF flags within the rFLAGs register. All other flags are unaffected. The PF and AF flags are always cleared. The ZF and SF flags are set or cleared based on attributes of the source strings and the CF and OF flags are set or cleared based on attributes of the summary bit vector after post processing.

The CF flag is cleared if the summary bit vector, after post processing, is zero; the flag is set if one or more of the bits in the post-processed bit vector are 1. The OF flag is updated to match the value of the least significant bit of the post-processed summary bit vector.

The ZF flag is set if the length of the second string operand (SUT) is shorter than *m*, the maximum number of 8-bit or 16-bit characters that can be packed into 128 bits. Similarly, the SF flag is set if the length of the first string operand (prototype) is shorter than *m*.

This information is summarized in Table 1-8 below.

Table 1-8. State of Affected Flags After Execution

Unconditional		Source String Length		Post-processed Bit Vector	
PF	**AF**	**SF**	**ZF**	**CF**	**OF**
0	0	($l_1 < m$)	($l_2 < m$)	pCmprSumm ≠ 0	pCmprSumm [0]

2 Instruction Reference

Instructions are listed by mnemonic, in alphabetic order. Each entry describes instruction function, syntax, opcodes, affected flags and exceptions related to the instruction.

Figure 2-1 shows the conventions used in the descriptions. Items that do not pertain to a particular instruction, such as a synopsis of the 256-bit form, may be omitted.

INST **Instruction**
VINST **Mnemonic Expansion**

Brief functional description

INST

Description of legacy version of instruction.

VINST

Description of extended version of instruction.

XMM Encoding

Description of 128-bit extended instruction.

YMM Encoding

Description of 256-bit extended instruction.

Information about CPUID functions related to the instruction set.

Synopsis diagrams for legacy and extended versions of the instruction.

Mnemonic	Opcode	Description
INST *xmm1*, *xmm2*/*mem128*	FF FF /r	Brief summary of legacy operation.

Mnemonic	Encoding			
	VEX	RXB.mmmmm	W.vvvv.L.pp	Opcode
VINST *xmm1*, *xmm2*/*mem128, xmm3*	C4	$\overline{RXB}$.11	0.$\overline{src}$.0.00	FF */r*
VINST *ymm1*, *ymm2*/*mem256, ymm3*	C4	$\overline{RXB}$.11	0.$\overline{src}$.0.00	FF */r*

Related Instructions

Instructions that perform similar or related functions.

rFLAGS Affected

Rflags diagram.

MXCSR Flags Affected

MXCSR diagram.

Exceptions

Exception summary table.

Figure 2-1. Typical Instruction Description

Instruction Exceptions

Under various conditions instructions described below can cause exceptions. The conditions that cause these exceptions can differ based on processor mode and instruction subset. This information is summarized at the end of each instruction reference page in an Exception Table. Rows list the applicable exceptions and the different conditions that trigger each exception for the instruction. For each processor mode (real, virtual, and protected) a symbol in the table indicates whether this exception condition applies.

Each AVX instruction has a legacy form that comes from one of the legacy (SSE1, SSE2, ...) subsets. An “X” at the intersection of a processor mode column and an exception cause row indicates that the causing condition and potential exception applies to both the AVX instruction and the legacy SSE instruction. “A” indicates that the causing condition applies only to the AVX instruction and “S” indicates that the condition applies to the SSE legacy instruction.

Note that XOP and FMA4 instructions do not have corresponding instructions from the SSE legacy subsets. In the exception tables for these instructions, “X” represents the XOP instruction and “F” represents the FMA4 instruction.

ADDPD Add
VADDPD Packed Double-Precision Floating-Point

Adds each packed double-precision floating-point value of the first source operand to the corresponding value of the second source operand and writes the result of each addition into the corresponding quadword of the destination.

There are legacy and extended forms of the instruction:

ADDPD

Adds two pairs of values.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VADDPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Adds two pairs of values.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Adds four pairs of values.

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
ADDPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VADDPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ADDPD *xmm1, xmm2/mem128*	66 0F 58 /r	Adds two packed double-precision floating-point values in *xmm1* to corresponding values in *xmm2* or *mem128*. Writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VADDPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	58 /r
VADDPD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.01	58 /r

Related Instructions

(V)ADDPS, (V)ADDSD, (V)ADDSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

ADDPS Add
VADDPS Packed Single-Precision Floating-Point

Adds each packed single-precision floating-point value of the first source operand to the corresponding value of the second source operand and writes the result of each addition into the corresponding elements of the destination.

There are legacy and extended forms of the instruction:

ADDPS

Adds four pairs of values.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VADDPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Adds four pairs of values.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Adds eight pairs of values.

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
ADDPS	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VADDPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ADDPS *xmm1*, *xmm2/mem128*	0F 58 /r	Adds four packed single-precision floating-point values in *xmm1* to corresponding values in *xmm2* or *mem128*. Writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VADDPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.00	58 /r
VADDPS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.00	58 /r

Related Instructions

(V)ADDPD, (V)ADDSD, (V)ADDSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

ADDSD Add
VADDSD Scalar Double-Precision Floating-Point

Adds the double-precision floating-point value in the low-order quadword of the first source operand to the corresponding value in the low-order quadword of the second source operand and writes the result into the low-order quadword of the destination.

There are legacy and extended forms of the instruction:

ADDSD

The first source operand is an XMM register and the second source operand is either an XMM register or a 64-bit memory location. The first source register is also the destination register. Bits [127:64] of the destination and bits [255:128] of the corresponding YMM register are not affected.

VADDSD

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 64-bit memory location. The destination is a third XMM register. Bits [127:64] of the first source operand are copied to bits [127:64] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
ADDSD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VADDSD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ADDSD *xmm1*, *xmm2/mem64*	F2 0F 58 /r	Adds low-order double-precision floating-point values in *xmm1* to corresponding values in *xmm2* or *mem64*. Writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VADDSD *xmm1*, *xmm2*, *xmm3*/*mem64*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.11	58 /r

Related Instructions

(V)ADDPD, (V)ADDPS, (V)ADDSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

ADDSS Add
VADDSS Scalar Single-Precision Floating-Point

Adds the single-precision floating-point value in the low-order doubleword of the first source operand to the corresponding value in the low-order doubleword of the second source operand and writes the result into the low-order doubleword of the destination.

There are legacy and extended forms of the instruction:

ADDSS

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location. The first source register is also the destination. Bits [127:32] of the destination register and bits [255:128] of the corresponding YMM register are not affected.

VADDSS

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location. The destination is a third XMM register. Bits [127:32] of the first source register are copied to bits [127:32] of the of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
ADDSS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VADDSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ADDSS *xmm1*, *xmm2*/*mem32*	F3 0F 58 /r	Adds a single-precision floating-point value in the low-order doubleword of *xmm1* to a corresponding value in *xmm2* or *mem32*. Writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VADDSS *xmm1*, *xmm2*, *xmm3*/*mem32*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{\textit{src}}$.X.10	58 /r

Related Instructions

(V)ADDPD, (V)ADDPS, (V)ADDSD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

ADDSUBPD Alternating Addition and Subtraction
VADDSUBPD Packed Double-Precision Floating-Point

Adds the odd-numbered packed double-precision floating-point values of the first source operand to the corresponding values of the second source operand and writes the sum to the corresponding odd-numbered element of the destination; subtracts the even-numbered packed double-precision floating-point values of the second source operand from the corresponding values of the first source operand and writes the differences to the corresponding even-numbered element of the destination.

There are legacy and extended forms of the instruction:

ADDSUBPD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VADDSUBPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
ADDSUBPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VADDSUBPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ADDSUBPD *xmm1*, *xmm2/mem128*	66 0F D0 /r	Adds a value in the upper 64 bits of *xmm1* to the corresponding value in *xmm2* and writes the result to the upper 64 bits of *xmm1*; subtracts the value in the lower 64 bits of *xmm1* from the corresponding value in *xmm2* and writes the result to the lower 64 bits of *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VADDSUBPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	D0 /r
VADDSUBPD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.01	D0 /r

Related Instructions

(V)ADDSUBPS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

ADDSUBPS VADDSUBPS — Alternating Addition and Subtraction Packed Single-Precision Floating Point

Adds the second and fourth single-precision floating-point values of the first source operand to the corresponding values of the second source operand and writes the sums to the second and fourth elements of the destination. Subtracts the first and third single-precision floating-point values of the second source operand from the corresponding values of the first source operand and writes the differences to the first and third elements of the destination.

There are legacy and extended forms of the instruction:

ADDSUBPS

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VADDSUBPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
ADDSUBPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VADDSUBPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ADDSUBPS *xmm1*, *xmm2/mem128*	F2 0F D0 /r	Adds the second and fourth packed single-precision values in *xmm2* or *mem128* to the corresponding values in *xmm1* and writes results to the corresponding positions of *xmm1*. Subtracts the first and third packed single-precision values in *xmm2* or *mem128* from the corresponding values in *xmm1* and writes results to the corresponding positions of *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VADDSUBPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.11	D0 /r
VADDSUBPS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.11	D0 /r

Related Instructions

(V)ADDSUBPD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

AESDEC AES
VAESDEC Decryption Round

Performs a single round of AES decryption. Transforms a state value specified by the first source operand using a round key value specified by the second source operand, and writes the result to the destination.

See Appendix A on page 975 for more information about the operation of the AES instructions.

Decryption consists of 1, …, $N_r - 1$ iterations of sequences of operations called rounds, terminated by a unique final round, N_r. The AESDEC and VAESDEC instructions perform all the rounds except the last; the AESDECLAST and VAESDECLAST instructions perform the final round.

The 128-bit state and round key vectors are interpreted as 16-byte column-major entries in a 4-by-4 matrix of bytes.The transformed state is written to the destination in column-major order. For both instructions, the destination register is the same as the first source register.

There are legacy and extended forms of the instruction:

AESDEC

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VAESDEC

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
AESDEC	AES	CPUID Fn0000_0001_ECX[AES] (bit 25)
VAESDEC 128	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VAESDEC 256	VAES	CPUID Fn0000_0007_ECX[VAES]_x0 (bit 9)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
AESDEC *xmm1*, *xmm2/mem128*	66 0F 38 DE /r	Performs one decryption round on a state value in *xmm1* using the key value in *xmm2* or *mem128*. Writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	**RXB.map_select**	**W.vvvv.L.pp**	**Opcode**
VAESDEC *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00010	X.$\overline{src}$.0.01	DE /r
VAESDEC *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00010	X.$\overline{src}$.1.01	DE /r

Related Instructions

(V)AESENC, (V)AESENCLAST, (V)AESIMC, (V)AESKEYGENASSIST

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	**Virt**	**Prot**	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

AESDECLAST AES
VAESDECLAST Last Decryption Round

Performs the final round of AES decryption. Completes transformation of a state value specified by the first source operand using a round key value specified by the second source operand, and writes the result to the destination.

See Appendix A on page 975 for more information about the operation of the AES instructions.

Decryption consists of 1, …, $N_r - 1$ iterations of sequences of operations called rounds, terminated by a unique final round, N_r.The AESDEC and VAESDEC instructions perform all the rounds before the final round; the AESDECLAST and VAESDECLAST instructions perform the final round.

The 128-bit state and round key vectors are interpreted as 16-byte column-major entries in a 4-by-4 matrix of bytes.The transformed state is written to the destination in column-major order. For both instructions, the destination register is the same as the first source register.

There are legacy and extended forms of the instruction:

AESDECLAST

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VAESDECLAST

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
AESDECLAST	AES	CPUID Fn0000_0001_ECX[AES] (bit 25)
VAESDECLAST 128	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VAESDECLAST 256	VAES	CPUID Fn0000_0007_ECX[VAES]_x0 (bit 9)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
AESDECLAST *xmm1*, *xmm2/mem128*	66 0F 38 DF/r	Performs the last decryption round on a state value in *xmm1* using the key value in *xmm2* or *mem128*. Writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode

VAESDECLAST *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00010	X.$\overline{src}$.0.01	DF /r
VAESDECLAST *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00010	X.$\overline{src}$.1.01	DF /r

Related Instructions

(V)AESENC, (V)AESENCLAST, (V)AESIMC, (V)AESKEYGENASSIST

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

AESENC AES
VAESENC Encryption Round

Performs a single round of AES encryption. Transforms a state value specified by the first source operand using a round key value specified by the second source operand, and writes the result to the destination.

See Appendix A on page 975 for more information about the operation of the AES instructions.

Encryption consists of 1, …, $N_r - 1$ iterations of sequences of operations called rounds, terminated by a unique final round, N_r. The AESENC and VAESENC instructions perform all the rounds before the final round; the AESENCLAST and VAESENCLAST instructions perform the final round.

The 128-bit state and round key vectors are interpreted as 16-byte column-major entries in a 4-by-4 matrix of bytes.The transformed state is written to the destination in column-major order. For both instructions, the destination register is the same as the first source register

There are legacy and extended forms of the instruction:

AESENC

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VAESENC

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
AESENC	AES	CPUID Fn0000_0001_ECX[AES] (bit 25)
VAESENC 128	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VAESENC 256	VAES	CPUID Fn0000_0007_ECX[VAES]_x0 (bit 9)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
AESENC *xmm1*, *xmm2/mem128*	66 0F 38 DC /r	Performs one encryption round on a state value in *xmm1* using the key value in *xmm2* or *mem128*. Writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode

VAESENC *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00010	X.$\overline{src}$.0.01	DC /r
VAESENC *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00010	X.$\overline{src}$.1.01	DC /r

Related Instructions

(V)AESDEC, (V)AESDECLAST, (V)AESIMC, (V)AESKEYGENASSIST

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

AESENCLAST AES
VAESENCLAST Last Encryption Round

Performs the final round of AES encryption. Completes transformation of a state value specified by the first source operand using a round key value specified by the second source operand, and writes the result to the destination.

See Appendix A on page 975 for more information about the operation of the AES instructions.

Encryption consists of 1, …, $N_r - 1$ iterations of sequences of operations called rounds, terminated by a unique final round, N_r. The AESENC and VAESENC instructions perform all the rounds before the final round; the AESENCLAST and VAESENCLAST instructions perform the final round.

The 128-bit state and round key vectors are interpreted as 16-byte column-major entries in a 4-by-4 matrix of bytes.The transformed state is written to the destination in column-major order. For both instructions, the destination register is the same as the first source register.

There are legacy and extended forms of the instruction:

AESENCLAST

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VAESENCLAST

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
AESENCLAST	AES	CPUID Fn0000_0001_ECX[AES] (bit 25)
VAESENCLAST 128	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VAESENCLAST 256	VAES	CPUID Fn0000_0007_ECX[VAES]_x0 (bit 9)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
AESENCLAST *xmm1*, *xmm2/mem128*	66 0F 38 DD /r	Performs the last encryption round on a state value in *xmm1* using the key value in *xmm2* or *mem128*. Writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode

VAESENCLAST *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00010	X.$\overline{src}$.0.01	DD /r
VAESENCLAST *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00010	X.$\overline{src}$.1.01	DD /r

Related Instructions

(V)AESDEC, (V)AESDECLAST, (V)AESIMC, (V)AESKEYGENASSIST

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

AESIMC AES
VAESIMC InvMixColumn Transformation

Applies the AES *InvMixColumns*( ) transformation to expanded round keys in preparation for decryption. Transforms an expanded key specified by the second source operand and writes the result to a destination register.

See Appendix A on page 975 for more information about the operation of the AES instructions.

The 128-bit round key vector is interpreted as 16-byte column-major entries in a 4-by-4 matrix of bytes.The transformed result is written to the destination in column-major order.

AESIMC and VAESIMC are not used to transform the first and last round key in a decryption sequence.

There are legacy and extended forms of the instruction:

AESIMC

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VAESIMC

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
AESIMC	AES	CPUID Fn0000_0001_ECX[AES] (bit 25)
VAESIMC	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
AESIMC *xmm1*, *xmm2/mem128*	66 0F 38 DB /r	Performs AES InvMixColumn transformation on a round key in the *xmm2* or *mem128* and stores the result in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VAESIMC *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00010	X.$\overline{src}$.0.01	DB /r

Related Instructions

(V)AESDEC, (V)AESDECLAST, (V)AESENC, (V)AESENCLAST, (V)AESKEYGENASSIST

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

AESKEYGENASSIST VAESKEYGENASSIST AES Assist Round Key Generation

Expands a round key for encryption. Transforms a 128-bit round key operand using an 8-bit round constant and writes the result to a destination register.

See Appendix A on page 975 for more information about the operation of the AES instructions.

The round key is provided by the second source operand and the round constant is specified by an immediate operand. The 128-bit round key vector is interpreted as 16-byte column-major entries in a 4-by-4 matrix of bytes. The transformed result is written to the destination in column-major order.

There are legacy and extended forms of the instruction:

AESKEYGENASSIST

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VAESKEYGENASSIST

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
AESKEYGENASSIST	AES	CPUID Fn0000_0001_ECX[AES] (bit 25)
VAESKEYGENASSIST	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
AESKEYGENASSIST *xmm1*, *xmm2/mem128, imm8*	66 0F 3A DF /r ib	Expands a round key in *xmm2* or *mem128* using an immediate round constant. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
AESKEYGENASSIST *xmm1*, *xmm2 /mem128, imm8*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.0.01	DF /r ib

Related Instructions

(V)AESDEC, (V)AESDECLAST, (V)AESENC, (V)AESENCLAST,(V)AESIMC

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

ANDNPD AND NOT
VANDNPD Packed Double-Precision Floating-Point

Performs a bitwise AND of two packed double-precision floating-point values in the second source operand with the ones'-complement of the two corresponding packed double-precision floating-point values in the first source operand and writes the result into the destination.

There are legacy and extended forms of the instruction:

ANDNPD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VANDNPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
ANDNPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VANDNPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ANDNPD *xmm1*, *xmm2/mem128*	66 0F 55 /r	Performs bitwise AND of two packed double-precision floating-point values in *xmm2* or *mem128* with the ones'-complement of two packed double-precision floating-point values in *xmm1*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VANDNPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	55 /r
VANDNPD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.01	55 /r

Related Instructions

(V)ANDNPS, (V)ANDPD, (V)ANDPS, (V)ORPD, (V)ORPS, (V)XORPD, (V)XORPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

ANDNPS AND NOT
VANDNPS Packed Single-Precision Floating-Point

Performs a bitwise AND of four packed single-precision floating-point values in the second source operand with the ones'-complement of the four corresponding packed single-precision floating-point values in the first source operand, and writes the result in the destination.

There are legacy and extended forms of the instruction:

ANDNPS

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VANDNPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
ANDNPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VANDNPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ANDNPS *xmm1*, *xmm2/mem128*	0F 55 /r	Performs bitwise AND of four packed single-precision floating-point values in *xmm2* or *mem128* with the ones'-complement of four packed single-precision floating-point values in *xmm1*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VANDNPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.00	55 /r
VANDNPS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.00	55 /r

Related Instructions

(V)ANDNPD, (V)ANDPD, (V)ANDPS, (V)ORPD, (V)ORPS, (V)XORPD, (V)XORPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

ANDPD AND
VANDPD Packed Double-Precision Floating-Point

Performs bitwise AND of two packed double-precision floating-point values in the first source operand with the corresponding two packed double-precision floating-point values in the second source operand and writes the results into the corresponding elements of the destination.

There are legacy and extended forms of the instruction:

ANDPD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VANDPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
ANDPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VANDPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ANDPD *xmm1*, *xmm2/mem128*	66 0F 54 /r	Performs bitwise AND of two packed double-precision floating-point values in *xmm1* with corresponding values in *xmm2* or *mem128*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VANDPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	54 /r
VANDPD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.01	54 /r

Related Instructions

(V)ANDNPD, (V)ANDNPS, (V)ANDPS, (V)ORPD, (V)ORPS, (V)XORPD, (V)XORPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

ANDPS AND
VANDPS Packed Single-Precision Floating-Point

Performs bitwise AND of the four packed single-precision floating-point values in the first source operand with the corresponding four packed single-precision floating-point values in the second source operand, and writes the result into the corresponding elements of the destination.

There are legacy and extended forms of the instruction:

ANDPS

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VANDPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
ANDPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VANDPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ANDPS *xmm1*, *xmm2/mem128*	0F 54 /r	Performs bitwise AND of four packed single-precision floating-point values in *xmm1* with corresponding values in *xmm2* or *mem128*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VANDPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.00	54 /r
VANDPS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.00	54 /r

Related Instructions

(V)ANDNPD, (V)ANDNPS, (V)ANDPD, (V)ORPD, (V)ORPS, (V)XORPD, (V)XORPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

BLENDPD Blend
VBLENDPD Packed Double-Precision Floating-Point

Copies packed double-precision floating-point values from either of two sources to a destination, as specified by an 8-bit mask operand.

Each mask bit specifies a 64-bit element in a source location and a corresponding 64-bit element in the destination register. When a mask bit = 0, the specified element of the first source is copied to the corresponding position in the destination register. When a mask bit = 1, the specified element of the second source is copied to the corresponding position in the destination register.

There are legacy and extended forms of the instruction:

BLENDPD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected. Only mask bits [1:0] are used.

VBLENDPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared. Only mask bits [1:0] are used.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register. Only mask bits [3:0] are used.

Instruction Support

Form	Subset	Feature Flag
BLENDPD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VBLENDPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
BLENDPD *xmm1*, *xmm2/mem128, imm8*	66 0F 3A 0D /r ib	Copies values from *xmm1* or *xmm2*/*mem128* to *xmm1*, as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VBLENDPD *xmm1*, *xmm2*, *xmm3/mem128*, *imm8*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.0.01	0D /r ib
VBLENDPD *ymm1, ymm2, ymm3/mem256*, *imm8*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.1.01	0D /r ib

Related Instructions

(V)BLENDPS, (B)BLENDVPD, (V)BLENDVPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

BLENDPS Blend
VBLENDPS Packed Single-Precision Floating-Point

Copies packed single-precision floating-point values from either of two sources to a destination, as specified by an 8-bit mask operand.

Each mask bit specifies a 32-bit element in a source location and a corresponding 32-bit element in the destination register. When a mask bit = 0, the specified element of the first source is copied to the corresponding position in the destination register. When a mask bit = 1, the specified element of the second source is copied to the corresponding position in the destination register.

There are legacy and extended forms of the instruction:

BLENDPS

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected. Only mask bits [3:0] are used.

VBLENDPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.Only mask bits [3:0] are used.

YMM Encoding

The first operand is a YMM register and the second operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register. All 8 bits of the mask are used.

Instruction Support

Form	Subset	Feature Flag
BLENDPS	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VBLENDPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
BLENDPS *xmm1*, *xmm2/mem128, imm8*	66 0F 3A 0C /r ib	Copies values from *xmm1* or *xmm2*/*mem128* to *xmm1*, as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VBLENDPS *xmm1*, *xmm2*, *xmm3/mem128, imm8*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.0.01	0C /r ib
VBLENDPS *ymm1, ymm2, ymm3/mem256, imm8*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.1.01	0C /r ib

Related Instructions

(V)BLENDPD, (V)BLENDVPD, (V)BLENDVPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

BLENDVPD Variable Blend
VBLENDVPD Packed Double-Precision Floating-Point

Copies packed double-precision floating-point values from either of two sources to a destination, as specified by a mask operand.

Each mask bit specifies a 64-bit element of a source location and a corresponding 64-bit element of the destination. The position of a mask bit corresponds to the position of the most significant bit of a copied value. When a mask bit = 0, the specified element of the first source is copied to the corresponding position in the destination. When a mask bit = 1, the specified element of the second source is copied to the corresponding position in the destination.

There are legacy and extended forms of the instruction:

BLENDVPD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected. The mask is defined by bits 127 and 63 of the implicit register XMM0.

VBLENDVPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared. The mask is defined by bits 127 and 63 of a fourth XMM register.

YMM Encoding

The first operand is a YMM register and the second operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register. The mask is defined by bits 255, 191, 127, and 63 of a fourth YMM register.

Instruction Support

Form	Subset	Feature Flag
BLENDVPD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VBLENDVPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
BLENDVPD *xmm1*, *xmm2/mem128*	66 0F 38 15 /r	Copies values from *xmm1* or *xmm2*/*mem128* to *xmm1*, as specified by the MSB of corresponding elements of xmm0.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VBLENDVPD *xmm1*, *xmm2*, *xmm3/mem128*, *xmm4*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.0.01	4B /r
VBLENDVPD *ymm1*, *ymm2*, *ymm3/mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.1.01	4B /r

Related Instructions

(V)BLENDPD, (V)BLENDPS, (V)BLENDVPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

BLENDVPS Variable Blend
VBLENDVPS Packed Single-Precision Floating-Point

Copies packed single-precision floating-point values from either of two sources to a destination, as specified by a mask operand.

Each mask bit specifies a 32-bit element of a source location and a corresponding 32-bit element of the destination register. The position of a mask bits corresponds to the position of the most significant bit of a copied value. When a mask bit = 0, the specified element of the first source is copied to the corresponding position in the destination. When a mask bit = 1, the specified element of the second source is copied to the corresponding position in the destination.

There are legacy and extended forms of the instruction:

BLENDVPS

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected. The mask is defined by bits 127, 95, 63, and 31 of the implicit register XMM0.

VBLENDVPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared. The mask is defined by bits 127, 95, 63, and 31 of a fourth XMM register.

YMM Encoding

The first operand is a YMM register and the second operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register. The mask is defined by bits 255, 223, 191, 159, 127, 95, 63, and 31 of a fourth YMM register.

Instruction Support

Form	Subset	Feature Flag
BLENDVPS	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VBLENDVPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
BLENDVPS *xmm1*, *xmm2/mem128*	66 0F 38 14 /r	Copies packed single-precision floating-point values from *xmm1* or *xmm2/mem128* to *xmm1*, as specified by bits in xmm0.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VBLENDVPS *xmm1*, *xmm2*, *xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.0.01	4A /r
VBLENDVPS *ymm1, ymm2, ymm3/mem256, ymm4*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.1.01	4A /r

Related Instructions

(V)BLENDPD, (V)BLENDPS, (V)BLENDVPD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

CMPPD Compare
VCMPPD Packed Double-Precision Floating-Point

Compares each of the two packed double-precision floating-point values of the first source operand to the corresponding values of the second source operand and writes the result of each comparison to the corresponding 64-bit element of the destination. When a comparison is TRUE, all 64 bits of the destination element are set; when a comparison is FALSE, all 64 bits of the destination element are cleared. The type of comparison is specified by an immediate byte operand.

Signed comparisons return TRUE only when both operands are valid numbers and the numbers have the relation specified by the type of comparison operation. Ordered comparison returns TRUE when both operands are valid numbers, or FALSE when either operand is a NaN. Unordered comparison returns TRUE only when one or both operands are NaN and FALSE otherwise.

QNaN operands generate an Invalid Operation Exception (IE) only if the comparison type isn't Equal, Unequal, Ordered, or Unordered. SNaN operands always generate an IE.

There are legacy and extended forms of the instruction:

CMPPD

The first source operand is an XMM register and the second source operand is either an XMM register or a128-bit memory location.The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected. Comparison type is specified by bits [2:0] of an immediate byte operand.

VCMPPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared. Comparison type is specified by bits [4:0] of an immediate byte operand.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination operand is a YMM register. Comparison type is specified by bits [4:0] of an immediate byte operand.

Immediate Operand Encoding

CMPPD uses bits [2:0] of the 8-bit immediate operand and VCMPPD uses bits [4:0] of the 8-bit immediate operand. Although VCMPPD supports 20h encoding values, the comparison types echo those of CMPPD on 4-bit boundaries. The following table shows the immediate operand value for CMPPD and each of the VCMPPD echoes.

Some comparison operations that are not directly supported by immediate-byte encodings can be implemented by swapping the contents of the source and destination operands and executing the appropriate comparison of the swapped values. These additional comparison operations are shown with the directly supported comparison operations.

Immediate Operand Value	Compare Operation	Result If NaN Operand	QNaN Operand Causes Invalid Operation Exception
00h, 08h, 10h, 18h	Equal	FALSE	No
01h, 09h, 11h, 19h	Less than	FALSE	Yes
	Greater than (swapped operands)	FALSE	Yes
02h, 0Ah, 12h, 1Ah	Less than or equal	FALSE	Yes
	Greater than or equal (swapped operands)	FALSE	Yes
03h, 0Bh, 13h, 1Bh	Unordered	TRUE	No
04h, 0Ch, 14h, 1Ch	Not equal	TRUE	No
05h, 0Dh, 15h, 1Dh	Not less than	TRUE	Yes
	Not greater than (swapped operands)	TRUE	Yes
06h, 0Eh, 16h, 1Eh	Not less than or equal	TRUE	Yes
	Not greater than or equal (swapped operands)	TRUE	Yes
07h, 0Fh, 17h, 1Fh	Ordered	FALSE	No

The following alias mnemonics for (V)CMPPD with appropriate value of *imm8* are supported.

Mnemonic	Implied Value of *imm8*
(V)CMPEQPD	00h, 08h, 10h, 18h
(V)CMPLTPD	01h, 09h, 11h, 19h
(V)CMPLEPD	02h, 0Ah, 12h, 1Ah
(V)CMPUNORDPD	03h, 0Bh, 13h, 1Bh
(V)CMPNEQPD	04h, 0Ch, 14h, 1Ch
(V)CMPNLTPD	05h, 0Dh, 15h, 1Dh
(V)CMPNLEPD	06h, 0Eh, 16h, 1Eh
(V)CMPORDPD	07h, 0Fh, 17h, 1Fh

Instruction Support

Form	Subset	Feature Flag
CMPPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCMPPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CMPPD *xmm1*, *xmm2/mem128*, *imm8*	66 0F C2 /r ib	Compares two pairs of values in *xmm1* to corresponding values in *xmm2* or *mem128*. Comparison type is determined by *imm8*. Writes comparison results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCMPPD *xmm1*, *xmm2*, *xmm3/mem128*, *imm8*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	C2 /r ib
VCMPPD *ymm1, ymm2, ymm3/mem256*, *imm8*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.01	C2 /r ib

Related Instructions

(V)CMPPS, (V)CMPSD, (V)CMPSS, (V)COMISD, (V)COMISS, (V)UCOMISD, (V)UCOMISS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

CMPPS Compare
VCMPPS Packed Single-Precision Floating-Point

Compares each of the four packed single-precision floating-point values of the first source operand to the corresponding values of the second source operand and writes the result of each comparison to the corresponding 32-bit element of the destination. When a comparison is TRUE, all 32 bits of the destination element are set; when a comparison is FALSE, all 32 bits of the destination element are cleared. The type of comparison is specified by an immediate byte operand.

Signed comparisons return TRUE only when both operands are valid numbers and the numbers have the relation specified by the type of comparison operation. Ordered comparison returns TRUE when both operands are valid numbers, or FALSE when either operand is a NaN. Unordered comparison returns TRUE only when one or both operands are NaN and FALSE otherwise.

QNaN operands generate an Invalid Operation Exception (IE) only if the comparison type isn't Equal, Unequal, Ordered, or Unordered. SNaN operands always generate an IE.

There are legacy and extended forms of the instruction:

CMPPS

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected. Comparison type is specified by bits [2:0] of an immediate byte operand.

VCMPPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared. Comparison type is specified by bits [4:0] of an immediate byte operand.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination operand is a YMM register. Comparison type is specified by bits [4:0] of an immediate byte operand.

Immediate Operand Encoding

CMPPS uses bits [2:0] of the 8-bit immediate operand and VCMPPS uses bits [4:0] of the 8-bit immediate operand. Although VCMPPS supports 20h encoding values, the comparison types echo those of CMPPS on 4-bit boundaries. The following table shows the immediate operand value for CMPPS and each of the VCMPPDS echoes.

Some comparison operations that are not directly supported by immediate-byte encodings can be implemented by swapping the contents of the source and destination operands and executing the appropriate comparison of the swapped values. These additional comparison operations are shown in with the directly supported comparison operations.

Immediate Operand Value	Compare Operation	Result If NaN Operand	QNaN Operand Causes Invalid Operation Exception
00h, 08h, 10h, 18h	Equal	FALSE	No
01h, 09h, 11h, 19h	Less than	FALSE	Yes
	Greater than (swapped operands)	FALSE	Yes
02h, 0Ah, 12h, 1Ah	Less than or equal	FALSE	Yes
	Greater than or equal (swapped operands)	FALSE	Yes
03h, 0Bh, 13h, 1Bh	Unordered	TRUE	No
04h, 0Ch, 14h, 1Ch	Not equal	TRUE	No
05h, 0Dh, 15h, 1Dh	Not less than	TRUE	Yes
	Not greater than (swapped operands)	TRUE	Yes
06h, 0Eh, 16h, 1Eh	Not less than or equal	TRUE	Yes
	Not greater than or equal (swapped operands)	TRUE	Yes
07h, 0Fh, 17h, 1Fh	Ordered	FALSE	No

The following alias mnemonics for (V)CMPPS with appropriate value of *imm8* are supported.

Mnemonic	Implied Value of *imm8*
(V)CMPEQPS	00h, 08h, 10h, 18h
(V)CMPLTPS	01h, 09h, 11h, 19h
(V)CMPLEPS	02h, 0Ah, 12h, 1Ah
(V)CMPUNORDPS	03h, 0Bh, 13h, 1Bh
(V)CMPNEQPS	04h, 0Ch, 14h, 1Ch
(V)CMPNLTPS	05h, 0Dh, 15h, 1Dh
(V)CMPNLEPS	06h, 0Eh, 16h, 1Eh
(V)CMPORDPS	07h, 0Fh, 17h, 1Fh

Instruction Support

Form	Subset	Feature Flag
CMPPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VCMPPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CMPPS *xmm1*, *xmm2/mem128*, *imm8*	0F C2 /r ib	Compares four pairs of values in *xmm1* to corresponding values in *xmm2* or *mem128*. Comparison type is determined by *imm8*. Writes comparison results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCMPPS *xmm1*, *xmm2*, *xmm3/mem128*, *imm8*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.00	C2 /r ib
VCMPPS *ymm1, ymm2, ymm3/mem256*, *imm8*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.00	C2 /r ib

Related Instructions

(V)CMPPD, (V)CMPSD, (V)CMPSS, (V)COMISD, (V)COMISS, (V)UCOMISD, (V)UCOMISS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

CMPSD Compare
VCMPSD Scalar Double-Precision Floating-Point

Compares a double-precision floating-point value in the low-order 64 bits of the first source operand with a double-precision floating-point value in the low-order 64 bits of the second source operand and writes the result to the low-order 64 bits of the destination. When a comparison is TRUE, all 64 bits of the destination element are set; when a comparison is FALSE, all 64 bits of the destination element are cleared. Comparison type is specified by an immediate byte operand.

Signed comparisons return TRUE only when both operands are valid numbers and the numbers have the relation specified by the type of comparison operation. Ordered comparison returns TRUE when both operands are valid numbers, or FALSE when either operand is a NaN. Unordered comparison returns TRUE only when one or both operands are NaN and FALSE otherwise.

QNaN operands generate an Invalid Operation Exception (IE) only when the comparison type is not Equal, Unequal, Ordered, or Unordered. SNaN operands always generate an IE.

There are legacy and extended forms of the instruction:

CMPSD

The first source operand is an XMM register. The second source operand is either an XMM register or a 64-bit memory location. The first source register is also the destination. Bits [127:64] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected. Comparison type is specified by bits [2:0] of an immediate byte operand.

This CMPSD instruction must not be confused with the same-mnemonic CMPSD (compare strings by doubleword) instruction in the general-purpose instruction set. Assemblers can distinguish the instructions by the number and type of operands.

VCMPSD

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register. The second source operand is either an XMM register or a 64-bit memory location. The destination is a third XMM register. Bits [127:64] of the destination are copied from bits [127:64] of the first source. Bits [255:128] of the YMM register that corresponds to the destination are cleared. Comparison type is specified by bits [4:0] of an immediate byte operand.

Immediate Operand Encoding

CMPSD uses bits [2:0] of the 8-bit immediate operand and VCMPSD uses bits [4:0] of the 8-bit immediate operand. Although VCMPSD supports 20h encoding values, the comparison types echo those of CMPSD on 4-bit boundaries. The following table shows the immediate operand value for CMPSD and each of the VCMPSD echoes.

Some comparison operations that are not directly supported by immediate-byte encodings can be implemented by swapping the contents of the source and destination operands and executing the appropriate comparison of the swapped values. These additional comparison operations are shown with the directly supported comparison operations. When operands are swapped, the first source XMM register is overwritten by the result.

Immediate Operand Value	Compare Operation	Result If NaN Operand	QNaN Operand Causes Invalid Operation Exception
00h, 08h, 10h, 18h	Equal	FALSE	No
01h, 09h, 11h, 19h	Less than	FALSE	Yes
	Greater than (swapped operands)	FALSE	Yes
02h, 0Ah, 12h, 1Ah	Less than or equal	FALSE	Yes
	Greater than or equal (swapped operands)	FALSE	Yes
03h, 0Bh, 13h, 1Bh	Unordered	TRUE	No
04h, 0Ch, 14h, 1Ch	Not equal	TRUE	No
05h, 0Dh, 15h, 1Dh	Not less than	TRUE	Yes
	Not greater than (swapped operands)	TRUE	Yes
06h, 0Eh, 16h, 1Eh	Not less than or equal	TRUE	Yes
	Not greater than or equal (swapped operands)	TRUE	Yes
07h, 0Fh, 17h, 1Fh	Ordered	FALSE	No

The following alias mnemonics for (V)CMPSD with appropriate value of *imm8* are supported.

Mnemonic	Implied Value of *imm8*
(V)CMPEQSD	00h, 08h, 10h, 18h
(V)CMPLTSD	01h, 09h, 11h, 19h
(V)CMPLESD	02h, 0Ah, 12h, 1Ah
(V)CMPUNORDSD	03h, 0Bh, 13h, 1Bh
(V)CMPNEQSD	04h, 0Ch, 14h, 1Ch
(V)CMPNLTSD	05h, 0Dh, 15h, 1Dh
(V)CMPNLESD	06h, 0Eh, 16h, 1Eh
(V)CMPORDSD	07h, 0Fh, 17h, 1Fh

Instruction Support

Form	Subset	Feature Flag
CMPSD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCMPSD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CMPSD *xmm1*, *xmm2/mem64*, *imm8*	F2 0F C2 /r ib	Compares double-precision floating-point values in the low-order 64 bits of *xmm1* with corresponding values in *xmm2* or *mem64*. Comparison type is determined by *imm8*. Writes comparison results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCMPSD *xmm1*, *xmm2*, *xmm3/mem64*, *imm8*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.11	C2 /r ib

Related Instructions

(V)CMPPD, (V)CMPPS, (V)CMPSS, (V)COMISD, (V)COMISS, (V)UCOMISD, (V)UCOMISS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

CMPSS Compare
VCMPSS Scalar Single-Precision Floating-Point

Compares a single-precision floating-point value in the low-order 32 bits of the first source operand with a single-precision floating-point value in the low-order 32 bits of the second source operand and writes the result to the low-order 32 bits of the destination. When a comparison is TRUE, all 32 bits of the destination element are set; when a comparison is FALSE, all 32 bits of the destination element are cleared. Comparison type is specified by an immediate byte operand.

Signed comparisons return TRUE only when both operands are valid numbers and the numbers have the relation specified by the type of comparison operation. Ordered comparison returns TRUE when both operands are valid numbers, or FALSE when either operand is a NaN. Unordered comparison returns TRUE only when one or both operands are NaN and FALSE otherwise.

QNaN operands generate an Invalid Operation Exception (IE) only if the comparison type isn't Equal, Unequal, Ordered, or Unordered. SNaN operands always generate an IE.

There are legacy and extended forms of the instruction:

CMPSS

The first source operand is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The first source register is also the destination. Bits [127:32] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected. Comparison type is specified by bits [2:0] of an immediate byte operand.

VCMPSS

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is a third XMM register. Bits [127:32] of the destination are copied from bits [127L32] of the first source. Bits [255:128] of the YMM register that corresponds to the destination are cleared. Comparison type is specified by bits [4:0] of an immediate byte operand.

Immediate Operand Encoding

CMPSS uses bits [2:0] of the 8-bit immediate operand and VCMPSS uses bits [4:0] of the 8-bit immediate operand. Although VCMPSS supports 20h encoding values, the comparison types echo those of CMPSS on 4-bit boundaries. The following table shows the immediate operand value for CMPSS and each of the VCMPSS echoes.

Some comparison operations that are not directly supported by immediate-byte encodings can be implemented by swapping the contents of the source and destination operands and executing the appropriate comparison of the swapped values. These additional comparison operations are shown below with the directly supported comparison operations. When operands are swapped, the first source XMM register is overwritten by the result.

Immediate Operand Value	Compare Operation	Result If NaN Operand	QNaN Operand Causes Invalid Operation Exception
00h, 08h, 10h, 18h	Equal	FALSE	No
01h, 09h, 11h, 19h	Less than	FALSE	Yes
	Greater than (swapped operands)	FALSE	Yes
02h, 0Ah, 12h, 1Ah	Less than or equal	FALSE	Yes
	Greater than or equal (swapped operands)	FALSE	Yes
03h, 0Bh, 13h, 1Bh	Unordered	TRUE	No
04h, 0Ch, 14h, 1Ch	Not equal	TRUE	No
05h, 0Dh, 15h, 1Dh	Not less than	TRUE	Yes
	Not greater than (swapped operands)	TRUE	Yes
06h, 0Eh, 16h, 1Eh	Not less than or equal	TRUE	Yes
	Not greater than or equal (swapped operands)	TRUE	Yes
07h, 0Fh, 17h, 1Fh	Ordered	FALSE	No

The following alias mnemonics for (V)CMPSS with appropriate value of *imm8* are supported.

Mnemonic	Implied Value of *imm8*
(V)CMPEQSS	00h, 08h, 10h, 18h
(V)CMPLTSS	01h, 09h, 11h, 19h
(V)CMPLESS	02h, 0Ah, 12h, 1Ah
(V)CMPUNORDSS	03h, 0Bh, 13h, 1Bh
(V)CMPNEQSS	04h, 0Ch, 14h, 1Ch
(V)CMPNLTSS	05h, 0Dh, 15h, 1Dh
(V)CMPNLESS	06h, 0Eh, 16h, 1Eh
(V)CMPORDSS	07h, 0Fh, 17h, 1Fh

Instruction Support

Form	Subset	Feature Flag
CMPSS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VCMPSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CMPSS *xmm1*, *xmm2*/*mem32*, *imm8*	F3 0F C2 /r ib	Compares single-precision floating-point values in the low-order 32 bits of *xmm1* with corresponding values in *xmm2* or *mem32*. Comparison type is determined by *imm8*. Writes comparison results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCMPSS *xmm1*, *xmm2*, *xmm3/mem32*, *imm8*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.10	C2 /r ib

Related Instructions

(V)CMPPD, (V)CMPPS, (V)CMPSD, (V)COMISD, (V)COMISS, (V)UCOMISD, (V)UCOMISS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

COMISD Compare Ordered
VCOMISD Scalar Double-Precision Floating-Point

Compares a double-precision floating-point value in the low-order 64 bits of the first operand with a double-precision floating-point value in the low-order 64 bits of the second operand and sets rFLAGS.ZF, PF, and CF to show the result of the comparison:

Comparison	ZF	PF	CF
NaN input	1	1	1
operand 1 > operand 2	0	0	0
operand 1 < operand 2	0	0	1
operand 1 == operand 2	1	0	0

The result is unordered if one or both of the operand values is a NaN. The rFLAGS.OF, AF, and SF bits are cleared. If an #XF SIMD floating-point exception occurs the rFLAGS bits are not updated.

There are legacy and extended forms of the instruction:

COMISD

The first source operand is an XMM register and the second source operand is an XMM register or a 64-bit memory location.

VCOMISD

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 64-bit memory location.

Instruction Support

Form	Subset	Feature Flag
COMISD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCOMISD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
COMISD *xmm1*, *xmm2/mem64*	66 0F 2F /r	Compares double-precision floating-point values in *xmm1* with corresponding values in *xmm2* or *mem64* and sets rFLAGS.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCOMISD *xmm1*, *xmm2 /mem64*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.01	2F /r

Related Instructions

(V)CMPPD, (V)CMPPS, (V)CMPSD, (V)CMPSS, (V)COMISS, (V)UCOMISD, (V)UCOMISS

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL		OF	DF	IF	TF	SF	ZF	AF	PF	CF
									0				0	M	0	M	M
21	20	19	18	17	16	14	13	12	11	10	9	8	7	6	4	2	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected. Bits 31:22, 15, 5, 3, and 1 are reserved. For #XF, rFLAGS bits are not updated.*

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Unaffected flags are blank.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

COMISS Compare
VCOMISS Ordered Scalar Single-Precision Floating-Point

Compares a double-precision floating-point value in the low-order 32 bits of the first operand with a single-precision floating-point value in the low-order 32 bits of the second operand and sets rFLAGS.ZF, PF, and CF to show the result of the comparison:

Comparison	ZF	PF	CF
NaN input	1	1	1
operand 1 > operand 2	0	0	0
operand 1 < operand 2	0	0	1
operand 1 == operand 2	1	0	0

The result is unordered if one or both of the operand values is a NaN. The rFLAGS.OF, AF, and SF bits are cleared. If an #XF SIMD floating-point exception occurs the rFLAGS bits are not updated.

There are legacy and extended forms of the instruction:

COMISS

The first source operand is an XMM register and the second source operand is an XMM register or a 32-bit memory location.

VCOMISS

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location.

Instruction Support

Form	Subset	Feature Flag
COMISS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VCOMISS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
COMISS *xmm1*, *xmm2*/*mem32*	0F 2F /r	Compares single-precision floating-point values in *xmm1* with corresponding values in *xmm2* or *mem32* and sets rFLAGS.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCOMISS *xmm1*, *xmm2* /*mem32*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.00	2F /r

Related Instructions

(V)CMPPD, (V)CMPPS, (V)CMPSD, (V)CMPSS, (V)COMISD, (V)UCOMISD, (V)UCOMISS

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL		OF	DF	IF	TF	SF	ZF	AF	PF	CF
									0				0	M	0	M	M
21	20	19	18	17	16	14	13	12	11	10	9	8	7	6	4	2	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected. Bits 31:22, 15, 5, 3, and 1 are reserved. For #XF, rFLAGS bits are not updated.*

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

CVTDQ2PD VCVTDQ2PD Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point

Converts packed 32-bit signed integer values to packed double-precision floating-point values and writes the converted values to the destination.

There are legacy and extended forms of the instruction:

CVTDQ2PD

Converts two packed 32-bit signed integer values in the low-order 64 bits of an XMM register or in a 64-bit memory location to two packed double-precision floating-point values and writes the converted values to an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VCVTDQ2PD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Converts two packed 32-bit signed integer values in the low-order 64 bits of an XMM register or in a 64-bit memory location to two packed double-precision floating-point values and writes the converted values to an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Converts four packed 32-bit signed integer values in the low-order 128 bits of a YMM register or a 256-bit memory location to four packed double-precision floating-point values and writes the converted values to a YMM register.

Instruction Support

Form	Subset	Feature Flag
CVTDQ2PD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTDQ2PD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTDQ2PD *xmm1*, *xmm2/mem64*	F3 0F E6 /r	Converts packed doubleword signed integers in *xmm2* or *mem64* to double-precision floating-point values in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTDQ2PD *xmm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.10	E6 /r
VCVTDQ2PD *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.10	E6 /r

Related Instructions

(V)CVTPD2DQ, (V)CVTPI2PD, (V)CVTSD2SI, (V)CVTSI2SD, (V)CVTTPD2DQ, (V)CVTT-SD2SI

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference with alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

CVTDQ2PS VCVTDQ2PS Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point

Converts packed 32-bit signed integer values to packed single-precision floating-point values and writes the converted values to the destination. When the result is an inexact value, it is rounded as specified by MXCSR.RC.

There are legacy and extended forms of the instruction:

CVTDQ2PS

Converts four packed 32-bit signed integer values in an XMM register or a 128-bit memory location to four packed single-precision floating-point values and writes the converted values to an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VCVTDQ2PS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Converts four packed 32-bit signed integer values in an XMM register or a 128-bit memory location to four packed single-precision floating-point values and writes the converted values to an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Converts eight packed 32-bit signed integer values in a YMM register or a 256-bit memory location to eight packed single-precision floating-point values and writes the converted values to a YMM register.

Instruction Support

Form	Subset	Feature Flag
CVTDQ2PS	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTDQ2PS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTDQ2PS *xmm1*, *xmm2/mem128*	0F 5B /r	Converts packed doubleword integer values in *xmm2* or *mem128* to packed single-precision floating-point values in *xmm2*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTDQ2PS *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.00	5B /r
VCVTDQ2PS *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.00	5B /r

Related Instructions

(V)CVTPS2DQ, (V)CVTSI2SS, (V)CVTSS2SI, (V)CVTTPS2DQ, (V)CVTTSS2SI

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

CVTPD2DQ VCVTPD2DQ Convert Packed Double-Precision Floating-Point to Packed Doubleword Integer

Converts packed double-precision floating-point values to packed signed doubleword integers and writes the converted values to the destination.

When the result is an inexact value, it is rounded as specified by MXCSR.RC. When the floating-point value is a NaN, infinity, or the result of the conversion is larger than the maximum signed doubleword (-2^{31} to $+2^{31} - 1$), the instruction returns the 32-bit indefinite integer value (8000_0000h) when the invalid-operation exception (IE) is masked.

There are legacy and extended forms of the instruction:

CVTPD2DQ

Converts two packed double-precision floating-point values in an XMM register or a 128-bit memory location to two packed signed doubleword integers and writes the converted values to the two low-order doublewords of the destination XMM register. Bits [127:64] of the destination are cleared. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VCVTPD2DQ

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Converts two packed double-precision floating-point values in an XMM register or a 128-bit memory location to two signed doubleword values and writes the converted values to the lower two doubleword elements of the destination XMM register. Bits [127:64] of the destination are cleared. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Converts four packed double-precision floating-point values in a YMM register or a 256-bit memory location to four signed doubleword values and writes the converted values to an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
CVTPD2DQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTPD2DQ	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTPD2DQ *xmm1*, *xmm2/mem128*	F2 0F E6 /r	Converts two packed double-precision floating-point values in *xmm2* or *mem128* to packed doubleword integers in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTPD2DQ *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.11	E6 /r
VCVTPD2DQ *xmm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.11	E6 /r

Related Instructions

(V)CVTDQ2PD, (V)CVTPI2PD, (V)CVTSD2SI, (V)CVTSI2SD, (V)CVTTPD2DQ, (V)CVTT-SD2SI

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

CVTPD2PS Convert Packed Double-Precision Floating-Point
VCVTPD2PS to Packed Single-Precision Floating-Point

Converts packed double-precision floating-point values to packed single-precision floating-point values and writes the converted values to the low-order doubleword elements of the destination. When the result is an inexact value, it is rounded as specified by MXCSR.RC.

There are legacy and extended forms of the instruction:

CVTPD2PS

Converts two packed double-precision floating-point values in an XMM register or a 128-bit memory location to two packed single-precision floating-point values and writes the converted values to an XMM register. Bits [127:64] of the destination are cleared. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VCVTPD2PS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Converts two packed double-precision floating-point values in an XMM register or a 128-bit memory location to two packed single-precision floating-point values and writes the converted values to an XMM register. Bits [127:64] of the destination are cleared. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Converts four packed double-precision floating-point values in a YMM register or a 256-bit memory location to four packed single-precision floating-point values and writes the converted values to a YMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
CVTPD2PS	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTPD2PS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTPD2PS *xmm1*, *xmm2/mem128*	66 0F 5A /r	Converts packed double-precision floating-point values in *xmm2* or *mem128* to packed single-precision floating-point values in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTPD2PS *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	5A /r
VCVTPD2PS *xmm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.01	5A /r

Related Instructions

(V)CVTPS2PD, (V)CVTSD2SS, (V)CVTSS2SD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

CVTPS2DQ VCVTPS2DQ Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers

Converts packed single-precision floating-point values to packed signed doubleword integer values and writes the converted values to the destination.

When the result is an inexact value, it is rounded as specified by MXCSR.RC. When the floating-point value is a NaN, infinity, or the result of the conversion is larger than the maximum signed doubleword (-2^{31} to $+2^{31} - 1$), the instruction returns the 32-bit indefinite integer value (8000_0000h) when the invalid-operation exception (IE) is masked.

There are legacy and extended forms of the instruction:

CVTPS2DQ

Converts four packed single-precision floating-point values in an XMM register or a 128-bit memory location to four packed signed doubleword integer values and writes the converted values to an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VCVTPS2DQ

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Converts four packed single-precision floating-point values in an XMM register or a 128-bit memory location to four packed signed doubleword integer values and writes the converted values to an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Converts eight packed single-precision floating-point values in a YMM register or a 256-bit memory location to eight packed signed doubleword integer values and writes the converted values to a YMM register.

Instruction Support

Form	Subset	Feature Flag
CVTPS2DQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTPS2DQ	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTPS2DQ *xmm1*, *xmm2/mem128*	66 0F 5B /r	Converts four packed single-precision floating-point values in *xmm2* or *mem128* to four packed doubleword integers in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTPS2DQ *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	5B /r
VCVTPS2DQ *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.01	5B /r

Related Instructions

(V)CVTDQ2PS, (V)CVTSI2SS, (V)CVTSS2SI, (V)CVTTPS2DQ, (V)CVTTSS2SI

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

CVTPS2PD VCVTPS2PD Convert Packed Single-Precision Floating-Point to Packed Double-Precision Floating-Point

Converts packed single-precision floating-point values to packed double-precision floating-point values and writes the converted values to the destination.

There are legacy and extended forms of the instruction:

CVTPS2PD

Converts two packed single-precision floating-point values in the two low order doubleword elements of an XMM register or a 64-bit memory location to two double-precision floating-point values and writes the converted values to an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VCVTPS2PD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Converts two packed single-precision floating-point values in the two low order doubleword elements of an XMM register or a 64-bit memory location to two double-precision floating-point values and writes the converted values to an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Converts four packed single-precision floating-point values in a YMM register or a 128-bit memory location to four double-precision floating-point values and writes the converted values to a YMM register.

Instruction Support

Form	Subset	Feature Flag
CVTPS2PD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTPS2PD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTPS2PD *xmm1*, *xmm2/mem64*	0F 5A /r	Converts packed single-precision floating-point values in *xmm2* or *mem64* to packed double-precision floating-point values in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTPS2PD *xmm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.00	5A /r
VCVTPS2PD *ymm1*, *ymm2/mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.00	5A /r

Related Instructions

(V)CVTPD2PS, (V)CVTSD2SS, (V)CVTSS2SD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

CVTSD2SI VCVTSD2SI Convert Scalar Double-Precision Floating-Point to Signed Doubleword or Quadword Integer

Converts a scalar double-precision floating-point value to a 32-bit or 64-bit signed integer value and writes the converted value to a general-purpose register.

When the result is an inexact value, it is rounded as specified by MXCSR.RC. When the floating-point value is a NaN, infinity, or the result of the conversion is larger than the maximum signed doubleword (-2^{31} to $+2^{31} - 1$) or quadword value (-2^{63} to $+2^{63} - 1$), the instruction returns the indefinite integer value (8000_0000h for 32-bit integers, 8000_0000_0000_0000h for 64-bit integers) when the invalid-operation exception (IE) is masked.

There are legacy and extended forms of the instruction:

CVTSD2SI

The legacy form has two encodings:

- When REX.W = 0, converts a scalar double-precision floating-point value in the low-order 64 bits of an XMM register or a 64-bit memory location to a 32-bit signed integer and writes the converted value to a 32-bit general purpose register.
- When REX.W = 1, converts a scalar double-precision floating-point value in the low-order 64 bits of an XMM register or a 64-bit memory location to a 64-bit sign-extended integer and writes the converted value to a 64-bit general purpose register.

VCVTSD2SI

The extended form of the instruction has two 128-bit encodings:

- When VEX.W = 0, converts a scalar double-precision floating-point value in the low-order 64 bits of an XMM register or a 64-bit memory location to a 32-bit signed integer and writes the converted value to a 32-bit general purpose register.
- When VEX.W = 1, converts a scalar double-precision floating-point value in the low-order 64 bits of an XMM register or a 64-bit memory location to a 64-bit sign-extended integer and writes the converted value to a 64-bit general purpose register.

Instruction Support

Form	Subset	Feature Flag
CVTSD2SI	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTSD2SI	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTSD2SI *reg32*, *xmm1*/*mem64*	F2 (W0) 0F 2D /r	Converts a packed double-precision floating-point value in *xmm1* or *mem64* to a doubleword integer in *reg32*.
CVTSD2SI *reg64*, *xmm1*/*mem64*	F2 (W1) 0F 2D /r	Converts a packed double-precision floating-point value in *xmm1* or *mem64* to a quadword integer in *reg64*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTSD2SI *reg32*, *xmm2*/*mem64*	C4	$\overline{\text{RXB}}$.00001	0.1111.X.11	2D /r
VCVTSD2SI *reg64*, *xmm2*/*mem64*	C4	$\overline{\text{RXB}}$.00001	1.1111.X.11	2D /r

Related Instructions

(V)CVTDQ2PD, (V)CVTPD2DQ, (V)CVTPI2PD, (V)CVTSI2SD, (V)CVTTPD2DQ, (V)CVTT-SD2SI

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

CVTSD2SS VCVTSD2SS Convert Scalar Double-Precision Floating-Point to Scalar Single-Precision Floating-Point

Converts a scalar double-precision floating-point value to a scalar single-precision floating-point value and writes the converted value to the low-order 32 bits of the destination. When the result is an inexact value, it is rounded as specified by MXCSR.RC.

There are legacy and extended forms of the instruction:

CVTSD2SS

Converts a scalar double-precision floating-point value in the low-order 64 bits of the second source XMM register or a 64-bit memory location to a scalar single-precision floating-point value and writes the converted value to the low-order 32 bits of a destination XMM register. Bits [127:32] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VCVTSD2SS

The extended form of the instruction has a 128-bit encoding only.

Converts a scalar double-precision floating-point value in the low-order 64 bits of a source XMM register or a 64-bit memory location to a scalar single-precision floating-point value and writes the converted value to the low-order 32 bits of the destination XMM register. Bits [127:32] of the destination are copied from the first source XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
CVTSD2SS	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTSD2SS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTSD2SS *xmm1*, *xmm2/mem64*	F2 0F 5A /r	Converts a scalar double-precision floating-point value in *xmm2* or *mem64* to a scalar single-precision floating-point value in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTSD2SS *xmm1, xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.11	5A /r

Related Instructions

(V)CVTPD2PS, (V)CVTPS2PD, (V)CVTSS2SD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

CVTSI2SD VCVTSI2SD Convert Signed Doubleword or Quadword Integer to Scalar Double-Precision Floating-Point

Converts a signed integer value to a double-precision floating-point value and writes the converted value to a destination register. When the result of the conversion is an inexact value, the value is rounded as specified by MXCSR.RC.

There are legacy and extended forms of the instruction:

CVTSI2SD

The legacy form as two encodings:

- When REX.W = 0, converts a signed doubleword integer value from a 32-bit source general-purpose register or a 32-bit memory location to a double-precision floating-point value and writes the converted value to the low-order 64 bits of an XMM register. Bits [127:64] of the destination XMM register and bits [255:128] of the corresponding YMM register are not affected.
- When REX.W = 1, converts a a signed quadword integer value from a 64-bit source general-purpose register or a 64-bit memory location to a 64-bit double-precision floating-point value and writes the converted value to the low-order 64 bits of an XMM register. Bits [127:64] of the destination XMM register and bits [255:128] of the corresponding YMM register are not affected.

VCVTSI2SD

The extended form of the instruction has two 128-bit encodings:

- When VEX.W = 0, converts a signed doubleword integer value from a 32-bit source general-purpose register or a 32-bit memory location to a double-precision floating-point value and writes the converted value to the low-order 64 bits of the destination XMM register. Bits [127:64] of the first source XMM register are copied to the destination XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.
- When VEX.W = 1, converts a signed quadword integer value from a 64-bit source general-purpose register or a 64-bit memory location to a double-precision floating-point value and writes the converted value to the low-order 64 bits of the destination XMM register. Bits [127:64] of the first source XMM register are copied to the destination XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
CVTSI2SD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTSI2SD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTSI2SD *xmm1*, *reg32*/*mem32*	F2 (W0) 0F 2A /r	Converts a doubleword integer in *reg32* or *mem32* to a double-precision floating-point value in *xmm1*.
CVTSI2SD *xmm1*, *reg64*/*mem64*	F2 (W1) 0F 2A /r	Converts a quadword integer in *reg64* or *mem64* to a double-precision floating-point value in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTSI2SD *xmm1*, *xmm2*, *reg32*/*mem32*	C4	$\overline{\text{RXB}}$.00001	0.$\overline{src}$.X.11	2A /r
VCVTSI2SD *xmm1*, *xmm2*, *reg64*/*mem64*	C4	$\overline{\text{RXB}}$.00001	1.$\overline{src}$.X.11	2A /r

Related Instructions

(V)CVTDQ2PD, (V)CVTPD2DQ, (V)CVTPI2PD, (V)CVTSD2SI, (V)CVTTPD2DQ, (V)CVTT-SD2SI

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

CVTSI2SS VCVTSI2SS Convert Signed Doubleword or Quadword Integer to Scalar Single-Precision Floating-Point

Converts a signed integer value to a single-precision floating-point value and writes the converted value to an XMM register. When the result of the conversion is an inexact value, the value is rounded as specified by MXCSR.RC.

There are legacy and extended forms of the instruction:

CVTSI2SS

The legacy form has two encodings:

- When REX.W = 0, converts a signed doubleword integer value from a 32-bit source general-purpose register or a 32-bit memory location to a single-precision floating-point value and writes the converted value to the low-order 32 bits of an XMM register. Bits [127:32] of the destination XMM register and bits [255:128] of the corresponding YMM register are not affected.
- When REX.W = 1, converts a a signed quadword integer value from a 64-bit source general-purpose register or a 64-bit memory location to a single-precision floating-point value and writes the converted value to the low-order 32 bits of an XMM register. Bits [127:32] of the destination XMM register and bits [255:128] of the corresponding YMM register are not affected.

VCVTSI2SS

The extended form of the instruction has two 128-bit encodings:

- When VEX.W = 0, converts a signed doubleword integer value from a 32-bit source general-purpose register or a 32-bit memory location to a single-precision floating-point value and writes the converted value to the low-order 32 bits of the destination XMM register. Bits [127:32] of the first source XMM register are copied to the destination XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.
- When VEX.W = 1, converts a signed quadword integer value from a 64-bit source general-purpose register or a 64-bit memory location to a single-precision floating-point value and writes the converted value to the low-order 32 bits of the destination XMM register. Bits [127:32] of the first source XMM register are copied to the destination XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
CVTSI2SS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VCVTSI2SS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTSI2SS *xmm1*, *reg32*/*mem32*	F3 (W0) 0F 2A /r	Converts a doubleword integer in reg32 or *mem32* to a single-precision floating-point value in *xmm1*.
CVTSI2SS *xmm1*, *reg64*/*mem64*	F3 (W1) 0F 2A /r	Converts a quadword integer in reg64 or *mem64* to a single-precision floating-point value in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTSI2SS *xmm1*, *xmm2*, *reg32*/*mem32*	C4	$\overline{\text{RXB}}$.00001	0.$\overline{src}$.X.10	2A /r
VCVTSI2SS *xmm1*, *xmm2*, *reg64*/*mem64*	C4	$\overline{\text{RXB}}$.00001	1.$\overline{src}$.X.10	2A /r

Related Instructions

(V)CVTDQ2PS, (V)CVTPS2DQ, (V)CVTSS2SI, (V)CVTTPS2DQ, (V)CVTTSS2SI

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

CVTSS2SD VCVTSS2SD Convert Scalar Single-Precision Floating-Point to Scalar Double-Precision Floating-Point

Converts a scalar single-precision floating-point value to a scalar double-precision floating-point value and writes the converted value to the low-order 64 bits of the destination.

There are legacy and extended forms of the instruction:

CVTSS2SD

Converts a scalar single-precision floating-point value in the low-order 32 bits of a source XMM register or a 32-bit memory location to a scalar double-precision floating-point value and writes the converted value to the low-order 64 bits of a destination XMM register. Bits [127:64] of the destination and bits [255:128] of the corresponding YMM register are not affected.

VCVTSS2SD

The extended form of the instruction has a 128-bit encoding only.

Converts a scalar single-precision floating-point value in the low-order 32 bits of the second source XMM register or 32-bit memory location to a scalar double-precision floating-point value and writes the converted value to the low-order 64 bits of the destination XMM register. Bits [127:64] of the destination are copied from the first source XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
CVTSS2SD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTSS2SD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTSS2SD *xmm1*, *xmm2*/*mem32*	F3 0F 5A /r	Converts a scalar single-precision floating-point value in *xmm2* or *mem32* to a scalar double-precision floating-point value in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTSS2SD *xmm1, xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.10	5A /r

Related Instructions

(V)CVTPD2PS, (V)CVTPS2PD, (V)CVTSD2SS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

CVTSS2SI VCVTSS2SI Convert Scalar Single-Precision Floating-Point to Signed Doubleword or Quadword Integer

Converts a single-precision floating-point value to a signed integer value and writes the converted value to a general-purpose register.

When the result of the conversion is an inexact value, the value is rounded as specified by MXCSR.RC. When the floating-point value is a NaN, infinity, or the result of the conversion is larger than the maximum signed doubleword (-2^{31} to $+2^{31} - 1$) or quadword value (-2^{63} to $+2^{63} - 1$), the indefinite integer value (8000_0000h for 32-bit integers, 8000_0000_0000_0000h for 64-bit integers) is returned when the invalid-operation exception (IE) is masked.

There are legacy and extended forms of the instruction:

CVTSS2SI

The legacy form has two encodings:

- When REX.W = 0, converts a single-precision floating-point value in the low-order 32 bits of an XMM register or a 32-bit memory location to a 32-bit signed integer value and writes the converted value to a 32-bit general-purpose register.
- When REX.W = 1, converts a single-precision floating-point value in the low-order 32 bits of an XMM register or a 32-bit memory location to a 64-bit signed integer value and writes the converted value to a 64-bit general-purpose register.

VCVTSS2SI

The extended form of the instruction has two 128-bit encodings:

- When VEX.W = 0, converts a single-precision floating-point value in the low-order 32 bits of an XMM register or a 32-bit memory location to a 32-bit signed integer value and writes the converted value to a 32-bit general-purpose register.
- When VEX.W = 1, converts a single-precision floating-point value in the low-order 32 bits of an XMM register or a 32-bit memory location to a 64-bit signed integer value and writes the converted value to a 64-bit general-purpose register.

Instruction Support

Form	Subset	Feature Flag
CVTSS2SI	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VCVTSS2SI	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTSS2SI *reg32, xmm1/mem32*	F3 (W0) 0F 2D /r	Converts a single-precision floating-point value in *xmm1* or *mem32* to a 32-bit integer value in reg32
CVTSS2SI *reg64, xmm1//mem64*	F3 (W1) 0F 2D /r	Converts a single-precision floating-point value in *xmm1* or *mem64* to a 64-bit integer value in reg64

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTSS2SI *reg32, xmm1/mem32*	C4	$\overline{\text{RXB}}$.00001	0.1111.X.10	2D /r
VCVTSS2SI *reg64, xmm1/mem64*	C4	$\overline{\text{RXB}}$.00001	1.1111.X.10	2D /r

Related Instructions

(V)CVTDQ2PS, (V)CVTPS2DQ, (V)CVTSI2SS, (V)CVTTPS2DQ, (V)CVTTSS2SI

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

CVTTPD2DQ Convert Packed Double-Precision Floating-Point VCVTTPD2DQ to Packed Doubleword Integer, Truncated

Converts packed double-precision floating-point values to packed signed doubleword integer values and writes the converted values to the destination.

When the result is an inexact value, it is truncated (rounded toward zero). When the floating-point value is a NaN, infinity, or the result of the conversion is larger than the maximum signed doubleword (-2^{31} to $+2^{31} - 1$), the instruction returns the 32-bit indefinite integer value (8000_0000h) when the invalid-operation exception (IE) is masked.

There are legacy and extended forms of the instruction:

CVTTPD2DQ

Converts two packed double-precision floating-point values in an XMM register or a 128-bit memory location to two packed signed doubleword integers and writes the converted values to the two low-order doublewords of the destination XMM register. Bits [127:64] of the destination are cleared. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VCVTTPD2DQ

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Converts two packed double-precision floating-point values in an XMM register or a 128-bit memory location to two signed doubleword values and writes the converted values to the lower two doubleword elements of the destination XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Converts four packed double-precision floating-point values in a YMM register or a 256-bit memory location to four signed doubleword integer values and writes the converted values to an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
CVTTPD2DQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTTPD2DQ	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTTPD2DQ *xmm1*, *xmm2/mem128*	66 0F E6 /r	Converts two packed double-precision floating-point values in *xmm2* or *mem128* to packed doubleword integers in *xmm1*. Truncates inexact result.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTTPD2DQ *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	E6 /r
VCVTTPD2DQ *xmm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.01	E6 /r

Related Instructions

(V)CVTDQ2PD, (V)CVTPD2DQ, (V)CVTPI2PD, (V)CVTSD2SI, (V)CVTSI2SD, (V)CVTTSD2SI

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

CVTTPS2DQ Convert Packed Single-Precision Floating-Point VCVTTPS2DQ to Packed Doubleword Integers, Truncated

Converts packed single-precision floating-point values to packed signed doubleword integer values and writes the converted values to the destination.

When the result of the conversion is an inexact value, the value is truncated (rounded toward zero). When the floating-point value is a NaN, infinity, or the result of the conversion is larger than the maximum signed doubleword (-2^{31} to $+2^{31} - 1$), the instruction returns the 32-bit indefinite integer value (8000_0000h) when the invalid-operation exception (IE) is masked.

There are legacy and extended forms of the instruction:

CVTTPS2DQ

Converts four packed single-precision floating-point values in an XMM register or a 128-bit memory location to four packed signed doubleword integer values and writes the converted values to an XMM register. The high-order 128-bits of the corresponding YMM register are not affected.

VCVTTPS2DQ

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Converts four packed single-precision floating-point values in an XMM register or a 128-bit memory location to four packed signed doubleword integer values and writes the converted values to an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Converts eight packed single-precision floating-point values in a YMM register or a 256-bit memory location to eight packed signed doubleword integer values and writes the converted values to a YMM register.

Instruction Support

Form	Subset	Feature Flag
CVTTPS2DQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTTPS2DQ	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTTPS2DQ *xmm1*, *xmm2/mem128*	F3 0F 5B /r	Converts four packed single-precision floating-point values in *xmm2* or *mem128* to four packed doubleword integers in *xmm1*. Truncates inexact result.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTTPS2DQ *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.10	5B /r
VCVTTPS2DQ *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.10	5B /r

Related Instructions

(V)CVTDQ2PS, (V)CVTPS2DQ, (V)CVTSI2SS, (V)CVTSS2SI, (V)CVTTSS2SI

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

CVTTSD2SI Convert Scalar Double-Precision Floating-Point VCVTTSD2SI to Signed Double- or Quadword Integer, Truncated

Converts a scalar double-precision floating-point value to a signed integer value and writes the converted value to a general-purpose register.

When the result of the conversion is an inexact value, the value is truncated (rounded toward zero). When the floating-point value is a NaN, infinity, or the result of the conversion is larger than the maximum signed doubleword (-2^{31} to $+2^{31} - 1$) or quadword value (-2^{63} to $+2^{63} - 1$), the instruction returns the indefinite integer value (8000_0000h for 32-bit integers, 8000_0000_0000_0000h for 64-bit integers) when the invalid-operation exception (IE) is masked.

There are legacy and extended forms of the instruction:

CVTTSD2SI

The legacy form of the instruction has two encodings:

- When REX.W = 0, converts a scalar double-precision floating-point value in the low-order 64 bits of an XMM register or a 64-bit memory location to a 32-bit signed integer and writes the converted value to a 32-bit general purpose register.
- When REX.W = 1, converts a scalar double-precision floating-point value in the low-order 64 bits of an XMM register or a 64-bit memory location to a 64-bit sign-extended integer and writes the converted value to a 64-bit general purpose register.

VCVTTSD2SI

The extended form of the instruction has two 128-bit encodings.

- When VEX.W = 0, converts a scalar double-precision floating-point value in the low-order 64 bits of an XMM register or a 64-bit memory location to a 32-bit signed integer and writes the converted value to a 32-bit general purpose register.
- When VEX.W = 1, converts a scalar double-precision floating-point value in the low-order 64 bits of an XMM register or a 64-bit memory location to a 64-bit sign-extended integer and writes the converted value to a 64-bit general purpose register.

Instruction Support

Form	Subset	Feature Flag
CVTTSD2SI	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VCVTTSD2SI	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTTSD2SI *reg32*, *xmm1*/*mem64*	F2 (W0) 0F 2C /r	Converts a packed double-precision floating-point value in *xmm1* or *mem64* to a doubleword integer in *reg32*. Truncates inexact result.
CVTTSD2SI *reg64*, *xmm1*/*mem64*	F2 (W1) 0F 2C /r	Converts a packed double-precision floating-point value in *xmm1* or *mem64* to a quadword integer in *reg64*.Truncates inexact result.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTTSD2SI *reg32*, *xmm2*/*mem64*	C4	$\overline{\text{RXB}}$.00001	0.1111.X.11	2C /r
VCVTTSD2SI *reg64*, *xmm2*/*mem64*	C4	$\overline{\text{RXB}}$.00001	1.1111.X.11	2C /r

Related Instructions

(V)CVTDQ2PD, (V)CVTPD2DQ, (V)CVTPI2PD, (V)CVTSD2SI, (V)CVTSI2SD, (V)CVTTPD2DQ

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

CVTTSS2SI Convert Scalar Single-Precision Floating-Point VCVTTSS2SI to Signed Double or Quadword Integer, Truncated

Converts a single-precision floating-point value to a signed integer value and writes the converted value to a general-purpose register.

When the result of the conversion is an inexact value, the value is truncated (rounded toward zero). When the floating-point value is a NaN, infinity, or the result of the conversion is larger than the maximum signed doubleword (-2^{31} to $+2^{31} - 1$) or quadword value (-2^{63} to $+2^{63} - 1$), the indefinite integer value (8000_0000h for 32-bit integers, 8000_0000_0000_0000h for 64-bit integers) is returned when the invalid-operation exception (IE) is masked.

There are legacy and extended forms of the instruction:

CVTTSS2SI

The legacy form of the instruction has two encodings:

- When REX.W = 0, converts a single-precision floating-point value in the low-order 32 bits of an XMM register or a 32-bit memory location to a 32-bit signed integer value and writes the converted value to a 32-bit general-purpose register. Bits [255:128] of the YMM register that corresponds to the source are not affected.
- When REX.W = 1, converts a single-precision floating-point value in the low-order 32 bits of an XMM register or a 32-bit memory location to a 64-bit signed integer value and writes the converted value to a 64-bit general-purpose register. Bits [255:128] of the YMM register that corresponds to the source are not affected.

VCVTTSS2SI

The extended form of the instruction has two 128-bit encodings:

- When VEX.W = 0, converts a single-precision floating-point value in the low-order 32 bits of an XMM register or a 32-bit memory location to a 32-bit signed integer value and writes the converted value to a 32-bit general-purpose register. Bits [255:128] of the YMM register that corresponds to the source are cleared.
- When VEX.W = 1, converts a single-precision floating-point value in the low-order 32 bits of an XMM register or a 32-bit memory location to a 64-bit signed integer value and writes the converted value to a 64-bit general-purpose register. Bits [255:128] of the YMM register that corresponds to the source are cleared.

Instruction Support

Form	Subset	Feature Flag
CVTTSS2SI	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VCVTTSS2SI	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
CVTTSS2SI *reg32, xmm1/mem32*	F3 (W0) 0F 2C /r	Converts a single-precision floating-point value in *xmm1* or *mem32* to a 32-bit integer value in *reg32*. Truncates inexact result.
CVTTSS2SI *reg64, xmm1/mem64*	F3 (W1) 0F 2C /r	Converts a single-precision floating-point value in *xmm1* or *mem64* to a 64-bit integer value in *reg64*. Truncates inexact result.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTTSS2SI *reg32, xmm1/mem32*	C4	$\overline{\text{RXB}}$.00001	0.1111.X.10	2C /r
VCVTTSS2SI *reg64, xmm1/mem64*	C4	$\overline{\text{RXB}}$.00001	1.1111.X.10	2C /r

Related Instructions

(V)CVTDQ2PS, (V)CVTPS2DQ, (V)CVTSI2SS, (V)CVTSS2SI, (V)CVTTPS2DQ

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

DIVPD Divide
VDIVPD Packed Double-Precision Floating-Point

Divides each of the packed double-precision floating-point values of the first source operand by the corresponding packed double-precision floating-point values of the second source operand and writes the quotients to the destination.

There are legacy and extended forms of the instruction:

DIVPD

Divides two packed double-precision floating-point values in the first source XMM register by the corresponding packed double-precision floating-point values in either a second source XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VDIVPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Divides two packed double-precision floating-point values in the first source XMM register by the corresponding packed double-precision floating-point values in either a second source XMM register or a 128-bit memory location and writes the two results a destination XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Divides four packed double-precision floating-point values in the first source YMM register by the corresponding packed double-precision floating-point values in either a second source YMM register or a 256-bit memory location and writes the two results a destination YMM register.

Instruction Support

Form	Subset	Feature Flag
DIVPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VDIVPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
DIVPD *xmm1*, *xmm2/mem128*	66 0F 5E /r	Divides packed double-precision floating-point values in *xmm1* by the packed double-precision floating-point values in xmm2 or *mem128*. Writes quotients to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VDIVPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	5E /r
VDIVPD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.01	5E /r

Related Instructions

(V)DIVPS, (V)DIVSD, (V)DIVSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M	M	M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Division by zero, ZE	S	S	X	Division of finite dividend by zero-value divisor.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

DIVPS VDIVPS — Divide Packed Single-Precision Floating-Point

Divides each of the packed single-precision floating-point values of the first source operand by the corresponding packed single-precision floating-point values of the second source operand and writes the quotients to the destination.

There are legacy and extended forms of the instruction:

DIVPS

Divides four packed single-precision floating-point values in the first source XMM register by the corresponding packed single-precision floating-point values in either a second source XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VDIVPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Divides four packed single-precision floating-point values in the first source XMM register by the corresponding packed single-precision floating-point values in either a second source XMM register or a 128-bit memory location and writes two results to a third destination XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Divides eight packed single-precision floating-point values in the first source YMM register by the corresponding packed single-precision floating-point values in either a second source YMM register or a 256-bit memory location and writes the two results a destination YMM register.

Instruction Support

Form	Subset	Feature Flag
DIVPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VDIVPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
DIVPS *xmm1*, *xmm2/mem128*	0F 5E /r	Divides packed single-precision floating-point values in *xmm1* by the corresponding values in *xmm2* or *mem128*. Writes quotients to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VDIVPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.00	5E /r
VDIVPS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.00	5E /r

Related Instructions

(V)DIVPD, (V)DIVSD, (V)DIVSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M	M	M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Division by zero, ZE	S	S	X	Division of finite dividend by zero-value divisor.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

DIVSD Divide
VDIVSD Scalar Double-Precision Floating-Point

Divides the double-precision floating-point value in the low-order quadword of the first source operand by the double-precision floating-point value in the low-order quadword of the second source operand and writes the quotient to the low-order quadword of the destination.

There are legacy and extended forms of the instruction:

DIVSD

The first source operand is an XMM register and the second source operand is either an XMM register or a 64-bit memory location. The first source register is also the destination register. Bits [127:64] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VDIVSD

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 64-bit memory location. Bits [127:64] of the first source operand are copied to bits [127:64] of the destination. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
DIVSD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VDIVSD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
DIVSD *xmm1*, *xmm2/mem64*	F2 0F 5E /r	Divides the double-precision floating-point value in the low-order 64 bits of *xmm1*by the corresponding value in *xmm2* or *mem64*. Writes quotient to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VDIVSD *xmm1*, *xmm2*, *xmm3/mem64*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.11	5E /r

Related Instructions

(V)DIVPD, (V)DIVPS, (V)DIVSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M	M	M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Division by zero, ZE	S	S	X	Division of finite dividend by zero-value divisor.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

DIVSS VDIVSS Divide Scalar Single-Precision Floating-Point

Divides the single-precision floating-point value in the low-order doubleword of the first source operand by the single-precision floating-point value in the low-order doubleword of the second source operand and writes the quotient to the low-order doubleword of the destination.

There are legacy and extended forms of the instruction:

DIVSS

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location. The first source register is also the destination register. Bits [127:32] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VDIVSS

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 64-bit memory location. The destination is a third XMM register. Bits [127:32] of the first source operand are copied to bits [127:32] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
DIVSS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VDIVSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
DIVSS *xmm1*, *xmm2*/*mem32*	F3 0F 5E /r	Divides a single-precision floating-point value in the low-order doubleword of *xmm1* by a corresponding value in *xmm2* or *mem32*. Writes the quotient to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VDIVSS *xmm1*, *xmm2*, *xmm3*/*mem32*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.10	5E /r

Related Instructions

(V)DIVPD, (V)DIVPS, (V)DIVSD

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M	M	M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Division by zero, ZE	S	S	X	Division of finite dividend by zero-value divisor.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

DPPD Dot Product
VDPPD Packed Double-Precision Floating-Point

Computes the dot-product of the input operands. An immediate operand specifies both the input values and the destination locations to which the products are written.

Selectively multiplies packed double-precision values in a source operand by the corresponding values in a second source operand, writes the results to a temporary location, adds the results, writes the sum to a second temporary location and selectively writes the sum to a destination.

Mask bits [5:4] of an 8-bit immediate operand perform multiplicative selection. Bit 5 selects bits [127:64] of the source operands; bit 4 selects bits [63:0] of the source operands. When a mask bit = 1, the corresponding packed double-precision floating point values are multiplied and the product is written to the corresponding position of a 128-bit temporary location. When a mask bit = 0, the corresponding position of the temporary location is cleared.

After the two 64-bit values in the first temporary location are added and written to the 64-bit second temporary location, mask bits [1:0] of the same 8-bit immediate operand perform write selection. Bit 1 selects bits [127:64] of the destination; bit 0 selects bits [63:0] of the destination. When a mask bit = 1, the 64-bit value of the second temporary location is written to the corresponding position of the destination. When a mask bit = 0, the corresponding position of the destination is cleared.

When the operation produces a NaN, its value is determined as follows.

Source Operands (in either order)		NaN Result[1]
QNaN	Any non-NaN floating-point value (or single-operand instruction)	Value of QNaN
SNaN	Any non-NaN floating-point value (or single-operand instruction)	Value of SNaN, converted to a QNaN[2]
QNaN	QNaN	First operand
QNaN	SNaN	First operand (converted to QNaN if SNaN
SNaN	SNaN	First operand converted to a QNaN[2]

Note: *1. A NaN result produced when the floating-point invalid-operation exception is masked.*
2. The conversion is done by changing the most-significant fraction bit to 1.

For each addition occurring in either the second or third step, for the purpose of NaN propagation, the addend of lower bit index is considered to be the first of the two operands. For example, when both multiplications produce NaNs, the one that corresponds to bits [64:0] is written to all indicated fields of the destination, regardless of how those NaNs were generated from the sources. When the high-order multiplication produces NaNs and the low-order multiplication produces infinities of opposite signs, the real indefinite QNaN (produced as the sum of the infinities) is written to the destination.

NaNs in source operands or in computational results result in at least one NaN in the destination. For the 256-bit version, NaNs are propagated within the two independent dot product operations only to their respective 128-bit results.

There are legacy and extended forms of the instruction:

DPPD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VDPPD

The extended form of the instruction has a single 128-bit encoding.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
DPPD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VDPPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
DPPD *xmm1*, *xmm2/mem128, imm8*	66 0F 3A 41 /r ib	Selectively multiplies packed double-precision floating-point values in *xmm2* or *mem128* by corresponding values in *xmm1*, adds interim products, selectively writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VDPPD *xmm1*, *xmm2*, *xmm3/mem128, imm8*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.0.01	41 /r ib

Related Instructions

(V)DPPS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected. Exceptions are determined separately for each add-multiply operation. Unmasked exceptions do not affect the destination*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

DPPS Dot Product
VDPPS Packed Single-Precision Floating-Point

Computes the dot-product of the input operands. An immediate operand specifies both the input values and the destination locations to which the products are written.

Selectively multiplies packed single-precision values in a source operand by corresponding values in a second source operand, writes results to a temporary location, adds pairs of results, writes the sums to additional temporary locations, and selectively writes a cumulative sum to a destination.

Mask bits [7:4] of an 8-bit immediate operand perform multiplicative selection. Each bit selects a 32-bit segment of the source operands; bit 7 selects bits [127:96], bit 6 selects bits [95:64], bit 5 selects bits [63:32], and bit 4 selects bits [31:0]. When a mask bit = 1, the corresponding packed single-precision floating point values are multiplied and the product is written to the corresponding position of a 128-bit temporary location. When a mask bit = 0, the corresponding position of the temporary location is cleared.

After multiplication, three pairs of 32-bit values are added and written to temporary locations. Bits [63:32] and [31:0] of temporary location 1 are added and written to 32-bit temporary location 2; bits [127:96] and [95:64] of temporary location 1 are added and written to 32-bit temporary location 3; then the contents of temporary locations 2 and 3 are added and written to 32-bit temporary location 4.

After addition, mask bits [3:0] of the same 8-bit immediate operand perform write selection. Each bit selects a 32-bit segment of the source operands; bit 3 selects bits [127:96], bit 2 selects bits [95:64], bit 1 selects bits [63:32], and bit 0 selects bits [31:0] of the destination. When a mask bit = 1, the 64-bit value of the fourth temporary location is written to the corresponding position of the destination. When a mask bit = 0, the corresponding position of the destination is cleared.

For the 256-bit extended encoding, this process is performed on the upper and lower 128 bits of the affected YMM registers.

When the operation produces a NaN, its value is determined as follows.

Source Operands (in either order)		NaN Result[1]
QNaN	Any non-NaN floating-point value (or single-operand instruction)	Value of QNaN
SNaN	Any non-NaN floating-point value (or single-operand instruction)	Value of SNaN, converted to a QNaN[2]
QNaN	QNaN	First operand
QNaN	SNaN	First operand (converted to QNaN if SNaN
SNaN	SNaN	First operand converted to a QNaN[2]

Note: *1. A NaN result produced when the floating-point invalid-operation exception is masked.*
2. The conversion is done by changing the most-significant fraction bit to 1.

For each addition occurring in either the second or third step, for the purpose of NaN propagation, the addend of lower bit index is considered to be the first of the two operands. For example, when all four multiplications produce NaNs, the one that corresponds to bits [31:0] is written to all indicated fields

of the destination, regardless of how those NaNs were generated from the sources. When the two highest-order multiplication produce NaNs and the two lowest-low-order multiplications produce infinities of opposite signs, the real indefinite QNaN (produced as the sum of the infinities) is written to the destination.

NaNs in source operands or in computational results result in at least one NaN in the destination. For the 256-bit version, NaNs are propagated within the two independent dot product operations only to their respective 128-bit results.

There are legacy and extended forms of the instruction:

DPPS

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VDPPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
DPPS	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VDPPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
DPPS *xmm1*, *xmm2/mem128, imm8*	66 0F 3A 40 /r ib	Selectively multiplies packed single-precision floating-point values in *xmm2* or *mem128* by corresponding values in *xmm1*, adds interim products, selectively writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VDPPS *xmm1*, *xmm2*, *xmm3/mem128, imm8*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.0.01	40 /r ib
VDPPS *ymm1, ymm2, ymm3/mem256, imm8*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.1.01	40 /r ib

Related Instructions

(V)DPPD

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected. Exceptions are determined separately for each add-multiply operation. Unmasked exceptions do not affect the destination*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

EXTRACTPS Extract
VEXTRACTPS Packed Single-Precision Floating-Point

Copies one of four packed single-precision floating-point values from a source XMM register to a general purpose register or a 32-bit memory location.

Bits [1:0] of an immediate byte operand specify the location of the 32-bit value that is copied. 00b corresponds to the low word of the source register and 11b corresponds to the high word of the source register. Bits [7:2] of the immediate operand are ignored.

There are legacy and extended forms of the instruction:

EXTRACTPS

The source operand is an XMM register. The destination can be a general purpose register or a 32-bit memory location. A 32-bit single-precision value extracted to a general purpose register is zero-extended to 64-bits.

VEXTRACTPS

The extended form of the instruction has a single 128-bit encoding.

The source operand is an XMM register. The destination can be a general purpose register or a 32-bit memory location.

Instruction Support

Form	Subset	Feature Flag
EXTRACTPS	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VEXTRACTPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
EXTRACTPS *reg32/mem32*, *xmm1* *imm8*	66 0F 3A 17 /r ib	Extract the single-precision floating-point element of *xmm1* specified by *imm8* to *reg32/mem32*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VEXTRACTPS *reg32/mem32*, *xmm1*, *imm8*	C4	$\overline{\text{RXB}}$.00011	X.1111.0.01	17 /r ib

Related Instructions

(V)INSERTPS

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

EXTRQ Extract Field From Register

Extracts specified bits from the lower 64 bits of the first operand (the destination XMM register). The extracted bits are saved in the least-significant bit positions of the lower quadword of the destination; the remaining bits in the lower quadword of the destination register are cleared to 0. The upper quadword of the destination register is undefined.

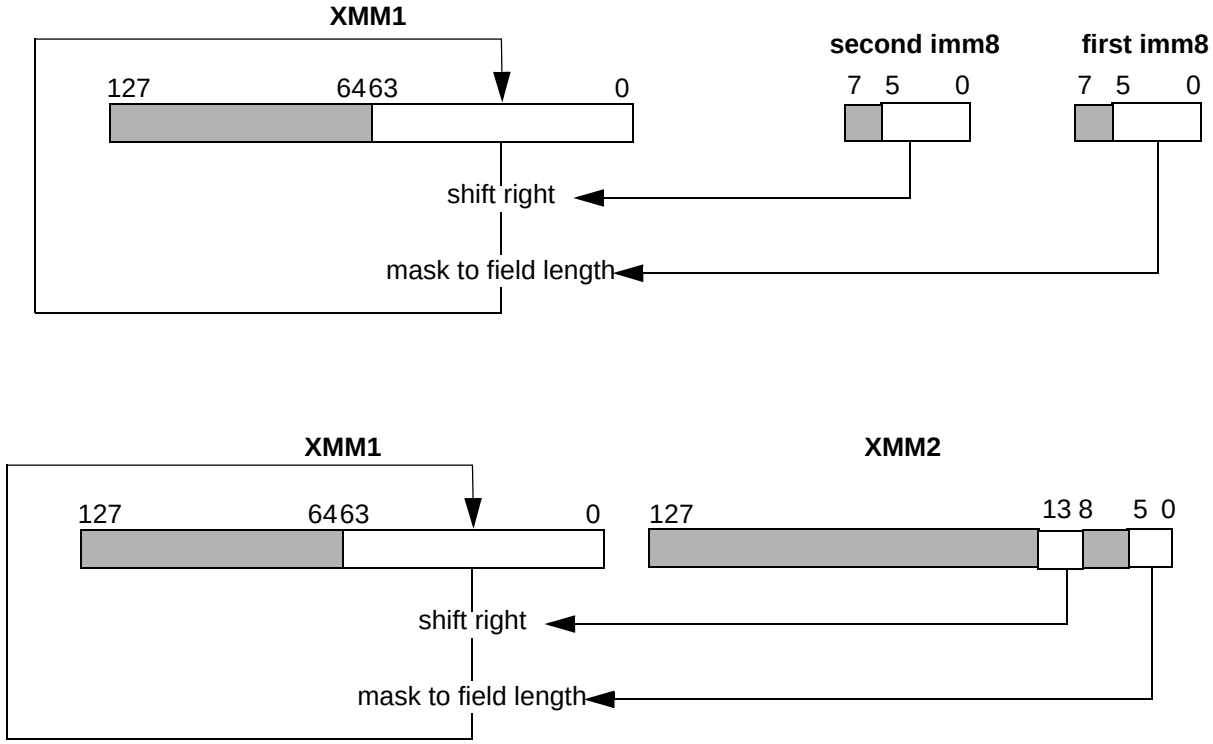
The portion of the source data being extracted is defined by the bit index and the field length. The bit index defines the least-significant bit of the source operand being extracted. Bits [*bit index* + *length field* – 1]:[*bit index*] are extracted. If the sum of the *bit index* + *length field* is greater than 64, the results are undefined.

For example, if the bit index is 32 (20h) and the field length is 16 (10h), then the result in the destination register will be source [47:32] in bits 15:0, with zeros in bits 63:16.

A value of zero in the field length is defined as a length of 64. If the *length field* is 0 and the *bit index* is 0, bits 63:0 of the source are extracted. For any other value of the *bit index*, the results are undefined.

The *bit index* and *field length* can be specified as immediate values (second and first immediate operands, respectively, in the case of the three argument version of the instruction), or they can both be specified by fields in an XMM source operand. In the latter case, bits [5:0] of the XMM register specify the number of bits to extract (the *field length*) and bits [13:8] of the XMM register specify the index of the first bit in the field to extract. The *bit index* and *field length* are each six bits in length; other bits of the field are ignored.

The diagram below illustrates the operation of this instruction.



Instruction Support

Form	Subset	Feature Flag
EXTRQ	SSE4A	CPUID Fn8000_0001_ECX[SSE4A] (bit 6)

Software *must* check the CPUID bit once per program or library initialization before using the instruction, or inconsistent behavior may result. For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
EXTRQ *xmm1*, *imm8*, *imm8*	66 0F 78 /0 ib ib	Extract field from *xmm1*, with the least significant bit of the extracted data starting at the bit index specified by [5:0] of the second immediate byte, with the length specified by [5:0] of the first immediate byte.
EXTRQ *xmm1*, *xmm2*	66 0F 79 /r	Extract field from *xmm1*, with the least significant bit of the extracted data starting at the bit index specified by xmm2[13:8], with the length specified by xmm2[5:0].

Related Instructions

INSERTQ, PINSRW, PEXTRW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	SSE4A instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[SSE4A] = 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 is cleared to 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.

HADDPD Horizontal Add
VHADDPD Packed Double-Precision Floating-Point

Adds adjacent pairs of double-precision floating-point values in two source operands and writes the sums to a destination.

There are legacy and extended forms of the instruction:

HADDPD

Adds the packed double-precision values in bits [127:64] and bits [63:0] of the first source XMM register and writes the sum to bits [63:0] of the destination; adds the corresponding doublewords of the second source XMM register or a 128-bit memory location and writes the sum to bits [127:64] of the destination. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VHADDPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Adds the packed double-precision values in bits [127:64] and bits [63:0] of the first source XMM register and writes the sum to bits [63:0] of the destination XMM register; adds the corresponding doublewords of the second source XMM register or a 128-bit memory location and writes the sum to bits [127:64] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Adds the packed double-precision values in bits [127:64] and bits [63:0] of the of the first source YMM register and writes the sum to bits [63:0] of the destination YMM register; adds the corresponding doublewords of the second source YMM register or a 256-bit memory location and writes the sum to bits [127:64] of the destination. Performs the same process for the upper 128 bits of the sources and destination.

Instruction Support

Form	Subset	Feature Flag
HADDPD	SSE3	CPUID Fn0000_0001_ECX[SSE3] (bit 0)
VHADDPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
HADDPD *xmm1*, *xmm2/mem128*	66 0F 7C /r	Adds adjacent pairs of double-precision values in *xmm1* and *xmm2* or *mem128*. Writes the sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VHADDPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	7C /r
VHADDPD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.01	7C /r

Related Instructions

(V)HADDPS, (V)HSUBPD, (V)HSUBPS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

HADDPS Horizontal Add
VHADDPS Packed Single-Precision

Adds adjacent pairs of single-precision floating-point values in two source operands and writes the sums to a destination.

There are legacy and extended forms of the instruction:

HADDPS

Adds the packed single-precision values in bits [63:32] and bits [31:0] of the first source XMM register and writes the sum to bits [31:0] of the destination; adds the packed single-precision values in bits [127:96] and bits [95:64] of the first source register and writes the sum to bits [63:32] of the destination. Adds the corresponding values in the second source XMM register or a 128-bit memory location and writes the sum to bits [95:64] and [127:96] of the destination. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VHADDPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Adds the packed single-precision values in bits [63:32] and bits [31:0] of the first source XMM register and writes the sum to bits [31:0] of the destination XMM register; adds the packed single-precision values in bits [127:96] and bits [95:64] of the first source register and writes the sum to bits [63:32] of the destination. Adds the corresponding values in the second source XMM register or a 128-bit memory location and writes the sum to bits [95:64] and [127:96] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Adds the packed single-precision values in bits [63:32] and bits [31:0] of the first source YMM register and writes the sum to bits [31:0] of the destination YMM register; adds the packed single-precision values in bits [127:96] and bits [95:64] of the first source register and writes the sum to bits [63:32] of the destination. Adds the corresponding values in the second source YMM register or a 256-bit memory location and writes the sums to bits [95:64] and [127:96] of the destination. Performs the same process for the upper 128 bits of the sources and destination.

Instruction Support

Form	Subset	Feature Flag
HADDPS	SSE3	CPUID Fn0000_0001_ECX[SSE3] (bit 0)
VHADDPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
HADDPS *xmm1*, *xmm2/mem128*	F2 0F 7C /r	Adds adjacent pairs of single-precision values in *xmm1* and *xmm2* or *mem128*. Writes the sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VHADDPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	RXB.00001	X.*src*.0.11	7C /r
VHADDPS *ymm1, ymm2, ymm3/mem256*	C4	RXB.00001	X.*src*.1.11	7C /r

Related Instructions

(V)HADDPD, (V)HSUBPD, (V)HSUBPS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

HSUBPD Horizontal Subtract
VHSUBPD Packed Double-Precision

Subtracts adjacent pairs of double-precision floating-point values in two source operands and writes the sums to a destination.

There are legacy and extended forms of the instruction:

HSUBPD

The first source register is also the destination.

Subtracts the packed double-precision value in bits [127:64] from the value in bits [63:0] of the first source XMM register and writes the difference to bits [63:0] of the destination; subtracts the corresponding values of the second source XMM register or a 128-bit memory location and writes the difference to bits [127:64] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VHSUBPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Subtracts the packed double-precision values in bits [127:64] from the value in bits [63:0] of the first source XMM register and writes the difference to bits [63:0] of the destination XMM register; subtracts the corresponding values of the second source XMM register or a 128-bit memory location and writes the difference to bits [127:64] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Subtracts the packed double-precision values in bits [127:64] from the value in bits [63:0] of the of the first source YMM register and writes the difference to bits [63:0] of the destination YMM register; subtracts the corresponding values of the second source YMM register or a 256-bit memory location and writes the difference to bits [127:64] of the destination. Performs the same process for the upper 128 bits of the sources and destination.

Instruction Support

Form	Subset	Feature Flag
HSUBPD	SSE3	CPUID Fn0000_0001_ECX[SSE3] (bit 0)
VHSUBPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
HSUBPD *xmm1*, *xmm2/mem128*	66 0F 7D /r	Subtracts adjacent pairs of double-precision floating-point values in *xmm1* and *xmm2* or *mem128*. Writes the differences to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VHSUBPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	7D /r
VHSUBPD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.01	7D /r

Related Instructions

(V)HSUBPS, (V)HADDPD, (V)HADDPS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

HSUBPS Horizontal Subtract Packed Single
VHSUBPS

Subtracts adjacent pairs of single-precision floating-point values in two source operands and writes the differences to a destination.

There are legacy and extended forms of the instruction:

HSUBPS

Subtracts the packed single-precision values in bits [63:32] from the values in bits [31:0] of the first source XMM register and writes the difference to bits [31:0] of the destination; subtracts the packed single-precision values in bits [127:96] from the value in bits [95:64] of the first source register and writes the difference to bits [63:32] of the destination. Subtracts the corresponding values of the second source XMM register or a 128-bit memory location and writes the differences to bits [95:64] and [127:96] of the destination. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VHSUBPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Subtracts the packed single-precision values in bits [63:32] from the value in bits [31:0] of the first source XMM register and writes the difference to bits [31:0] of the destination XMM register; subtracts the packed single-precision values in bits [127:96] from the value bits [95:64] of the first source register and writes the sum to bits [63:32] of the destination. Subtracts the corresponding values of the second source XMM register or a 128-bit memory location and writes the differences to bits [95:64] and [127:96] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Subtracts the packed single-precision values in bits [63:32] from the value in bits [31:0] of the first source YMM register and writes the difference to bits [31:0] of the destination YMM register; subtracts the packed single-precision values in bits [127:96] from the value in bits [95:64] of the first source register and writes the difference to bits [63:32] of the destination. Subtracts the corresponding values of the second source YMM register or a 256-bit memory location and writes the differences to bits [95:64] and [127:96] of the destination. Performs the same process for the upper 128 bits of the sources and destination.

Instruction Support

Form	Subset	Feature Flag
HSUBPS	SSE3	CPUID Fn0000_0001_ECX[SSE3] (bit 0)
VHSUBPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
HSUBPS *xmm1*, *xmm2/mem128*	F2 0F 7D /r	Subtracts adjacent pairs of values in *xmm1* and *xmm2* or *mem128*. Writes differences to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VHSUBPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	RXB.00001	X.*src*.0.11	7D /r
VHSUBPS *ymm1, ymm2, ymm3/mem256*	C4	RXB.00001	X.*src*.1.11	7D /r

Related Instructions

(V)HSUBPD, (V)HADDPD, (V)HADDPS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

INSERTPS Insert
VINSERTPS Packed Single-Precision Floating-Point

Copies a selected single-precision floating-point value from a source operand to a selected location in a destination register and optionally clears selected elements of the destination. The legacy and extended forms of the instruction treat the remaining elements of the destination in different ways.

Selections are specified by three fields of an immediate 8-bit operand:

7	6	5	4	3	2	1	0
COUNT_S		COUNT_D		ZMASK			

COUNT_S — The binary value of the field specifies a 32-bit element of a source register, counting upward from the low-order doubleword. COUNT_S is used only for register source; when the source is a memory operand, COUNT_S = 0.

COUNT_D — The binary value of the field specifies a 32-bit destination element, counting upward from the low-order doubleword.

ZMASK — Set a bit to clear a 32-bit element of the destination.

There are legacy and extended forms of the instruction:

INSERTPS

The source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

When the source operand is a register, the instruction copies the 32-bit element of the source specified by Count_S to the location in the destination specified by Count_D, and clears destination elements as specified by ZMask. Elements of the destination that are not cleared are not affected.

When the source operand is a memory location, the instruction copies a 32-bit value from memory, to the location in the destination specified by Count_D, and clears destination elements as specified by ZMask. Elements of the destination that are not cleared are not affected.

VINSERTPS

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

When the second source operand is a register, the instruction copies the 32-bit element of the source specified by Count_S to the location in the destination specified by Count_D. The other elements of the destination are either copied from the first source operand or cleared as specified by ZMask.

When the second source operand is a memory location, the instruction copies a 32-bit value from the source to the location in the destination specified by Count_D. The other elements of the destination are either copied from the first source operand or cleared as specified by ZMask.

Instruction Support

Form	Subset	Feature Flag
INSERTPS	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VINSERTPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
INSERTPS *xmm1*, *xmm2/mem32, imm8*	66 0F 3A 21 /r ib	Insert a selected single-precision floating-point value from *xmm2* or from *mem32* at a selected location in *xmm1* and clear selected elements of *xmm1*. Selections specified by imm8.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VINSERTPS *xmm1*, *xmm2*, *xmm3/mem128, imm8*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.0.01	21 /r ib

Related Instructions

(V)EXTRACTPS

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

INSERTQ Insert Field

Inserts bits from the lower 64 bits of the source operand into the lower 64 bits of the destination operand. No other bits in the lower 64 bits of the destination are modified. The upper 64 bits of the destination are undefined.

The least-significant *l* bits of the source operand are inserted into the destination, with the least-significant bit of the source operand inserted at bit position *n*, where *l* and *n* are defined as the *field length* and *bit index*, respectively.

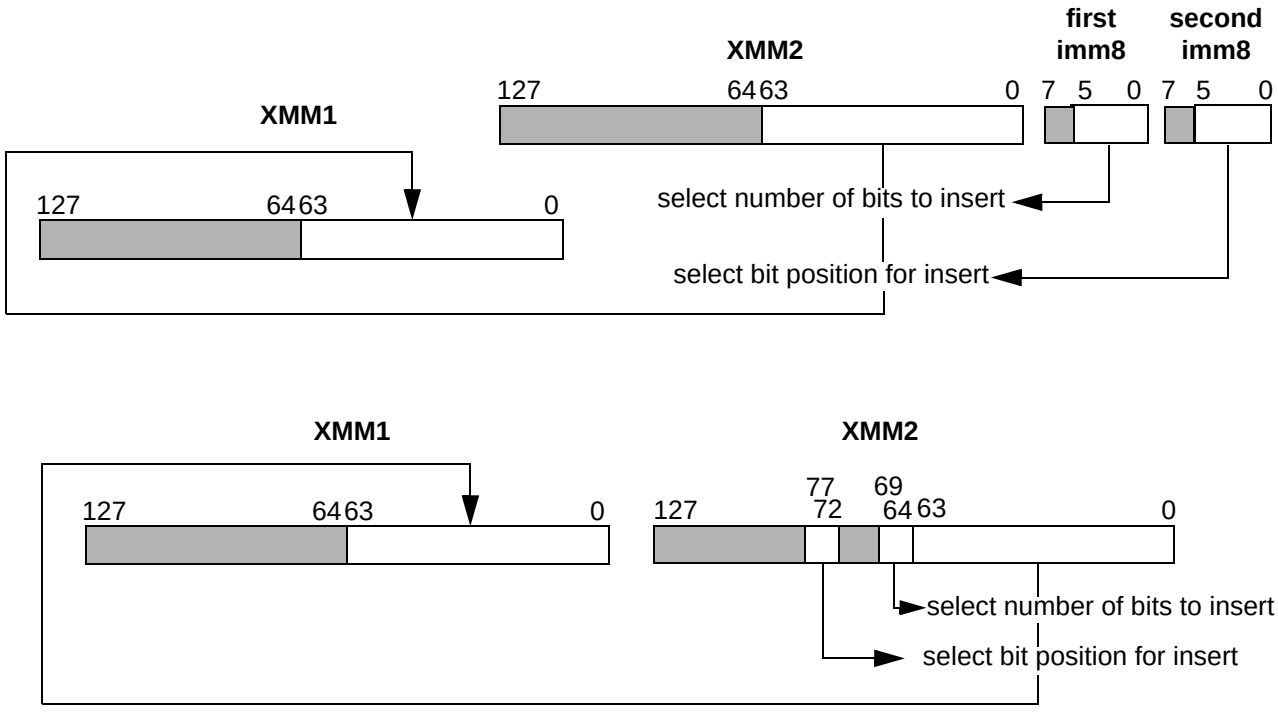
Bits (*field length* – 1):0 of the source operand are inserted into bits (*bit index* + *field length* – 1):(*bit index*) of the destination. If the sum of the *bit index* + *length field* is greater than 64, the results are undefined.

For example, if the *bit index* is 32 (20h) and the field length is 16 (10h), then the result in the destination register will be *source operand*[15:0] in bits 47:32. Bits 63:48 and bits 31:0 are not modified.

A value of zero in the *field length* is defined as a length of 64. If the *length field* is 0 and the *bit index* is 0, bits 63:0 of the source operand are inserted. For any other value of the *bit index*, the results are undefined.

The bits to insert are located in the XMM2 source operand. The *bit index* and *field length* can be specified as immediate values or can be specified in the XMM source operand. In the immediate form, the *bit index* and the *field length* are specified by the fourth (second immediate byte) and third operands (first immediate byte), respectively. In the register form, the *bit index* and *field length* are specified in bits [77:72] and bits [69:64] of the source XMM register, respectively. The *bit index* and *field length* are each six bits in length; other bits in the field are ignored.

The diagram below illustrates the operation of this instruction.



Instruction Support

Form	Subset	Feature Flag
INSERTQ	SSE4A	CPUID Fn8000_0001_ECX[SSE4A] (bit 6)

Software *must* check the CPUID bit once per program or library initialization before using the instruction, or inconsistent behavior may result. For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
INSERTQ *xmm1*, *xmm2*, *imm8*, *imm8*	F2 0F 78 /r ib ib	Insert field starting at bit 0 of xmm2 with the length specified by [5:0] of the first immediate byte. This field is inserted into xmm1 starting at the bit position specified by [5:0] of the second immediate byte.
INSERTQ *xmm1*, *xmm2*	F2 0F 79 /r	Insert field starting at bit 0 of xmm2 with the length specified by xmm2[69:64]. This field is inserted into xmm1 starting at the bit position specified by xmm2[77:72].

Related Instructions

EXTRQ, PINSRW, PEXTRW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	SSE4A instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[SSE4A] = 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 is cleared to 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.

LDDQU Load
VLDDQU Unaligned Double Quadword

Loads unaligned double quadwords from a memory location to a destination register.

Like the (V)MOVUPD instructions, (V)LDDQU loads a 128-bit or 256-bit operand from an unaligned memory location. However, to improve performance when the memory operand is actually misaligned, (V)LDDQU may read an aligned 16 or 32 bytes to get the first part of the operand, and an aligned 16 or 32 bytes to get the second part of the operand. This behavior is implementation-specific, and (V)LDDQU may only read the exact 16 or 32 bytes needed for the memory operand. If the memory operand is in a memory range where reading extra bytes can cause performance or functional issues, use (V)MOVUPD instead of (V)LDDQU.

Memory operands that are not aligned on 16-byte or 32-byte boundaries do not cause general-protection exceptions.

There are legacy and extended forms of the instruction:

LDDQU

The source operand is an unaligned 128-bit memory location. The destination operand is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination register are not affected.

VLDDQU

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The source operand is an unaligned 128-bit memory location. The destination operand is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination register are cleared.

YMM Encoding

The source operand is an unaligned 256-bit memory location. The destination operand is a YMM register.

Instruction Support

Form	Subset	Feature Flag
LDDQU	SSE3	CPUID Fn0000_0001_ECX[SSE3] (bit 0)
VLDDQU	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
LDDQU *xmm1*, *mem128*	F2 0F F0 /r	Loads a 128-bit value from an unaligned *mem128* to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VLDDQU *xmm1*, *mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.11	F0 /r
VLDDQU *ymm1*, *mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.11	F0 /r

Related Instructions

(V)MOVDQU

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	X	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

LDMXCSR Load
VLDMXCSR MXCSR Control/Status Register

Loads the MXCSR register with a 32-bit value from memory.

For both legacy LDMXCSR and extended VLDMXCSR forms of the instruction, the source operand is a 32-bit memory location and the destination operand is the MXCSR.

If an MXCSR load clears a SIMD floating-point exception mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception is not generated immediately. An exception is generated only when the next instruction that operates on an XMM or YMM register operand and causes that particular SIMD floating-point exception to be reported executes.

A general protection exception occurs if the instruction attempts to load non-zero values into reserved MXCSR bits. Software can use MXCSR_MASK to determine which bits are reserved. For details, see “128-Bit, 64-Bit, and x87 Programming” in Volume 2.

The MXCSR register is described in “Registers” in Volume 1.

Instruction Support

Form	Subset	Feature Flag
LDMXCSR	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VLDMXCSR	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
LDMXCSR *mem32*	0F AE */2*	Loads MXCSR register with 32-bit value from memory.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VLDMXCSR *mem32*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.00	AE /2

Related Instructions

(V)STMXCSR

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Null data segment used to reference memory.
	S	S	X	Attempt to load non-zero values into reserved MXCSR bits
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MASKMOVDQU Masked Move
VMASKMOVDQU Double Quadword Unaligned

Moves bytes from the first source operand to a memory location specified by the DS:rDI register. Bytes are selected by mask bits in the second source operand. The memory location may be unaligned.

The mask consists of the most significant bit of each byte of the second source register. When a mask bit = 1, the corresponding byte of the first source register is written to the destination; when a mask bit = 0, the corresponding byte is not written.

Exception and trap behavior for elements not selected for storage to memory is implementation dependent. For instance, a given implementation may signal a data breakpoint or a page fault for bytes that are zero-masked and not actually written.

The instruction implicitly uses weakly-ordered, write-combining buffering for the data, as described in “Buffering and Combining Memory Writes” in Volume 2. For data that is shared by multiple processors, this instruction should be used together with a fence instruction in order to ensure data coherency (see “Cache and TLB Management” in Volume 2).

There are legacy and extended forms of the instruction:

MASKMOVDQU

The first source operand is an XMM register and the second source operand is an XMM register. The destination is a 128-bit memory location.

VMASKMOVDQU

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is an XMM register. The destination is a 128-bit memory location.

Instruction Support

Form	Subset	Feature Flag
MASKMOVDQU	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMASKMOVDQU	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MASKMOVDQU *xmm1*, *xmm2*	66 0F F7 /r	Move bytes selected by a mask value in *xmm2* from *xmm1* to the memory location specified by DS:rDI.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMASKMOVDQU *xmm1*, *xmm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	F7 /r

Related Instructions

(V)MASKMOVPD, (V)MASKMOVPS

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MAXPD Maximum
VMAXPD Packed Double-Precision Floating-Point

Compares each packed double-precision floating-point value of the first source operand to the corresponding value of the second source operand and writes the numerically greater value into the corresponding location of the destination.

If both source operands are equal to zero, the value of the second source operand is returned. If either operand is a NaN (SNaN or QNaN), and invalid-operation exceptions are masked, the second source operand is written to the destination.

There are legacy and extended forms of the instruction:

MAXPD

Compares two pairs of packed double-precision floating-point values.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMAXPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Compares two pairs of packed double-precision floating-point values.

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Compares four pairs of packed double-precision floating-point values.

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
MAXPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMAXPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MAXPD *xmm1*, *xmm2/mem128*	66 0F 5F /r	Compares two pairs of packed double-precision values in *xmm1* and *xmm2* or *mem128* and writes the greater value to the corresponding position in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMAXPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	5F /r
VMAXPD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.01	5F /r

Related Instructions

(V)MAXPS, (V)MAXSD, (V)MAXSS, (V)MINPD, (V)MINPS, (V)MINSD, (V)MINSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MAXPS Maximum
VMAXPS Packed Single-Precision Floating-Point

Compares each packed single-precision floating-point value of the first source operand to the corresponding value of the second source operand and writes the numerically greater value into the corresponding location of the destination.

If both source operands are equal to zero, the value of the second source operand is returned. If either operand is a NaN (SNaN or QNaN), and invalid-operation exceptions are masked, the second source operand is written to the destination.

There are legacy and extended forms of the instruction:

MAXPS

Compares four pairs of packed single-precision floating-point values.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMAXPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Compares four pairs of packed single-precision floating-point values.

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Compares eight pairs of packed single-precision floating-point values.

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
MAXPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMAXPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MAXPS *xmm1*, *xmm2/mem128*	0F 5F /r	Compares four pairs of packed single-precision values in *xmm1* and *xmm2* or *mem128* and writes the greater values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMAXPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.00	5F /r
VMAXPS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.00	5F /r

Related Instructions

(V)MAXPD, (V)MAXSD, (V)MAXSS, (V)MINPD, (V)MINPS, (V)MINSD, (V)MINSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

MAXSD Maximum
VMAXSD Scalar Double-Precision Floating-Point

Compares the scalar double-precision floating-point value in the low-order 64 bits of the first source operand to a corresponding value in the second source operand and writes the numerically greater value into the low-order 64 bits of the destination.

If both source operands are equal to zero, the value of the second source operand is returned. If either operand is a NaN (SNaN or QNaN), and invalid-operation exceptions are masked, the second source operand is written to the destination.

There are legacy and extended forms of the instruction:

MAXSD

The first source operand is an XMM register. The second source operand is either an XMM register or a 64-bit memory location. The first source register is also the destination. When the second source is a 64-bit memory location, the upper 64 bits of the first source register are copied to the destination. Bits [127:64] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMAXSD

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. When the second source is a 64-bit memory location, the upper 64 bits of the first source register are copied to the destination. Bits [127:64] of the destination are copied from bits [127:64] of the first source. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MAXSD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMAXSD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MAXSD *xmm1*, *xmm2*/*mem64*	F2 0F 5F /r	Compares a pair of scalar double-precision values in the low-order 64 bits of *xmm1* and *xmm2* or *mem64* and writes the greater value to the low-order 64 bits of *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMAXSD *xmm1*, *xmm2*, *xmm3*/*mem64*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.11	5F /r

Related Instructions

(V)MAXPD, (V)MAXPS, (V)MAXSS, (V)MINPD, (V)MINPS, (V)MINSD, (V)MINSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

MAXSS Maximum
VMAXSS Scalar Single-Precision Floating-Point

Compares the scalar single-precision floating-point value in the low-order 32 bits of the first source operand to a corresponding value in the second source operand and writes the numerically greater value into the low-order 32 bits of the destination.

If both source operands are equal to zero, the value of the second source operand is returned. If either operand is a NaN (SNaN or QNaN), and invalid-operation exceptions are masked, the second source operand is written to the destination.

There are legacy and extended forms of the instruction:

MAXSS

The first source operand is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The first source register is also the destination. Bits [127:32] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMAXSS

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [127:32] of the destination are copied from the first source operand. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MAXSS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMAXSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MAXSS *xmm1*, *xmm2*/*mem32*	F3 0F 5F /r	Compares a pair of scalar single-precision values in the low-order 32 bits of *xmm1* and *xmm2* or *mem32* and writes the greater value to the low-order 32 bits of *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMAXSS *xmm1*, *xmm2*, *xmm3*/*mem32*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.10	5F /r

Related Instructions

(V)MAXPD, (V)MAXPS, (V)MAXSD, (V)MINPD, (V)MINPS, (V)MINSD, (V)MINSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

MINPD Minimum
VMINPD Packed Double-Precision Floating-Point

Compares each packed double-precision floating-point value of the first source operand to the corresponding value of the second source operand and writes the numerically lesser value into the corresponding location of the destination.

If both source operands are equal to zero, the value of the second source operand is returned. If either operand is a NaN (SNaN or QNaN), and invalid-operation exceptions are masked, the second source operand is written to the destination.

There are legacy and extended forms of the instruction:

MINPD

Compares two pairs of packed double-precision floating-point values.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMINPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Compares two pairs of packed double-precision floating-point values.

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Compares four pairs of packed double-precision floating-point values.

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
MINPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMINPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MINPD *xmm1*, *xmm2/mem128*	66 0F 5D /r	Compares two pairs of packed double-precision values in *xmm1* and *xmm2* or *mem128* and writes the lesser value to the corresponding position in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMINPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	5D /r
VMINPD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.01	5D /r

Related Instructions

(V)MAXPD, (V)MAXPS, (V)MAXSD, (V)MAXSS, (V)MINPS, (V)MINSD, (V)MINSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

MINPS Minimum
VMINPS Packed Single-Precision Floating-Point

Compares each packed single-precision floating-point value of the first source operand to the corresponding value of the second source operand and writes the numerically lesser value into the corresponding location of the destination.

If both source operands are equal to zero, the value of the second source operand is returned. If either operand is a NaN (SNaN or QNaN), and invalid-operation exceptions are masked, the second source operand is written to the destination.

There are legacy and extended forms of the instruction:

MINPS

Compares four pairs of packed single-precision floating-point values.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMINPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Compares four pairs of packed single-precision floating-point values.

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Compares eight pairs of packed single-precision floating-point values.

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
MINPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMINPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MINPS *xmm1*, *xmm2/mem128*	0F 5D /r	Compares four pairs of packed single-precision values in *xmm1* and *xmm2* or *mem128* and writes the lesser values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMINPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.00	5D /r
VMINPS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.00	5D /r

Related Instructions

(V)MAXPD, (V)MAXPS, (V)MAXSD, (V)MAXSS, (V)MINPD, (V)MINSD, (V)MINSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

MINSD Minimum
VMINSD Scalar Double-Precision Floating-Point

Compares the scalar double-precision floating-point value in the low-order 64 bits of the first source operand to a corresponding value in the second source operand and writes the numerically lesser value into the low-order 64 bits of the destination.

If both source operands are equal to zero, the value of the second source operand is returned. If either operand is a NaN (SNaN or QNaN), and invalid-operation exceptions are masked, the second source operand is written to the destination.

There are legacy and extended forms of the instruction:

MINSD

The first source operand is an XMM register. The second source operand is either an XMM register or a 64-bit memory location. The first source register is also the destination. Bits [127:64] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMINSD

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [127:64] of the destination are copied from the first source operand. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MINSD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMINSD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MINSD *xmm1*, *xmm2*/*mem64*	F2 0F 5D /r	Compares a pair of scalar double-precision values in the low-order 64 bits of *xmm1* and *xmm2* or *mem64* and writes the lesser value to the low-order 64 bits of *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMINSD *xmm1*, *xmm2*, *xmm3*/*mem64*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.11	5D /r

Related Instructions

(V)MAXPD, (V)MAXPS, (V)MAXSD, (V)MAXSS, (V)MINPD, (V)MINPS, (V)MINSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

MINSS Minimum
VMINSS Scalar Single-Precision Floating-Point

Compares the scalar single-precision floating-point value in the low-order 32 bits of the first source operand to a corresponding value in the second source operand and writes the numerically lesser value into the low-order 32 bits of the destination.

If both source operands are equal to zero, the value of the second source operand is returned. If either operand is a NaN (SNaN or QNaN), and invalid-operation exceptions are masked, the second source operand is written to the destination.

There are legacy and extended forms of the instruction:

MINSS

The first source operand is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The first source register is also the destination. Bits [127:32] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMINSS

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [127:32] of the destination are copied from the first source operand. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MINSS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMINSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MINSS *xmm1*, *xmm2*/*mem32*	F3 0F 5D /r	Compares a pair of scalar single-precision values in the low-order 32 bits of *xmm1* and *xmm2* or *mem32* and writes the lesser value to the low-order 32 bits of *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMINSS *xmm1*, *xmm2*, *xmm3*/*mem32*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.10	5D /r

Related Instructions

(V)MAXPD, (V)MAXPS, (V)MAXSD, (V)MAXSS, (V)MINPD, (V)MINPS, (V)MINSD

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVAPD Move Aligned
VMOVAPD Packed Double-Precision Floating-Point

Moves packed double-precision floating-point values. Values can be moved from a register or memory location to a register; or from a register to a register or memory location.

A memory operand that is not aligned causes a general-protection exception.

There are legacy and extended forms of the instruction:

MOVAPD

Moves two double-precision floating-point values. There are encodings for each type of move.

- The source operand is either an XMM register or a 128-bit memory location. The destination operand is an XMM register.
- The source operand is an XMM register. The destination operand is either an XMM register or a 128-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVAPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Moves two double-precision floating-point values. There are encodings for each type of move:

- The source operand is either an XMM register or a 128-bit memory location. The destination operand is an XMM register.
- The source operand is an XMM register. The destination operand is either an XMM register or a 128-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Moves four double-precision floating-point values. There are encodings for each type of move:

- The source operand is either a YMM register or a 256-bit memory location. The destination operand is a YMM register.
- The source operand is a YMM register. The destination operand is either a YMM register or a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
MOVAPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMOVAPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVAPD *xmm1*, *xmm2/mem128*	66 0F 28 /r	Moves two packed double-precision floating-point values from *xmm2* or *mem128* to *xmm1*.
MOVAPD *xmm1/mem128*, *xmm2*	66 0F 29 /r	Moves two packed double-precision floating-point values from *xmm1* or *mem128* to *xmm2*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVAPD *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	28 /r
VMOVAPD *xmm1/mem128*, *xmm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	29 /r
VMOVAPD *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.01	28 /r
VMOVAPD *ymm1/mem256*, *ymm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.01	29 /r

Related Instructions

(V)MOVHPD, (V)MOVLPD, (V)MOVMSKPD, (V)MOVSD, (V)MOVUPD

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not aligned on a 16-byte boundary.
	S	S	X	Write to a read-only data segment.
			A	VEX256: Memory operand not 32-byte aligned. VEX128: Memory operand not 16-byte aligned.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVAPS Move Aligned
VMOVAPS Packed Single-Precision Floating-Point

Moves packed single-precision floating-point values. Values can be moved from a register or memory location to a register; or from a register to a register or memory location.

A memory operand that is not aligned causes a general-protection exception.

There are legacy and extended forms of the instruction:

MOVAPS

Moves four single-precision floating-point values.

There are encodings for each type of move.

- The source operand is either an XMM register or a 128-bit memory location. The destination operand is an XMM register.
- The source operand is an XMM register. The destination operand is either an XMM register or a 128-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVAPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Moves four single-precision floating-point values. There are encodings for each type of move.

- The source operand is either an XMM register or a 128-bit memory location. The destination operand is an XMM register.
- The source operand is an XMM register. The destination operand is either an XMM register or a 128-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Moves eight single-precision floating-point values. There are encodings for each type of move.

- The source operand is either a YMM register or a 256-bit memory location. The destination operand is a YMM register.
- The source operand is a YMM register. The destination operand is either a YMM register or a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
MOVAPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMOVAPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVAPS *xmm1*, *xmm2/mem128*	0F 28 /r	Moves four packed single-precision floating-point values from *xmm2* or *mem128* to *xmm1*.
MOVAPS *xmm1/mem128*, *xmm2*	0F 29 /r	Moves four packed single-precision floating-point values from *xmm1* or *mem128* to *xmm2*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVAPS *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.00	28 /r
VMOVAPS *xmm1/mem128*, *xmm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.00	29 /r
VMOVAPS *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.00	28 /r
VMOVAPS *ymm1/mem256*, *ymm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.00	29 /r

Related Instructions

(V)MOVHLPS, (V)MOVHPS, (V)MOVLHPS, (V)MOVLPS, (V)MOVMSKPS, (V)MOVSS, (V)MOVUPS

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not aligned on a 16-byte boundary.
	S	S	X	Write to a read-only data segment.
			A	VEX256: Memory operand not 32-byte aligned. VEX128: Memory operand not 16-byte aligned.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

MOVD Move
VMOVD Doubleword or Quadword

Moves 32-bit and 64-bit values. A value can be moved from a general-purpose register or memory location to the corresponding low-order bits of an XMM register, with zero-extension to 128 bits; or from the low-order bits of an XMM register to a general-purpose register or memory location.

The quadword form of this instruction is distinct from the differently-encoded (V)MOVQ instruction.

There are legacy and extended forms of the instruction:

MOVD

There are two encodings for 32-bit moves, characterized by REX.W = 0.

- The source operand is either a 32-bit general-purpose register or a 32-bit memory location. The destination is an XMM register. The 32-bit value is zero-extended to 128 bits.
- The source operand is an XMM register. The destination is either a 32-bit general-purpose register or a 32-bit memory location.

There are two encodings for 64-bit moves, characterized by REX.W = 1.

- The source operand is either a 64-bit general-purpose register or a 64-bit memory location. The destination is an XMM register. The 64-bit value is zero-extended to 128 bits.
- The source operand is an XMM register. The destination is either a 64-bit general-purpose register or a 64-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVD

The extended form of the instruction has four 128-bit encodings:

There are two encodings for 32-bit moves, characterized by VEX.W = 0.

- The source operand is either a 32-bit general-purpose register or a 32-bit memory location. The destination is an XMM register. The 32-bit value is zero-extended to 128 bits.
- The source operand is an XMM register. The destination is either a 32-bit general-purpose register or a 32-bit memory location.

There are two encodings for 64-bit moves, characterized by VEX.W = 1.

- The source operand is either a 64-bit general-purpose register or a 64-bit memory location. The destination is an XMM register. The 64-bit value is zero-extended to 128 bits.
- The source operand is an XMM register. The destination is either a 64-bit general-purpose register or a 64-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MOVD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMOVD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVD *xmm*, *reg32*/*mem32*	66 (W0) 0F 6E /r	Move a 32-bit value from *reg32*/*mem32* to *xmm*.
MOVD *xmm*, *reg64*/*mem64*	66 (W1) 0F 6E /r	Move a 64-bit value from *reg64*/*mem64* to *xmm*.
MOVD *reg32*/*mem32*, *xmm*	66 (W0) 0F 7E /r	Move a 32-bit value from *xmm* to *reg32*/*mem32*
MOVD *reg64*/*mem64*, *xmm*	66 (W1) 0F 7E /r	Move a 64-bit value from *xmm* to *reg64*/*mem64*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVD[1] *xmm*, *reg32*/*mem32*	C4	$\overline{\text{RXB}}$.00001	0.1111.0.01	6E /r
VMOVQ *xmm*, *reg64*/*mem64*	C4	$\overline{\text{RXB}}$.00001	1.1111.0.01	6E /r
VMOVD[1] r*eg32*/*mem32*, *xmm*	C4	$\overline{\text{RXB}}$.00001	0.1111.0.01	7E /r
VMOVQ r*eg64*/*mem64*, *xmm*	C4	$\overline{\text{RXB}}$.00001	1.1111.0.01	7E /r

Note: *1. Also known as MOVQ in some developer tools.*

Related Instructions

(V)MOVDQA, (V)MOVDQU, (V)MOVQ

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVDDUP VMOVDDUP — Move and Duplicate Double-Precision Floating-Point

Moves and duplicates double-precision floating-point values.

There are legacy and extended forms of the instruction:

MOVDDUP

Moves and duplicates one quadword value.

The source operand is either the low 64 bits of an XMM register or the address of the least-significant byte of 64 bits of data in memory. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVDDUP

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Moves and duplicates one quadword value.

The source operand is either the low 64 bits of an XMM register or the address of the least-significant byte of 64 bits of data in memory. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Moves and duplicates two even-indexed quadword values.

The source operand is either a YMM register or the address of the least-significant byte of 256 bits of data in memory. The destination is a YMM register.Bits [63:0] of the source are written to bits [127:64] and [63:0] of the destination; bits [191:128] of the source are written to bits [255:192] and [191:128] of the destination.

Instruction Support

Form	Subset	Feature Flag
MOVDDUP	SSE3	CPUID Fn0000_0001_ECX[SSE3] (bit 0)
VMOVDDUP	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVDDUP *xmm1*, *xmm2/mem64*	F2 0F 12 /r	Moves two copies of the low 64 bits of *xmm2* or *mem64* to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
MOVDDUP *xmm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.11	12 /r
MOVDDUP *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.11	12 /r

Related Instructions

(V)MOVSHDUP, (V)MOVSLDUP

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference with alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVDQA
VMOVDQA

Move Aligned Double Quadword

Moves aligned packed integer values. Values can be moved from a register or a memory location to a register, or from a register to a register or a memory location.

A memory operand that is not aligned causes a general-protection exception.

There are legacy and extended forms of the instruction:

MOVDQA

Moves two aligned quadwords (128-bit move). There are two encodings.

- The source operand is an XMM register. The destination is either an XMM register or a 128-bit memory location.
- The source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register.

Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVDQA

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Moves two aligned quadwords (128-bit move). There are two encodings.

- The source operand is an XMM register. The destination is either an XMM register or a 128-bit memory location.
- The source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Moves four aligned quadwords (256-bit move). There are two encodings.

- The source operand is a YMM register. The destination is either a YMM register or a 256-bit memory location.
- The source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
MOVDQA	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMOVDQA	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVDQA *xmm1*, *xmm2/mem128*	66 0F 6F /r	Moves aligned packed integer values from *xmm2* or*mem128* to *xmm1*.
MOVDQA *xmm1/mem128*, *xmm2*	66 0F 7F /r	Moves aligned packed integer values from *xmm1* or *mem128* to *xmm2*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVDQA *xmm1*, *xmm2*/*mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	6F /r
VMOVDQA *xmm1/mem128*, *xmm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	6F /r
VMOVDQA *ymm1*, *xmm2*/*mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.01	7F /r
VMOVDQA *ymm1*/*mem256*, *ymm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.01	7F /r

Related Instructions

(V)MOVD, (V)MOVDQU, (V)MOVQ

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not aligned on a 16-byte boundary.
	S	S	X	Write to a read-only data segment.
			A	VEX256: Memory operand not 32-byte aligned. VEX128: Memory operand not 16-byte aligned.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVDQU Move
VMOVDQU Unaligned Double Quadword

Moves unaligned packed integer values. Values can be moved from a register or a memory location to a register, or from a register to a register or a memory location.

There are legacy and extended forms of the instruction:

MOVDQU

Moves two unaligned quadwords (128-bit move). There are two encodings.

- The source operand is an XMM register. The destination is either an XMM register or a 128-bit memory location.
- The source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register.

Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVDQU

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Moves two unaligned quadwords (128-bit move). There are two encodings:

- The source operand is an XMM register. The destination is either an XMM register or a 128-bit memory location.
- The source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Moves four unaligned quadwords (256-bit move). There are two encodings:

- The source operand is a YMM register. The destination is either a YMM register or a 256-bit memory location.
- The source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
MOVDQU	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMOVDQU	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVDQU *xmm1*, *xmm2/mem128*	F3 0F 6F /r	Moves unaligned packed integer values from *xmm2* or *mem128* to *xmm1*.
MOVDQU *xmm1/mem128*, *xmm2*	F3 0F 7F /r	Moves unaligned packed integer values from *xmm1* or *mem128* to *xmm2*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVDQU *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.10	6F /r
VMOVDQU *xmm1/mem128*, *xmm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.10	6F /r
VMOVDQU *ymm1*, *xmm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.10	7F /r
VMOVDQU *ymm1/mem256*, *ymm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.10	7F /r

Related Instructions

(V)MOVD, (V)MOVDQA, (V)MOVQ

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	X	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVHLPS Move High to Low
VMOVHLPS Packed Single-Precision Floating-Point

Moves two packed single-precision floating-point values from the high quadword of an XMM register to the low quadword of an XMM register.

There are legacy and extended forms of the instruction:

MOVHLPS

The source operand is bits [127:64] of an XMM register. The destination is bits [63:0] of an XMM register. Bits [127:64] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVHLPS

The extended form of the instruction has a 128-bit encoding only.

The source operands are bits [127:64] of two XMM registers. The destination is a third XMM register. Bits [127:64] of the first source are moved to bits [127:64] of the destination; bits [127:64] of the second source are moved to bits [63:0] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MOVHLPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMOVHLPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVHLPS *xmm1*, *xmm2*	0F 12 /r	Moves two packed single-precision floating-point values from *xmm2[127:64]* to *xmm1[63:0]*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVHLPS *xmm1*, *xmm2*, *xmm3*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.00	12 /r

Related Instructions

(V)MOVAPS, (V)MOVHPS, (V)MOVLHPS, (V)MOVLPS, (V)MOVMSKPS, (V)MOVSS, (V)MOVUPS

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVHPD Move High
VMOVHPD Packed Double-Precision Floating-Point

Moves a packed double-precision floating-point value. Values can be moved from a 64-bit memory location to the high-order quadword of an XMM register, or from the high-order quadword of an XMM register to a 64-bit memory location.

There are legacy and extended forms of the instruction:

MOVHPD

There are two encodings.

- The source operand is a 64-bit memory location. The destination is bits [127:64] of an XMM register.
- The source operand is bits [127:64] of an XMM register. The destination is a 64-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVHPD

The extended form of the instruction has two 128-bit encodings:

- There are two source operands. The first source is an XMM register. The second source is a 64-bit memory location. The destination is an XMM register. Bits [63:0] of the source register are written to bits [63:0] of the destination; bits [63:0] of the source memory location are written to bits [127:64] of the destination.
- The source operand is bits [127:64] of an XMM register. The destination is a 64-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MOVHPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMOVHPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVHPD *xmm1*, *mem64*	66 0F 16 /r	Moves a packed double-precision floating-point value from *mem64* to *xmm1[127:64].*
MOVHPD *mem64*, *xmm1*	66 0F 17 /r	Moves a packed double-precision floating-point value from *xmm1[127:64]* to *mem64*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVHPD *xmm1*, *xmm2*, *mem64*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	16 /r
VMOVHPD *mem64*, *xmm1*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	17 /r

Related Instructions

(V)MOVAPD, (V)MOVLPD, (V)MOVMSKPD, (V)MOVSD, (V)MOVUPD

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b (for memory destination encoding only).
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVHPS Move High
VMOVHPS Packed Single-Precision Floating-Point

Moves two packed single-precision floating-point value. Values can be moved from a 64-bit memory location to the high-order quadword of an XMM register, or from the high-order quadword of an XMM register to a 64-bit memory location.

There are legacy and extended forms of the instruction:

MOVHPS

There are two encodings.

- The source operand is a 64-bit memory location. The destination is bits [127:64] of an XMM register.
- The source operand is bits [127:64] of an XMM register. The destination is a 64-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVHPS

The extended form of the instruction has two 128-bit encodings:

- There are two source operands. The first source is an XMM register. The second source is a 64-bit memory location. The destination is an XMM register. Bits [63:0] of the source register are written to bits [63:0] of the destination; bits [63:0] of the source memory location are written to bits [127:64] of the destination.
- The source operand is bits [127:64] of an XMM register. The destination is a 64-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MOVHPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMOVHPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVHPS *xmm1*, *mem64*	0F 16 /r	Moves two packed double-precision floating-point value from *mem64* to *xmm1[127:64].*
MOVHPS *mem64*, *xmm1*	0F 17 /r	Moves two packed double-precision floating-point value from *xmm1[127:64]* to *mem64*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVHPS *xmm1*, *xmm2*, *mem64*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.00	16 /r
VMOVHPS *mem64*, *xmm1*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.00	17 /r

Related Instructions

(V)MOVAPS, (V)MOVHLPS, (V)MOVLHPS, (V)MOVLPS, (V)MOVMSKPS, (V)MOVSS, (V)MOVUPS

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b (for memory destination encoding only).
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVLHPS Move Low to High
VMOVLHPS Packed Single-Precision Floating-Point

Moves two packed single-precision floating-point values from the low quadword of an XMM register to the high quadword of a second XMM register.

There are legacy and extended forms of the instruction:

MOVLHPS

The source operand is bits [63:0] of an XMM register. The destination is bits [127:64] of an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVLHPS

The extended form of the instruction has a 128-bit encoding only.

The source operands are bits [63:0] of two XMM registers. The destination is a third XMM register. Bits [63:0] of the first source are moved to bits [63:0] of the destination; bits [63:0] of the second source are moved to bits [127:64] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MOVLHPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMOVLHPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVLHPS *xmm1*, *xmm2*	0F 16 /r	Moves two packed single-precision floating-point values from *xmm2[63:0]* to *xmm1[127:64]*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVLHPS *xmm1*, *xmm2*, *xmm3*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.00	16 /r

Related Instructions

(V)MOVAPS, (V)MOVHLPS, (V)MOVHPS, (V)MOVLPS, (V)MOVMSKPS, (V)MOVSS, (V)MOVUPS

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVLPD Move Low
VMOVLPD Packed Double-Precision Floating-Point

Moves a packed double-precision floating-point value. Values can be moved from a 64-bit memory location to the low-order quadword of an XMM register, or from the low-order quadword of an XMM register to a 64-bit memory location.

There are legacy and extended forms of the instruction:

MOVLPD

There are two encodings.

- The source operand is a 64-bit memory location. The destination is bits [63:0] of an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.
- The source operand is bits [63:0] of an XMM register. The destination is a 64-bit memory location.

VMOVLPD

The extended form of the instruction has two 128-bit encodings.

- There are two source operands. The first source is an XMM register. The second source is a 64-bit memory location. The destination is an XMM register. Bits [127:64] of the source register are written to bits [127:64] of the destination; bits [63:0] of the source memory location are written to bits [63:0] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.
- The source operand is bits [63:0] of an XMM register. The destination is a 64-bit memory location.

Instruction Support

Form	Subset	Feature Flag
MOVLPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMOVLPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVLPD *xmm1*, *mem64*	66 0F 12 /r	Moves a packed double-precision floating-point value from *mem64* to *xmm1[63:0].*
MOVLPD *mem64*, *xmm1*	66 0F 13 /r	Moves a packed double-precision floating-point value from *xmm1[63:0]* to *mem64*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVLPD *xmm1*, *xmm2*, *mem64*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	12 /r
VMOVLPD *mem64*, *xmm1*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	13 /r

Related Instructions

(V)MOVAPD, (V)MOVHPD, (V)MOVMSKPD, (V)MOVSD, (V)MOVUPD

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b (for memory destination encoding only).
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVLPS VMOVLPS — Move Low Packed Single-Precision Floating-Point

Moves two packed single-precision floating-point values. Values can be moved from a 64-bit memory location to the low-order quadword of an XMM register, or from the low-order quadword of an XMM register to a 64-bit memory location.

There are legacy and extended forms of the instruction:

MOVLPS

There are two encodings.

- The source operand is a 64-bit memory location. The destination is bits [63:0] of an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.
- The source operand is bits [63:0] of an XMM register. The destination is a 64-bit memory location.

VMOVLPS

The extended form of the instruction has two 128-bit encodings.

- There are two source operands. The first source is an XMM register. The second source is a 64-bit memory location. The destination is an XMM register. Bits [127:64] of the source register are written to bits [127:64] of the destination; bits [63:0] of the source memory location are written to bits [63:0] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.
- The source operand is bits [63:0] of an XMM register. The destination is a 64-bit memory location.

Instruction Support

Form	Subset	Feature Flag
MOVLPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMOVLPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVLPS *xmm1*, *mem64*	0F 12 /r	Moves two packed single-precision floating-point value from *mem64* to *xmm1[63:0].*
MOVLPS *mem64*, *xmm1*	0F 13 /r	Moves two packed single-precision floating-point value from *xmm1[63:0]* to *mem64*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVLPS *xmm1*, *xmm2*, *mem64*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.00	12 /r
VMOVLPS *mem64*, *xmm1*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.00	13 /r

Related Instructions

(V)MOVAPS, (V)MOVHLPS, (V)MOVHPS, (V)MOVLHPS, (V)MOVMSKPS, (V)MOVSS, (V)MOVUPS

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b (for memory destination encoding only).
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVMSKPD Extract Sign Mask
VMOVMSKPD Packed Double-Precision Floating-Point

Extracts the sign bits of packed double-precision floating-point values from an XMM register, zero-extends the value, and writes it to the low-order bits of a general-purpose register.

There are legacy and extended forms of the instruction:

MOVMSKPD

Extracts two mask bits.

The source operand is an XMM register. The destination can be either a 64-bit or a 32-bit general purpose register. Writes the extracted bits to positions [1:0] of the destination and clears the remaining bits. Bits [255:128] of the YMM register that corresponds to the source are not affected.

MOVMSKPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Extracts two mask bits.

The source operand is an XMM register. The destination can be either a 64-bit or a 32-bit general purpose register. Writes the extracted bits to positions [1:0] of the destination and clears the remaining bits. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Extracts four mask bits.

The source operand is a YMM register. The destination can be either a 64-bit or a 32-bit general purpose register. Writes the extracted bits to positions [3:0] of the destination and clears the remaining bits.

Instruction Support

Form	Subset	Feature Flag
MOVMSKPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMOVMSKPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVMSKPD *reg*, *xmm*	66 0F 50 /r	Move zero-extended sign bits of packed double-precision values from *xmm* to a general-purpose register.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVMSKPD *reg*, *xmm*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	50 /r
VMOVMSKPD *reg*, *ymm*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.01	50 /r

Related Instructions

(V)MOVMSKPS, (V)PMOVMSKB

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVMSKPS Extract Sign Mask
VMOVMSKPS Packed Single-Precision Floating-Point

Extracts the sign bits of packed single-precision floating-point values from an XMM register, zero-extends the value, and writes it to the low-order bits of a general-purpose register.

There are legacy and extended forms of the instruction:

MOVMSKPS

Extracts four mask bits.

The source operand is an XMM register. The destination can be either a 64-bit or a 32-bit general purpose register. Writes the extracted bits to positions [3:0] of the destination and clears the remaining bits.

MOVMSKPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Extracts four mask bits.

The source operand is an XMM register. The destination can be either a 64-bit or a 32-bit general purpose register. Writes the extracted bits to positions [3:0] of the destination and clears the remaining bits.

YMM Encoding

Extracts eight mask bits.

The source operand is a YMM register. The destination can be either a 64-bit or a 32-bit general purpose register. Writes the extracted bits to positions [7:0] of the destination and clears the remaining bits.

Instruction Support

Form	Subset	Feature Flag
MOVMSKPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMOVMSKPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVMSKPS *reg*, *xmm*	0F 50 /r	Move zero-extended sign bits of packed single-precision values from *xmm* to a general-purpose register.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVMSKPS *reg*, *xmm*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.00	50 /r
VMOVMSKPS *reg*, *ymm*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.00	50 /r

Related Instructions

(V)MOVMSKPD, (V)PMOVMSKB

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVNTDQ
VMOVNTDQ

Move Non-Temporal Double Quadword

Moves double quadword values from a register to a memory location.

Indicates to the processor that the data is non-temporal, and is unlikely to be used again soon. The processor treats the store as a write-combining (WC) memory write, which minimizes cache pollution. The method of minimization depends on the hardware implementation of the instruction. For further information, see “Memory Optimization” in Volume 1.

The instruction is weakly-ordered with respect to other instructions that operate on memory. Software should use an SFENCE or MFENCE instruction to force strong memory ordering of MOVNTDQ with respect to other stores.

An attempted store to a non-aligned memory location results in a #GP exception.

There are legacy and extended forms of the instruction:

MOVNTDQ

Moves one 128-bit value.

The source operand is an XMM register. The destination is a 128-bit memory location.

VMOVNTDQ

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Moves one 128-bit value.

The source operand is an XMM register. The destination is a 128-bit memory location.

YMM Encoding

Moves two 128-bit values.

The source operand is a YMM register. The destination is a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
MOVNTDQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMOVNTDQ	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVNTDQ *mem128*, *xmm*	66 0F E7 /r	Moves a 128-bit value from *xmm* to *mem128*, minimizing cache pollution.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVNTDQ *mem128, xmm*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	E7 /r
VMOVNTDQ *mem256, ymm*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.01	E7 /r

Related Instructions

(V)MOVNTDQA, (V)MOVNTPD, (V)MOVNTPS

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not aligned on a 16-byte boundary.
	S	S	X	Write to a read-only data segment.
			A	VEX256: Memory operand not 32-byte aligned. VEX128: Memory operand not 16-byte aligned.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVNTDQA Move Non-Temporal
VMOVNTDQA Double Quadword Aligned

Loads an XMM/YMM register from a naturally-aligned 128-bit or 256-bit memory location.

Indicates to the processor that the data is non-temporal, and is unlikely to be used again soon. The processor treats the load as a write-combining (WC) memory read, which minimizes cache pollution. The method of minimization depends on the hardware implementation of the instruction. For further information, see “Memory Optimization” in Volume 1.

The instruction is weakly-ordered with respect to other instructions that operate on memory. Software should use an MFENCE instruction to force strong memory ordering of MOVNTDQA with respect to other reads.

An attempted load from a non-aligned memory location results in a #GP exception.

There are legacy and extended forms of the instruction:

MOVNTDQA

Loads a 128-bit value into the specified XMM register from a 16-byte aligned memory location.

VMOVNTDQA

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Loads a 128-bit value into the specified XMM register from a 16-byte aligned memory location.

YMM Encoding

Loads a 256-bit value into the specified YMM register from a 32-byte aligned memory location.

Instruction Support

Form	Subset	Feature Flag
MOVNTDQA	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VMOVNTDQA 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VMOVNTDQA 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVNTDQA *xmm, mem128*	66 0F 38 2A /r	Loads xmm from an aligned memory location, minimizing cache pollution.

		Encoding		
Mnemonic	**VEX**	**RXB.map_select**	**W.vvvv.L.pp**	**Opcode**
VMOVNTDQA *xmm, mem128*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	2A /r
VMOVNTDQA *ymm, mem256*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	2A /r

Related Instructions

(V)MOVNTDQ, (V)MOVNTPD, (V)MOVNTPS

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not aligned on a 16-byte boundary.
	S	S	X	Write to a read-only data segment.
			A	256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX, AVX2, and SSE exception
A — AVX, AVX2 exception
S — SSE exception

MOVNTPD VMOVNTPD Move Non-Temporal Packed Double-Precision Floating-Point

Moves packed double-precision floating-point values from a register to a memory location.

Indicates to the processor that the data is non-temporal, and is unlikely to be used again soon. The processor treats the store as a write-combining (WC) memory write, which minimizes cache pollution. The method of minimization depends on the hardware implementation of the instruction. For further information, see “Memory Optimization” in Volume 1.

The instruction is weakly-ordered with respect to other instructions that operate on memory. Software should use an SFENCE or MFENCE instruction to force strong memory ordering of MOVNTDQ with respect to other stores.

An attempted store to a non-aligned memory location results in a #GP exception.

There are legacy and extended forms of the instruction:

MOVNTPD

Moves two values.

The source operand is an XMM register. The destination is a 128-bit memory location.

MOVNTPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Moves two values.

The source operand is an XMM register. The destination is a 128-bit memory location.

YMM Encoding

Moves four values.

The source operand is a YMM register. The destination is a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
MOVNTPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMOVNTPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVNTPD *mem128*, *xmm*	66 0F 2B /r	Moves two packed double-precision floating-point values from *xmm* to *mem128*, minimizing cache pollution.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVNTPD *mem128, xmm*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	2B /r
VMOVNTPD *mem256, ymm*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.01	2B /r

Related Instructions

MOVNTDQ, MOVNTI, MOVNTPS, MOVNTQ

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not aligned on a 16-byte boundary.
	S	S	X	Write to a read-only data segment.
			A	VEX256: Memory operand not 32-byte aligned. VEX128: Memory operand not 16-byte aligned.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVNTPS VMOVNTPS — Move Non-Temporal Packed Single-Precision Floating-Point

Moves packed single-precision floating-point values from a register to a memory location.

Indicates to the processor that the data is non-temporal, and is unlikely to be used again soon. The processor treats the store as a write-combining (WC) memory write, which minimizes cache pollution. The method of minimization depends on the hardware implementation of the instruction. For further information, see “Memory Optimization” in Volume 1.

The instruction is weakly-ordered with respect to other instructions that operate on memory. Software should use an SFENCE or MFENCE instruction to force strong memory ordering of MOVNTDQ with respect to other stores.

An attempted store to a non-aligned memory location results in a #GP exception.

There are legacy and extended forms of the instruction:

MOVNTPS

Moves four values.

The source operand is an XMM register. The destination is a 128-bit memory location.

MOVNTPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Moves four values.

The source operand is an XMM register. The destination is a 128-bit memory location.

YMM Encoding

Moves eight values.

The source operand is a YMM register. The destination is a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
MOVNTPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMOVNTPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVNTPS *mem128*, *xmm*	0F 2B /r	Moves four packed double-precision floating-point values from *xmm* to *mem128*, minimizing cache pollution.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVNTPS *mem128, xmm*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.00	2B /r
VMOVNTPS *mem256, ymm*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.00	2B /r

Related Instructions

(V)MOVNTDQ, (V)MOVNTDQA, (V)MOVNTPD, (V)MOVNTQ

Exceptions

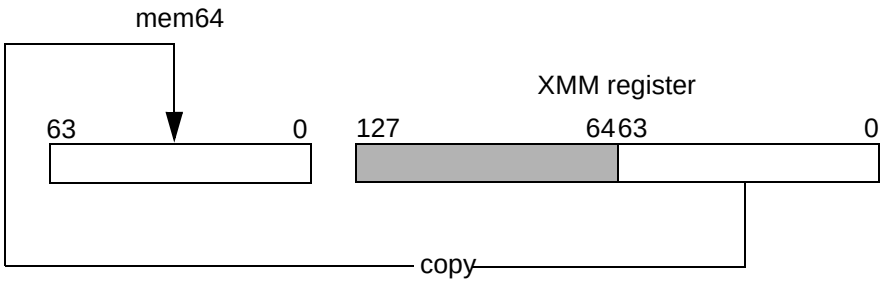
Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not aligned on a 16-byte boundary.
	S	S	X	Write to a read-only data segment.
			A	VEX256: Memory operand not 32-byte aligned. VEX128: Memory operand not 16-byte aligned.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVNTSD Move Non-Temporal Scalar Double-Precision Floating-Point

Stores one double-precision floating-point value from an XMM register to a 64-bit memory location. This instruction indicates to the processor that the data is non-temporal, and is unlikely to be used again soon. The processor treats the store as a write-combining memory write, which minimizes cache pollution.

The diagram below illustrates the operation of this instruction:



Instruction Support

Form	Subset	Feature Flag
MOVNTSD	SSE4A	CPUID Fn8000_0001_ECX[SSE4A] (bit 6)

Software *must* check the CPUID bit once per program or library initialization before using the instruction, or inconsistent behavior may result. For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic		Opcode	Description
MOVNTSD	*mem64*, *xmm*	F2 0F 2B /r	Stores one double-precision floating-point XMM register value into a 64 bit memory location. Treat as a non-temporal store.

Related Instructions

MOVNTDQ, MOVNTI, MOVNTPD, MOVNTPS, MOVNTQ, MOVNTSS

rFLAGS Affected

None

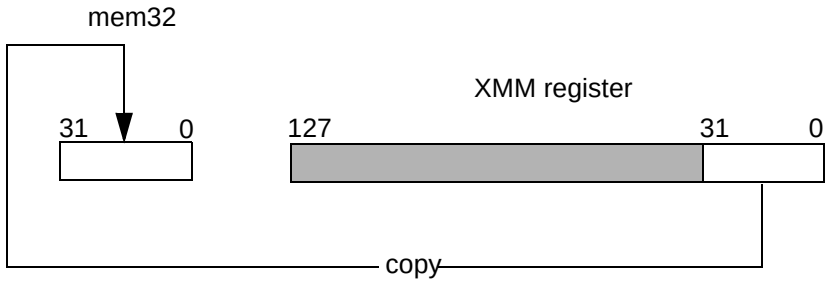
Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SSE4A instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[SSE4A] = 0.
	X	X	X	The emulate bit (CR0.EM) was set to 1.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (CR4.OSFXSR) was cleared to 0.
Device not available, #NM	X	X	X	The task-switch bit (CR0.TS) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
			X	The destination operand was in a non-writable segment.
Page fault, #PF		X	X	A page fault resulted from executing the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MOVNTSS Move Non-Temporal Scalar Single-Precision Floating-Point

Stores one single-precision floating-point value from an XMM register to a 32-bit memory location. This instruction indicates to the processor that the data is non-temporal, and is unlikely to be used again soon. The processor treats the store as a write-combining memory write, which minimizes cache pollution.

The diagram below illustrates the operation of this instruction:



Instruction Support

Form	Subset	Feature Flag
MOVNTSS	SSE4A	CPUID Fn8000_0001_ECX[SSE4A] (bit 6)

Software *must* check the CPUID bit once per program or library initialization before using the instruction, or inconsistent behavior may result. For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic		Opcode	Description
MOVNTSS	*mem32*, *xmm*	F3 0F 2B /r	Stores one single-precision floating-point XMM register value into a 32-bit memory location. Treat as a non-temporal store.

Related Instructions

MOVNTDQ, MOVNTI, MOVNTOPD, MOVNTPS, MOVNTQ, MOVNTSD

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SSE4A instructions are not supported, as indicated by CPUID Fn8000_0001_ECX[SSE4A] = 0.
	X	X	X	The emulate bit (CR0.EM) was set to 1.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (CR4.OSFXSR) was cleared to 0.
Device not available, #NM	X	X	X	The task-switch bit (CR0.TS) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
			X	The destination operand was in a non-writable segment.
Page fault, #PF		X	X	A page fault resulted from executing the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MOVQ VMOVQ Move Quadword

Moves 64-bit values. The source is either the low-order quadword of an XMM register or a 64-bit memory location. The destination is either the low-order quadword of an XMM register or a 64-bit memory location. When the destination is a register, the 64-bit value is zero-extended to 128 bits.

There are legacy and extended forms of the instruction:

MOVQ

There are two encodings:

- The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. The 64-bit value is zero-extended to 128 bits.
- The source operand is an XMM register. The destination is either an XMM register or a 64-bit memory location. When the destination is a register, the 64-bit value is zero-extended to 128 bits.

Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVQ

The extended form of the instruction has three 128-bit encodings:

- The source operand is an XMM register. The destination is an XMM register. The 64-bit value is zero-extended to 128 bits.
- The source operand is a 64-bit memory location. The destination is an XMM register. The 64-bit value is zero-extended to 128 bits.
- The source operand is an XMM register. The destination is either an XMM register or a 64-bit memory location. When the destination is a register, the 64-bit value is zero-extended to 128 bits.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MOVQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMOVQ	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVQ *xmm1, xmm2/mem64*	F3 0F 7E /r	Move a zero-extended 64-bit value from *xmm2* or *mem64* to *xmm1*.
MOVQ *xmm1/mem64*, *xmm2*	66 0F D6 /r	Move a 64-bit value from *xmm2* to *xmm1* or *mem64*. Zero-extends for register destination.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVQ *xmm1*, *xmm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.10	7E /r
VMOVQ *xmm1*, *mem64*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.10	7E /r
VMOVQ *xmm1/mem64*, *xmm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	D6 /r

Related Instructions

(V)MOVD, (V)MOVDQA, (V)MOVDQU

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVSD Move
VMOVSD Scalar Double-Precision Floating-Point

Moves scalar double-precision floating point values. The source is either a low-order quadword of an XMM register or a 64-bit memory location. The destination is either a low-order quadword of an XMM register or a 64-bit memory location.

There are legacy and extended forms of the instruction:

MOVSD

There are two encodings.

- The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. If the source operand is a register, bits [127:64] of the destination are not affected. If the source operand is a 64-bit memory location, the upper 64 bits of the destination are cleared.
- The source operand is an XMM register. The destination is either an XMM register or a 64-bit memory location. When the destination is a register, bits [127:64] of the destination are not affected.

Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVSD

The extended form of the instruction has four 128-bit encodings. Two of the encodings are functionally equivalent.

- The source operand is a 64-bit memory location. The destination is an XMM register. The 64-bit value is zero-extended to 128 bits.
- The source operand is an XMM register. The destination is a 64-bit memory location.
- Two functionally-equivalent encodings:
  There are two source XMM registers. The destination is an XMM register. Bits [127:64] of the first source register are copied to bits [127:64] of the destination; the 64-bit value in bits [63:0] of the second source register is written to bits [63:0] of the destination.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

This instruction must not be confused with the MOVSD (move string doubleword) instruction of the general-purpose instruction set. Assemblers can distinguish the instructions by the number and type of operands.

Instruction Support

Form	Subset	Feature Flag
MOVSD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMOVSD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVSD *xmm1, xmm2/mem64*	F2 0F 10 /r	Moves a 64-bit value from *xmm2 or mem64* to *xmm1*. Zero extends to 128 bits when source operand is memory.
MOVSD *xmm1/mem64*, *xmm2*	F2 0F 11 /r	Moves a 64-bit value from *xmm2* to *xmm1* or *mem64*.

	Encoding [1]			
Mnemonic	**VEX**	**RXB.map_select**	**W.vvvv.L.pp**	**Opcode**
VMOVSD *xmm1*, *mem64*	C4	$\overline{\text{RXB}}$.00001	X.1111.X.11	10 /r
VMOVSD *mem64*, *xmm1*	C4	$\overline{\text{RXB}}$.00001	X.1111.X.11	11 /r
VMOVSD *xmm1, xmm2*, *xmm3* [2]	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.11	10 /r
VMOVSD *xmm1, xmm2*, *xmm3* [2]	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.11	11 /r

Note 1: The addressing mode differentiates between the two operand form (where one operand is a memory location) and the three operand form (where all operands are held in registers).

Note 2: These two encodings are functionally equivalent.

Related Instructions

(V)MOVAPD, (V)MOVHPD, (V)MOVLPD, (V)MOVMSKPD, (V)MOVUPD

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b (for memory destination enoding only).
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVSHDUP VMOVSHDUP — Move High and Duplicate Single-Precision

Moves and duplicates odd-indexed single-precision floating-point values.

There are legacy and extended forms of the instruction:

MOVSHDUP

Moves and duplicates two odd-indexed single-precision floating-point values.

The source operand is an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [127:96] of the source are duplicated and written to bits [127:96] and [95:64] of the destination. Bits [63:32] of the source are duplicated and written to bits [63:32] and [31:0] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVSHDUP

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Moves and duplicates two odd-indexed single-precision floating-point values.

The source operand is an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [127:96] of the source are duplicated and written to bits [127:96] and [95:64] of the destination. Bits [63:32] of the source are duplicated and written to bits [63:32] and [31:0] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Moves and duplicates four odd-indexed single-precision floating-point values.

The source operand is a YMM register or a 256-bit memory location. The destination is a YMM register. Bits [255:224] of the source are duplicated and written to bits [255:224] and [223:192] of the destination. Bits [191:160] of the source are duplicated and written to bits [191:160] and [159:128] of the destination. Bits [127:96] of the source are duplicated and written to bits [127:96] and [95:64] of the destination. Bits [63:32] of the source are duplicated and written to bits [63:32] and [31:0] of the destination.

Instruction Support

Form	Subset	Feature Flag
MOVSHDUP	SSE3	CPUID Fn0000_0001_ECX[SSE3] (bit 0)
VMOVSHDUP	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVSHDUP *xmm1*, *xmm2*/*mem128*	F3 0F 16 /r	Moves and duplicates two odd-indexed single-precision floating-point values in *xmm2* or *mem128*. Writes to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVSHDUP *xmm1*, *xmm2*/*mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.10	16 /r
VMOVSHDUP *ymm1*, *ymm2*/*mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.10	16 /r

Related Instructions

(V)MOVDDUP, (V)MOVSLDUP

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVSLDUP VMOVSLDUP — Move Low and Duplicate Single-Precision

Moves and duplicates even-indexed single-precision floating-point values.

There are legacy and extended forms of the instruction:

MOVSLDUP

Moves and duplicates two even-indexed single-precision floating-point values.

The source operand is an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [95:64] of the source are duplicated and written to bits [127:96] and [95:64] of the destination. Bits [31:0] of the source are duplicated and written to bits [63:32] and [31:0] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVSLDUP

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Moves and duplicates two even-indexed single-precision floating-point values.

The source operand is an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [95:64] of the source are duplicated and written to bits [127:96] and [95:64] of the destination. Bits [31:0] of the source are duplicated and written to bits [63:32] and [31:0] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Moves and duplicates four even-indexed single-precision floating-point values.

The source operand is a YMM register or a 256-bit memory location. The destination is a YMM register. Bits [223:192] of the source are duplicated and written to bits [255:224] and [223:192] of the destination. Bits [159:128] of the source are duplicated and written to bits [191:160] and [159:128] of the destination. Bits [95:64] of the source are duplicated and written to bits [127:96] and [95:64] of the destination. Bits [31:0] of the source are duplicated and written to bits [63:32] and [31:0] of the destination.

Instruction Support

Form	Subset	Feature Flag
MOVSLDUP	SSE3	CPUID Fn0000_0001_ECX[SSE3] (bit 0)
VMOVSLDUP	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVSLDUP *xmm1*, *xmm2*/*mem128*	F3 0F 12 /r	Moves and duplicates two even-indexed single-precision floating-point values in *xmm2* or *mem128*. Writes to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVSLDUP *xmm1*, *xmm2*/*mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.10	12 /r
VMOVSLDUP *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.10	12 /r

Related Instructions

(V)MOVDDUP, (V)MOVSHDUP

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVSS Move
VMOVSS Scalar Single-Precision Floating-Point

Moves scalar single-precision floating point values. The source is either a low-order doubleword of an XMM register or a 32-bit memory location. The destination is either a low-order doubleword of an XMM register or a 32-bit memory location.

There are legacy and extended forms of the instruction:

MOVSS

There are three encodings.

- The source operand is an XMM register. The destination is an XMM register. Bits [127:32] of the destination are not affected.
- The source operand is a 32-bit memory location. The destination is an XMM register. The 32-bit value is zero-extended to 128 bits.
- The source operand is an XMM register. The destination is either an XMM register or a 32-bit memory location. When the destination is a register, bits [127:32] of the destination are not affected.

Bits [255:128] of the YMM register that corresponds to the source are not affected.

VMOVSS

The extended form of the instruction has four 128-bit encodings. Two of the encodings are functionally equivalent.

- The source operand is a 32-bit memory location. The destination is an XMM register. The 32-bit value is zero-extended to 128 bits.
- The source operand is an XMM register. The destination is a 32-bit memory location.
- Two functionally-equivalent encodings:
  There are two source XMM registers. The destination is an XMM register. Bits [127:64] of the first source register are copied to bits [127:64] of the destination; the 32-bit value in bits [31:0] of the second source register is written to bits [31:0] of the destination.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MOVSS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMOVSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVSS *xmm1, xmm2*	F3 0F 10 /r	Moves a 32-bit value from *xmm2* to *xmm1*.
MOVSS *xmm1, mem32*	F3 0F 10 /r	Moves a zero-extended 32-bit value from *mem32* to *xmm1*.
MOVSS *xmm2*/*mem32*, *xmm1*	F3 0F 11 /r	Moves a 32-bit value from *xmm1* to *xmm2* or *mem32.*

Mnemonic	Encoding[1]			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVSS *xmm1*, *mem32*	C4	$\overline{\text{RXB}}$.00001	X.1111.X.10	10 /r
VMOVSS *mem32*, *xmm1*	C4	$\overline{\text{RXB}}$.00001	X.1111.X.10	11 /r
VMOVSS *xmm1, xmm2*, *xmm3* [2]	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.10	10 /r
VMOVSS *xmm1, xmm2*, *xmm3* [2]	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.X.10	11 /r

Note 1: The addressing mode differentiates between the two operand form (where one operand is a memory location) and the three operand form (where all operands are held in registers).

Note 2: These two encodings are functionally equivalent.

Related Instructions

(V)MOVAPS, (V)MOVHLPS, (V)MOVHPS, (V)MOVLHPS, (V)MOVLPS, (V)MOVMSKPS, (V)MOVUPS

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b (for memory destination enoding only).
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVUPD Move Unaligned
VMOVUPD Packed Double-Precision Floating-Point

Moves packed double-precision floating-point values. Values can be moved from a register or memory location to a register; or from a register to a register or memory location.

A memory operand that is not aligned does not cause a general-protection exception.

There are legacy and extended forms of the instruction:

MOVUPD

Moves two double-precision floating-point values. There are encodings for each type of move.

- The source operand is either an XMM register or a 128-bit memory location. The destination operand is an XMM register.
- The source operand is an XMM register. The destination operand is either an XMM register or a 128-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVUPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Moves two double-precision floating-point values. There are encodings for each type of move.

- The source operand is either an XMM register or a 128-bit memory location. The destination operand is an XMM register.
- The source operand is an XMM register. The destination operand is either an XMM register or a 128-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Moves four double-precision floating-point values. There are encodings for each type of move.

- The source operand is either a YMM register or a 256-bit memory location. The destination operand is a YMM register.
- The source operand is a YMM register. The destination operand is either a YMM register or a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
MOVUPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMOVUPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVUPD *xmm1*, *xmm2/mem128*	66 0F 10 /r	Moves two packed double-precision floating-point values from *xmm2* or *mem128* to *xmm1*.
MOVUPD *xmm1/mem128*, *xmm2*	66 0F 11 /r	Moves two packed double-precision floating-point values from *xmm1* or *mem128* to *xmm2*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVUPD *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	10 /r
VMOVUPD *xmm1/mem128*, *xmm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.01	11 /r
VMOVUPD *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.01	10 /r
VMOVUPD *ymm1/mem256*, *ymm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.01	11 /r

Related Instructions

(V)MOVAPD, (V)MOVHPD, (V)MOVLPD, (V)MOVMSKPD, (V)MOVSD

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	X	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MOVUPS VMOVUPS Move Unaligned Packed Single-Precision Floating-Point

Moves packed single-precision floating-point values. Values can be moved from a register or memory location to a register; or from a register to a register or memory location.

A memory operand that is not aligned does not cause a general-protection exception.

There are legacy and extended forms of the instruction:

MOVUPS

Moves four single-precision floating-point values. There are encodings for each type of move.

- The source operand is either an XMM register or a 128-bit memory location. The destination operand is an XMM register.
- The source operand is an XMM register. The destination operand is either an XMM register or a 128-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMOVUPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Moves four single-precision floating-point values. There are encodings for each type of move.

- The source operand is either an XMM register or a 128-bit memory location. The destination operand is an XMM register.
- The source operand is an XMM register. The destination operand is either an XMM register or a 128-bit memory location.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Moves eight single-precision floating-point values. There are encodings for each type of move.

- The source operand is either a YMM register or a 256-bit memory location. The destination operand is a YMM register.
- The source operand is a YMM register. The destination operand is either a YMM register or a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
MOVUPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMOVUPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MOVUPS *xmm1*, *xmm2/mem128*	0F 10 /r	Moves four packed single-precision floating-point values from *xmm2* or unaligned *mem128* to *xmm1*.
MOVUPS *xmm1/mem128*, *xmm2*	0F 11 /r	Moves four packed single-precision floating-point values from *xmm1* or unaligned *mem128* to *xmm2*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVUPS *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.00	10 /r
VMOVUPS *xmm1/mem128*, *xmm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.0.00	11 /r
VMOVUPS *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.00	10 /r
VMOVUPS *ymm1/mem256*, *ymm2*	C4	$\overline{\text{RXB}}$.00001	X.1111.1.00	11 /r

Related Instructions

(V)MOVAPS, (V)MOVHLPS, (V)MOVHPS, (V)MOVLHPS, (V)MOVLPS, (V)MOVMSKPS, (V)MOVSS

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	X	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MPSADBW VMPSADBW Multiple Sum of Absolute Differences

Calculates 8 or 16 sums of absolute differences of sequentially selected groups of four contiguous unsigned byte integers in the first source operand and a selected group of four contiguous unsigned byte integers in a second source operand and writes the eight or sixteen 16-bit unsigned integer sums to sequential words of the destination register. The 256-bit form of the instruction additionally performs a similar but independent calculation using the upper 128 bits of the source operands.

Figure 2-2 on page 238 provides a graphical representation of the operation of the instruction. The following description accompanies it.

The computation uses as inputs 11 bytes from the first source operand and 4 bytes in the second source operand. Bit fields in the imm8 operand specify the index of the right-most byte of each group.

Bits [1:0] of the immediate operand determine the index of the right-most byte of four contiguous bytes within the second source operand used in the operation that produces the result (or, in the case of the 256-bit form of the instruction, the lower 128 bits of the result). Bit 2 of the immediate operand determines the right-most index of the 11contiguous bytes in the first source operand used in the same calculation. In the 128-bit form of the instruction, bits [7:3] of the immediate operand are ignored.

Bits [4:3] of the immediate operand determine the index of the right-most byte of four contiguous bytes within the second source operand used in the operation that produces the upper 128 bits of the result in the 256-bit form of the instruction. Bit 5 of the immediate operand determines the right-most index of the 11 contiguous bytes within in the upper half of the first 256-bit source operand used in the same calculation. In the 256-bit form of the instruction, bits [7:6] of the immediate operand are ignored.

Each word of the destination register receives the result of a separate computation of the sum of absolute differences function applied to a specific pair of four-element vectors derived from the source operands. The sum of absolute differences function SumAbsDiff (A, B) takes as input two 4-element unsigned 8-bit integer vectors and produces a single unsigned 16-bit integer result. The function is defined as:

SumAbsDiff(A, B) = | A[0]-B[0] | + | A[1]-B[1] | + | A[2]-B[2] | + | A[3]-B[3] |

The sum of absolute differences function produces a quantitative measure of the difference between two 4-element vectors. Each of the calculations that generates a result uses this metric to assess the difference between the selected 4-byte vector from operand 2 (B in the above equation) with each of eight overlapping 4-byte vectors (A in the equation) selected sequentially from the first source operand.

The right-most word (Word 0) of the destination receives the result of the comparison of the right-most 4 bytes of the selected group of 11 from operand 1 (src1[i1+3 : i1], as shown in the figure) to the selected 4 bytes from operand 2 (src2[j1+3:j1], in the figure). Word 1 of the destination receives the result of the comparison of the four bytes starting at an offset of 1 from the right-most byte of the group of 11 (src1[i1+4 : i1+1] in the figure) to the 4 bytes from operand 2. Word 2 of the destination receives the result of the comparison of the four bytes starting at an offset of 2 from the right-most byte of the group of 11 (src1[i1+5 : i1+2], in the figure) to the selected 4 bytes from operand 2. This continues in like manner until the left-most four bytes of the 11 are compared to the 4 bytes from operand 2 with the result being written to Word 7. This completes the generation of the lower 128 bits of the result.

The generation of the upper 128 bits of the result for the 256-bit form of the instruction is performed in like manner using separately selected groups of bytes from the upper half of the 256-bit operands, as described above.

The following is a more formal description of the operation of the (V)MPSADBW instruction:

For both the 128-bit and 256-bit form of the instruction, the following set of operations is performed:

```
src1 and src2 are byte vectors that overlay the first and second source operand respectively.
dest is a word vector that overlays the destination register.
tmp1[ ] is an array of 4-element vectors derived from the first source operand.
tmp2 and tmp3 are 4-element vectors derived from the second source operand.

i1 = imm8[2] * 4
j1= imm8[1:0] * 4

tmp1[0] = {src1[i1+3], src1[i1+2], src1[i1+1], src1[i1]}
tmp1[1] = {src1[i1+4], src1[i1+3], src1[i1+2], src1[i1+1]}
tmp1[2] = {src1[i1+5], src1[i1+4], src1[i1+3], src1[i1+2]}
tmp1[3] = {src1[i1+6], src1[i1+5], src1[i1+4], src1[i1+3]}
tmp1[4] = {src1[i1+7], src1[i1+6], src1[i1+5], src1[i1+4]}
tmp1[5] = {src1[i1+8], src1[i1+7], src1[i1+6], src1[i1+5]}
tmp1[6] = {src1[i1+9], src1[i1+8], src1[i1+7], src1[i1+6]}
tmp1[7] = {src1[i1+10], src1[i1+9], src1[i1+8], src1[i1+7]}
tmp2 = {src2[j1+3], src2[j1+2], src2[j1+1], src2[j1]}

dest[0] = SumAbsDiff(tmp1[0], tmp2)
dest[1] = SumAbsDiff(tmp1[1], tmp2)
dest[2] = SumAbsDiff(tmp1[2], tmp2)
dest[3] = SumAbsDiff(tmp1[3], tmp2)
dest[4] = SumAbsDiff(tmp1[4], tmp2)
dest[5] = SumAbsDiff(tmp1[5], tmp2)
dest[6] = SumAbsDiff(tmp1[6], tmp2)
dest[7] = SumAbsDiff(tmp1[7], tmp2)
```

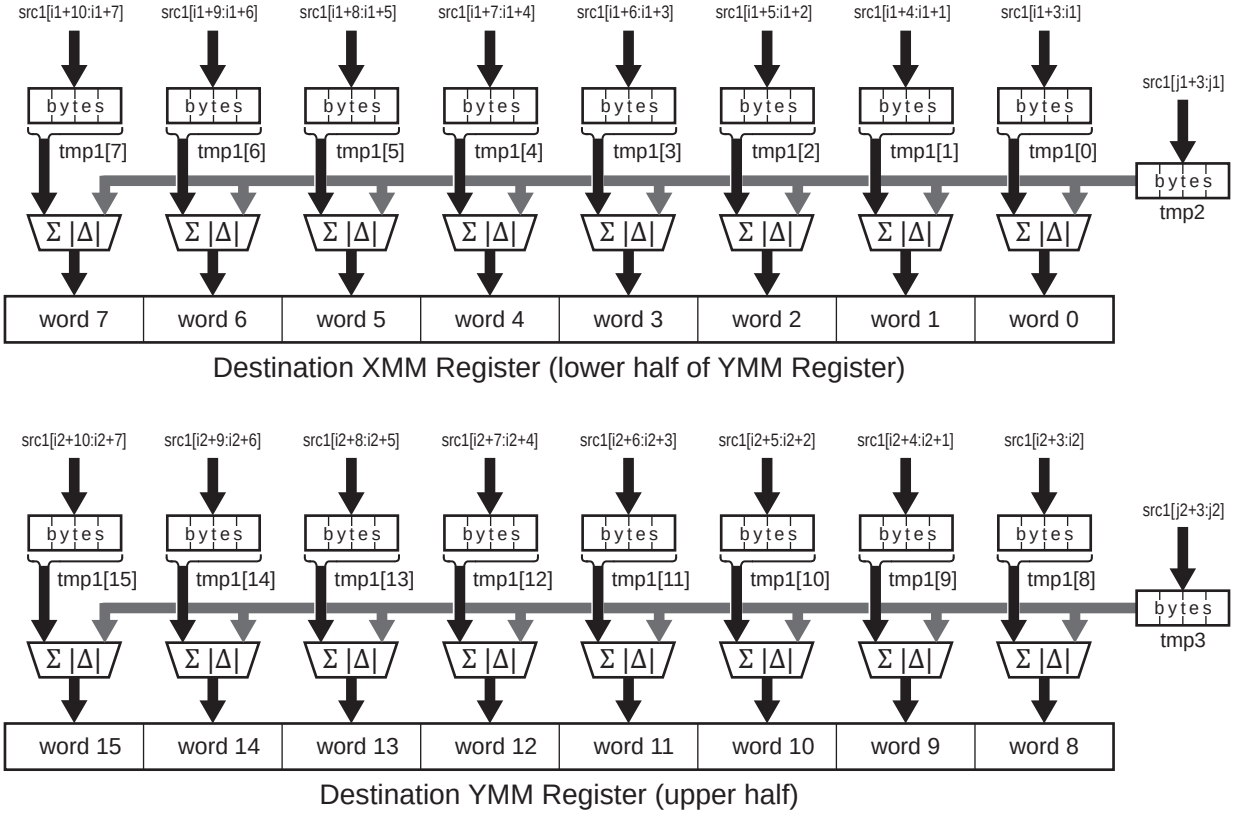
Additionally, for the 256-bit form of the instruction, the following set of operations is performed:

```
i2 = imm8[5] * 4 + 16
j2= imm8[4:3] * 4 +16

tmp1[8] = {src1[i2+3], src1[i2+2], src1[i2+1], src1[i2]}
tmp1[9] = {src1[i2+4], src1[i2+3], src1[i2+2], src1[i2+1]}
tmp1[10] = {src1[i2+5], src1[i2+4], src1[i2+3], src1[i2+2]}
tmp1[11] = {src1[i2+6], src1[i2+5], src1[i2+4], src1[i2+3]}
tmp1[12] = {src1[i2+7], src1[i2+6], src1[i2+5], src1[i2+4]}
tmp1[13] = {src1[i2+8], src1[i2+7], src1[i2+6], src1[i2+5]}
tmp1[14] = {src1[i2+9], src1[i2+8], src1[i2+7], src1[i2+6]}
tmp1[15] = {src1[i2+10], src1[i2+9], src1[i2+8], src1[i2+7]}
tmp3 = {src2[j2+3], src2[j2+2], src2[j2+1], src2[j2]}

dest[8] = SumAbsDiff(tmp1[8], tmp3)
dest[9] = SumAbsDiff(tmp1[9], tmp3)
dest[10] = SumAbsDiff(tmp1[10], tmp3)
dest[11] = SumAbsDiff(tmp1[11], tmp3)
```

dest[12] = SumAbsDiff(tmp1[12], tmp3)
dest[13] = SumAbsDiff(tmp1[13], tmp3)
dest[14] = SumAbsDiff(tmp1[14], tmp3)
dest[15] = SumAbsDiff(tmp1[15], tmp3)



Notes:
- i1 is a byte offset into source operand 1 (i1 = imm8[2] * 4).
- j1 is a byte offset into source operand 2 (j1 = imm8[1:0] * 4)
- i2 is a second byte offset into source operand 1 (i2 = imm8[5] * 4 + 16)
- j2 is a second byte offset into source operand 2 (j2 = imm8[4:3] * 4 + 16)
- Σ |Δ| represents the sum of absolute differences function which operates on two 4-element unsigned packed byte values and produces an unsigned 16-bit integer.

MPSADBW_instruct2.eps

Figure 2-2. (V)MPSADBW Instruction

There are legacy and extended forms of the instruction:

MPSADBW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMPSADBW

The extended form of the instruction has 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register. Bits [127:0] of the destination receive the results of the first 8 sums of absolute differences calculation using the selected bytes of the lower halves of the two source operands. Bits [255:128] of the destination receive the results of the second 8 sums of absolute differences calculation using selected bytes of the upper halves of the two source operands.

Instruction Support

Form	Subset	Feature Flag
MPSADBW	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VMPSADBW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VMPSADBW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MPSADBW *xmm1*, *xmm2/mem128, imm8*	66 0F 3A 42 /r ib	Sums absolute difference of groups of four 8-bit integer in *xmm1 and xmm2 or mem128.* Writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMPSADBW *xmm1*, *xmm2*, *xmm3/mem128, imm8*	C4	$\overline{\text{RXB}}$.03	X.$\overline{src1}$.0.01	42 /r ib
VMPSADBW *ymm1*, *ymm2*, *ymm3/mem256, imm8*	C4	$\overline{\text{RXB}}$.03	X.$\overline{src1}$.1.01	42 /r ib

Related Instructions

(V)PSADBW, (V)PABSB, (V)PABSD, (V)PABSW

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

MULPD VMULPD Multiply Packed Double-Precision Floating-Point

Multiplies each packed double-precision floating-point value of the first source operand by the corresponding packed double-precision floating-point value of the second source operand and writes the product of each multiplication into the corresponding quadword of the destination.

There are legacy and extended forms of the instruction:

MULPD

Multiplies two double-precision floating-point values in the first source XMM register by the corresponding double precision floating-point values in either a second XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMULPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Multiplies two double-precision floating-point values in the first source XMM register by the corresponding double-precision floating-point values in either a second source XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Multiplies four double-precision floating-point values in the first source YMM register by the corresponding double precision floating-point values in either a second source YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
MULPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMULPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MULPD *xmm1, xmm2/mem128*	66 0F 59 /r	Multiplies two packed double-precision floating-point values in *xmm1* by corresponding values in *xmm2* or *mem128*. Writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMULPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src}$.0.01	59 /r
VMULPD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src}$.1.01	59 /r

Related Instructions

(V)MULPS, (V)MULSD, (V)MULSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MULPS Multiply
VMULPS Packed Single-Precision Floating-Point

Multiplies each packed single-precision floating-point value of the first source operand by the corresponding packed single-precision floating-point value of the second source operand and writes the product of each multiplication into the corresponding elements of the destination.

There are legacy and extended forms of the instruction:

MULPS

Multiplies four single-precision floating-point values in the first source XMM register by the corresponding single-precision floating-point values of either a second source XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VMULPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Multiplies four single-precision floating-point values in the first source XMM register by the corresponding single-precision floating-point values of either a second source XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Multiplies eight single-precision floating-point values in the first source YMM register by the corresponding single-precision floating-point values of either a second source YMM register or a 256-bit memory location. Writes the results to a third YMM register.

Instruction Support

Form	Subset	Feature Flag
MULPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMULPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MULPS *xmm1*, *xmm2/mem128*	0F 59 /r	Multiplies four packed single-precision floating-point values in *xmm1* by corresponding values in *xmm2* or *mem128*. Writes the products to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMULPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.00	59 /r
VMULPS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.00	59 /r

Related Instructions

(V)MULPD, (V)MULSD, (V)MULSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MULSD VMULSD Multiply Scalar Double-Precision Floating-Point

Multiplies the double-precision floating-point value in the low-order quadword of the first source operand by the double-precision floating-point value in the low-order quadword of the second source operand and writes the product into the low-order quadword of the destination.

There are legacy and extended forms of the instruction:

MULSD

The first source operand is an XMM register and the second source operand is either an XMM register or a 64-bit memory location. The first source register is also the destination register. Bits [127:64] of the destination and bits [255:128] of the corresponding YMM register are not affected.

VMULSD

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 64-bit memory location. The destination is a third XMM register. Bits [127:64] of the first source operand are copied to bits [127:64] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MULSD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VMULSD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MULSD *xmm1*, *xmm2/mem64*	F2 0F 59 /r	Multiplies low-order double-precision floating-point values in *xmm1* by corresponding values in *xmm2* or *mem64*. Writes the products to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMULSD *xmm1*, *xmm2*, *xmm3*/*mem64*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.X.11	59 /r

Related Instructions

(V)MULPD, (V)MULPS, (V)MULSS

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

MULSS VMULSS Multiply Scalar Single-Precision Floating-Point

Multiplies the single-precision floating-point value in the low-order doubleword of the first source operand by the single-precision floating-point value in the low-order doubleword of the second source operand and writes the product into the low-order doubleword of the destination.

There are legacy and extended forms of the instruction:

MULSS

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location. The first source register is also the destination. Bits [127:32] of the destination register and bits [255:128] of the corresponding YMM register are not affected.

VMULSS

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location. The destination is a third XMM register. Bits [127:32] of the first source register are copied to bits [127:32] of the of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
MULSS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VMULSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
MULSS *xmm1*, *xmm2*/*mem32*	F3 0F 59 /r	Multiplies a single-precision floating-point value in the low-order doubleword of *xmm1* by a corresponding value in *xmm2* or *mem32*. Writes the product to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMULSS *xmm1*, *xmm2*, *xmm3*/*mem32*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.X.10	59 /r

Related Instructions

(V)MULPD, (V)MULPS, (V)MULSD

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

ORPD OR
VORPD Packed Double-Precision Floating-Point

Performs bitwise OR of two packed double-precision floating-point values in the first source operand with the corresponding two packed double-precision floating-point values in the second source operand and writes the results into the corresponding elements of the destination.

There are legacy and extended forms of the instruction:

ORPD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VORPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
ORPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VORPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ORPD *xmm1*, *xmm2/mem128*	66 0F 56 /r	Performs bitwise OR of two packed double-precision floating-point values in *xmm1* with corresponding values in *xmm2* or *mem128*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VORPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	56 /r
VORPD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	56 /r

Related Instructions

(V)ANDNPS, (V)ANDPD, (V)ANDPS, (V)ORPS, (V)XORPD, (V)XORPS

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

ORPS VORPS OR Packed Single-Precision Floating-Point

Performs bitwise OR of the four packed single-precision floating-point values in the first source operand with the corresponding four packed single-precision floating-point values in the second source operand, and writes the result into the corresponding elements of the destination.

There are legacy and extended forms of the instruction:

ORPS

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VORPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
ORPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VORPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ORPS *xmm1*, *xmm2/mem128*	0F 56 /r	Performs bitwise OR of four packed double-precision floating-point values in *xmm1* with corresponding values in *xmm2* or *mem128*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VORPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.00	56 /r
VORPS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.00	56 /r

Related Instructions

(V)ANDNPD, (V)ANDNPS, (V)ANDPD, (V)ANDPS, (V)ORPD, (V)XORPD, (V)XORPS

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PABSB VPABSB — Packed Absolute Value Signed Byte

Computes the absolute value of 16 or 32 packed 8-bit signed integers in the source operand. Each byte of the destination receives an unsigned 8-bit integer that is the absolute value of the signed 8-bit integer in the corresponding byte of the source operand.

There are legacy and extended forms of the instruction:

PABSB

The source operand is an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPABSB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is a YMM register or a 256-bit memory location. The destination is a YMM register. All 32 bytes of the destination are written.

Instruction Support

Form	Subset	Feature Flag
PABSB	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPABSB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPABSB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PABSB *xmm1, xmm2/mem128*	0F 38 1C /r	Computes the absolute value of each packed 8-bit signed integer value in *xmm2/mem128* and writes the 8-bit unsigned results to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPABSB *xmm1, xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	1C /r
VPABSB *ymm1, ymm2/mem256*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	1C /r

Related Instructions

(V)PABSW, (V)PABSD

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PABSD
VPABSD

Packed Absolute Value Signed Doubleword

Computes the absolute value of four or eight packed 32-bit signed integers in the source operand. Each doubleword of the destination receives an unsigned 32-bit integer that is the absolute value of the signed 32-bit integer in the corresponding doubleword of the source operand.

There are legacy and extended forms of the instruction:

PABSD

The source operand is an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPABSD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is a YMM register or a 256-bit memory location. The destination is a YMM register. All four doublewords of the destination are written.

Instruction Support

Form	Subset	Feature Flag
PABSD	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPABSD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPABSD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PABSD *xmm1, xmm2/mem128*	0F 38 1E /r	Computes the absolute value of each packed 32-bit signed integer value in *xmm2/mem128* and writes the 32-bit unsigned results to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPABSD *xmm1, xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	1E /r
VPABSD *ymm1, ymm2/mem256*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	1E /r

Related Instructions

(V)PABSB, (V)PABSW

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PABSW VPABSW Packed Absolute Value Signed Word

Computes the absolute value of eight or sixteen packed 16-bit signed integers in the source operand. Each word of the destination receives an unsigned 16-bit integer that is the absolute value of the signed 16-bit integer in the corresponding word of the source operand.

There are legacy and extended forms of the instruction:

PABSW

The source operand is an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPABSW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is a YMM register or a 256-bit memory location. The destination is a YMM register. All 16 words of the destination are written.

Instruction Support

Form	Subset	Feature Flag
PABSW	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPABSW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPABSW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PABSW *xmm1, xmm2/mem128*	0F 38 1D /r	Computes the absolute value of each packed 16-bit signed integer value in *xmm2/mem128* and writes the 16-bit unsigned results to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPABSW *xmm1, xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	1D /r
VPABSW *ymm1, ymm2/mem256*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	1D /r

Related Instructions

(V)PABSB, (V)PABSD

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PACKSSDW VPACKSSDW — Pack with Signed Saturation Doubleword to Word

Converts four or eight 32-bit signed integers from the first source operand and the second source operand into 16-bit signed integers and packs the results into the destination.

Positive source value greater than 7FFFh are saturated to 7FFFh; negative source values less than 8000h are saturated to 8000h.

Converted values from the first source operand are packed into the low-order words of the destination; converted values from the second source operand are packed into the high-order words of the destination.

There are legacy and extended forms of the instruction:

PACKSSDW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPACKSSDW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PACKSSDW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPACKSSDW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPACKSSDW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PACKSSDW *xmm1*, *xmm2/mem128*	66 0F 6B /r	Converts 32-bit signed integers in *xmm1* and *xmm2* or *mem128* into 16-bit signed integers with saturation. Writes packed results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPACKSSDW *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	6B /r
VPACKSSDW *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	6B /r

Related Instructions

(V)PACKSSWB, (V)PACKUSDW, (V)PACKUSWB

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PACKSSWB Pack with Signed Saturation
VPACKSSWB Word to Byte

Converts eight or sixteen 16-bit signed integers from the first source operand and the second source operand into sixteen or thirty two 8-bit signed integers and packs the results into the destination.

Positive source values greater than 7Fh are saturated to 7Fh; negative source values less than 80h are saturated to 80h.

Converted values from the first source operand are packed into the low-order bytes of the destination; converted values from the second source operand are packed into the high-order bytes of the destination.

There are legacy and extended forms of the instruction:

PACKSSWB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPACKSSWB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PACKSSWB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPACKSSWB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPACKSSWB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PACKSSWB *xmm1*, *xmm2/mem128*	66 0F 63 /r	Converts 16-bit signed integers in *xmm1* and *xmm2* or *mem128* into 8-bit signed integers with saturation. Writes packed results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPACKSSWB *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	63 /r
VPACKSSWB *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	63 /r

Related Instructions

(V)PACKSSDW, (V)PACKUSDW, (V)PACKUSWB

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PACKUSDW VPACKUSDW — Pack with Unsigned Saturation Doubleword to Word

Converts four or eight 32-bit signed integers from the first source operand and the second source operand into eight or sixteen 16-bit unsigned integers and packs the results into the destination.

Source values greater than FFFFh are saturated to FFFFh; source values less than 0000h are saturated to 0000h.

Packs converted values from the first source operand into the low-order words of the destination; packs converted values from the second source operand into the high-order words of the destination.

There are legacy and extended forms of the instruction:

PACKUSDW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPACKUSDW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PACKUSDW	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPACKUSDW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPACKUSDW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PACKUSDW *xmm1*, *xmm2/mem128*	66 0F 38 2B /r	Converts 32-bit signed integers in *xmm1* and *xmm2* or *mem128* into 16-bit unsigned integers with saturation. Writes packed results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPACKUSDW *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	2B /r
VPACKUSDW *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	2B /r

Related Instructions

(V)PACKSSDW, (V)PACKSSWB, (V)PACKUSWB

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PACKUSWB VPACKUSWB — Pack with Unsigned Saturation Word to Byte

Converts eight or sixteen 16-bit signed integers from the first source operand and the second source operand into sixteen or thirty two 8-bit unsigned integers and packs the results into the destination.

When a source value is greater than 7Fh it is saturated to FFh; when source value is less than 00h, it is saturated to 00h.

Packs converted values from the first source operand into the low-order bytes of the destination; packs converted values from the second source operand into the high-order bytes of the destination.

There are legacy and extended forms of the instruction:

PACKUSWB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPACKUSWB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PACKUSWB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPACKUSWB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPACKUSWB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PACKUSWB *xmm1*, *xmm2/mem128*	66 0F 67 /r	Converts 16-bit signed integers in *xmm1* and *xmm2* or *mem128* into 8-bit signed integers with saturation. Writes packed results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPACKUSWB *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	67 /r
VPACKUSWB *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	67 /r

Related Instructions

(V)PACKSSDW, (V)PACKSSWB, (V)PACKUSDW

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PADDB VPADDB — Packed Add Bytes

Adds 16 or 32 packed 8-bit integer values in the first source operand to corresponding values in the second source operand and writes the integer sums to the corresponding bytes of the destination.

This instruction operates on both signed and unsigned integers. When a result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 8 bits of each result are written to the destination.

There are legacy and extended forms of the instruction:

PADDB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPADDB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PADDB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPADDB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPADDB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PADDB *xmm1*, *xmm2/mem128*	66 0F FC /r	Adds packed byte integer values in *xmm1* and *xmm2* or *mem128* Writes the sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPADDB *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	FC /r
VPADDB *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	FC /r

Related Instructions

(V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDSW, (V)PADDUSB, (V)PADDUSW, (V)PADDW

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PADDD VPADDD — Packed Add Doublewords

Adds 4 or 8 packed 32-bit integer value in the first source operand to corresponding values in the second source operand and writes integer sums to the corresponding doublewords of the destination.

This instruction operates on both signed and unsigned integers. When a result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 32 bits of each result are written to the destination.

There are legacy and extended forms of the instruction:

PADDD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPADDD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PADDD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPADDD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPADDD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PADDD *xmm1*, *xmm2/mem128*	66 0F FE /r	Adds packed doubleword integer values in *xmm1* and *xmm2* or *mem128* Writes the sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPADDD *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	FE /r
VPADDD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	FE /r

Related Instructions

(V)PADDB, (V)PADDQ, (V)PADDSB, (V)PADDSW, (V)PADDUSB, (V)PADDUSW, (V)PADDW

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PADDQ VPADDQ Packed Add Quadwords

Adds 2 or 4 packed 64-bit integer values in the first source operand to corresponding values in the second source operand and writes the integer sums to the corresponding quadwords of the destination.

This instruction operates on both signed and unsigned integers. When a result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 64 bits of each result are written to the destination.

There are legacy and extended forms of the instruction:

PADDQ

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPADDQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PADDQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPADDQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPADDQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PADDQ *xmm1*, *xmm2/mem128*	66 0F D4 /r	Adds packed quadword integer values in *xmm1* and *xmm2* or *mem128* Writes the sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPADDQ *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src1}$.0.01	D4 /r
VPADDQ *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src1}$.1.01	D4 /r

Related Instructions

(V)PADDB, (V)PADDD, (V)PADDSB, (V)PADDSW, (V)PADDUSB, (V)PADDUSW, (V)PADDW

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PADDSB VPADDSB — Packed Add with Signed Saturation Bytes

Adds 16 or 32 packed 8-bit signed integer values in the first source operand to the corresponding values in the second source operand and writes the signed integer sums to corresponding bytes of the destination.

Positive sums greater than 7Fh are saturated to 7Fh; negative sums less than 80h are saturated to 80h.

There are legacy and extended forms of the instruction:

PADDSB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPADDSB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PADDSB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPADDSB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPADDSB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PADDSB *xmm1*, *xmm2/mem128*	66 0F EC /r	Adds packed signed 8-bit integer values in *xmm1* and *xmm2* or *mem128* with signed saturation. Writes the sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPADDSB *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	EC /r
VPADDSB *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	EC /r

Related Instructions

(V)PADDB, (V)PADDD, (V)PADDQ, (V)PADDSW, (V)PADDUSB, (V)PADDUSW, (V)PADDW

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PADDSW VPADDSW Packed Add with Signed Saturation Words

Adds 8 or 16 packed 16-bit signed integer value in the first source operand to the corresponding values in the second source operand and writes the signed integer sums to the corresponding words of the destination.

Positive sums greater than 7FFFh are saturated to 7FFFh; negative sums less than 8000h are saturated to 8000h.

There are legacy and extended forms of the instruction:

PADDSW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPADDSW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PADDSW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPADDSW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPADDSW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PADDSW *xmm1*, *xmm2/mem128*	66 0F ED /r	Adds packed signed 16-bit integer values in *xmm1* and *xmm2* or *mem128* with signed saturation. Writes the sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPADDSW *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src1}$.0.01	ED /r
VPADDSW *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src1}$.1.01	ED /r

Related Instructions

(V)PADDB, (V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDUSB, (V)PADDUSW, (V)PADDW

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PADDUSB VPADDUSB Packed Add with Unsigned Saturation Bytes

Adds 16 or 32 packed 8-bit unsigned integer values in the first source operand to the corresponding values in the second source operand and writes the unsigned integer sums to the corresponding bytes of the destination.

Sums greater than FFh are saturated to FFh.

There are legacy and extended forms of the instruction:

PADDUSB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPADDUSB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PADDUSB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPADDUSB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPADDUSB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PADDUSB *xmm1*, *xmm2/mem128*	66 0F DC /r	Adds packed unsigned 8-bit integer values in *xmm1* and *xmm2* or *mem128* with unsigned saturation. Writes the sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPADDUSB *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	DC /r
VPADDUSB *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	DC /r

Related Instructions

(V)PADDB, (V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDSW, (V)PADDUSW, (V)PADDW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PADDUSW VPADDUSW Packed Add with Unsigned Saturation Words

Adds 8 or 16 packed 16-bit unsigned integer value in the first source operand to the corresponding values in the second source operand and writes the unsigned integer sums to the corresponding words of the destination.

Sums greater than FFFFh are saturated to FFFFh.

There are legacy and extended forms of the instruction:

PADDUSW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPADDUSW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PADDUSW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPADDUSW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPADDUSW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PADDUSW *xmm1*, *xmm2/mem128*	66 0F DD /r	Adds packed unsigned 16-bit integer values in *xmm1* and *xmm2* or *mem128* with unsigned saturation. Writes the sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPADDUSW *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	DD /r
VPADDUSW *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	DD /r

Related Instructions

(V)PADDB, (V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDSW, (V)PADDUSB, (V)PADDW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PADDW VPADDW — Packed Add Words

Adds or 16 packed 16-bit integer value in the first source operand to the corresponding values in the second source operand and writes the integer sums to the corresponding word of the destination.

This instruction operates on both signed and unsigned integers. When a result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 16 bits of each result are written to the destination.

There are legacy and extended forms of the instruction:

PADDW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPADDW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PADDW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPADDW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPADDW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PADDW *xmm1*, *xmm2/mem128*	66 0F FD /r	Adds packed 16-bit integer values in *xmm1* and *xmm2* or *mem128*. Writes the sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPADDW *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	FD /r
VPADDW *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	FD /r

Related Instructions

(V)PADDB, (V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDSW, (V)PADDUSB, (V)PADDUSW

RFlags Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PALIGNR Packed Align Right
VPALIGNR

Concatenates one or two pairs of 16-byte values from the first and second source operands and right-shifts the concatenated values the number of bytes specified by the unsigned immediate operand. Writes the least-significant 16 bytes of the shifted result to the destination or writes the least-significant 16 bytes of the two shifted results to the upper and lower halves of the destination.

For the 128-bit form of the instruction, the first and second 128-bit source operands are concatenated to form a temporary 256-bit value with the first source operand occupying the most-significant half of the temporary value. After the right-shift operation, the lower 128 bits of the result are written to the destination.

For the 256-bit form of the instruction, the lower 16 bytes of the first and second source operands are concatenated to form a first temporary 256-bit value with the bytes from the first source operand occupying the most-significant half of the temporary value. The upper 16 bytes of the first and second source operands are concatenated to form a second temporary 256-bit value with the bytes from the first source operand occupying the most-significant half of the second temporary value. Both temporary values are right-shifted the number of bytes specified by the immediate operand. After the right-shift operation, the lower 16 bytes of the first temporary value are written to the lower 128 bits of the destination and the lower 16 bytes of the second temporary value are written to the upper 128 bits of the destination.

The binary value of the immediate operand determines the byte shift value. On each shift the most-significant byte is set to zero. When the byte shift value is greater than 31, the destination is zeroed.

There are two forms of the instruction.

PALIGNR

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPALIGNR

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PALIGNR	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPALIGNR 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPALIGNR 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PALIGNR *xmm1, xmm2/mem128, imm8*	66 0F 3A 0F /r ib	Right-shifts *xmm1:xmm2/mem128 imm8* bytes. Writes shifted result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPALIGNR *xmm1, xmm2, xmm3/mem128, imm8*	C4	$\overline{\text{RXB}}$.03	X.$\overline{\textit{src1}}$.0.01	0F /r ib
VPALIGNR *ymm1, ymm2, ymm3/mem256, imm8*	C4	$\overline{\text{RXB}}$.03	X.$\overline{\textit{src1}}$.1.01	0F /r ib

Related Instructions

None

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PAND Packed AND
VPAND

Performs a bitwise AND of the packed values in the first and second source operands and writes the result to the destination.

There are legacy and extended forms of the instruction:

PAND

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPAND

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PAND	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPAND 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPAND 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PAND *xmm1*, *xmm2/mem128*	66 0F DB /r	Performs bitwise AND of values in *xmm1* and *xmm2* or *mem128*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPAND *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	DB /r
VPAND *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	DB /r

Related Instructions

(V)PANDN, (V)POR, (V)PXOR

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PANDN Packed AND NOT
VPANDN

Generates the ones' complement of the value in the first source operand and performs a bitwise AND of the complement and the value in the second source operand. Writes the result to the destination.

There are legacy and extended forms of the instruction:

PANDN

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPANDN

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PANDN	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPANDN 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPANDN 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PANDN *xmm1*, *xmm2/mem128*	66 0F DF /r	Generates ones' complement of *xmm1*, then performs bitwise AND with value in *xmm2* or *mem128*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPANDN *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src}$.0.01	DF /r
VPANDN *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src}$.1.01	DF /r

Related Instructions

(V)PAND, (V)POR, (V)PXOR

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PAVGB
VPAVGB

Packed Average Unsigned Bytes

Computes the rounded averages of 16 or 32 packed unsigned 8-bit integer values in the first source operand and the corresponding values of the second source operand. Writes each average to the corresponding byte of the destination.

An average is computed by adding pairs of 8-bit integer values in corresponding positions in the two operands, adding 1 to a 9-bit temporary sum, and right-shifting the temporary sum by one bit position.

There are legacy and extended forms of the instruction:

PAVGB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPAVGB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PAVGB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPAVGB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPAVGB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PAVGB *xmm1*, *xmm2/mem128*	66 0F E0 /r	Averages pairs of packed 8-bit unsigned integer values in *xmm1* and *xmm2* or *mem128*. Writes the averages to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPAVGB *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	E0 /r
VPAVGB *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	E0 /r

Related Instructions

PAVGW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PAVGW
VPAVGW
Packed Average Unsigned Words

Computes the rounded average of packed unsigned 16-bit integer values in the first source operand and the corresponding values of the second source operand. Writes each average to the corresponding word of the destination.

An average is computed by adding pairs of 16-bit integer values in corresponding positions in the two operands, adding 1 to a 17-bit temporary sum, and right-shifting the temporary sum by one bit position.

There are legacy and extended forms of the instruction:

PAVGW

The first source operand is an XMM register and the second source operand is an XMM register or 128-bit memory location. The destination is the same XMM register as the first source operand; the upper 128-bits of the corresponding YMM register are not affected.

VPAVGW

The extended form of the instruction has128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PAVGW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPAVGW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPAVGW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PAVGW *xmm1*, *xmm2/mem128*	66 0F E3 /r	Averages pairs of packed 16-bit unsigned integer values in *xmm1* and *xmm2* or *mem128*. Writes the averages to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPAVGW *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	E3 /r
VPAVGW *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	E3 /r

Related Instructions

(V)PAVGB

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PBLENDVB Variable Blend
VPBLENDVB Packed Bytes

Copies packed bytes from either of two sources to a destination, as specified by a mask operand.

The mask is defined by the most significant bit of each byte of the mask operand. The position of a mask bit corresponds to the position of the most significant bit of a copied value.

- When a mask bit = 0, the specified element of the first source is copied to the corresponding position in the destination.
- When a mask bit = 1, the specified element of the second source is copied to the corresponding position in the destination.

There are legacy and extended forms of the instruction:

PBLENDVB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected. The mask operand is the implicit register XMM0.

VPBLENDVB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared. The mask operand is a fourth XMM register selected by bits [7:4] of an immediate byte.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register. The mask operand is a fourth YMM register selected by bits [7:4] of an immediate byte.

Instruction Support

Form	Subset	Feature Flag
PBLENDVB	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPBLENDVB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPBLENDVB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PBLENDVB *xmm1*, *xmm2/mem128*	66 0F 38 10 /r	Selects byte values from *xmm1* or *xmm2/mem128*, depending on the value of corresponding mask bits in XMM0. Writes the selected values to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPBLENDVB *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.0.01	4C /r is4
VPBLENDVB *ymm1, ymm2, ymm3/mem256, ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.1.01	4C /r is4

Related Instructions

(V)BLENDVPD, (V)BLENDVPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PBLENDW VPBLENDW — Blend Packed Words

Copies packed words from either of two sources to a destination, as specified by an immediate 8-bit mask operand. For the 256-bit form, the same 8-bit mask is applied twice; once to select words to be written to the lower 128 bits of the destination and again to select words to be written to the upper 128 bits of the destination.

Each bit of the mask selects a word from one of the source operands based on the position of the word within the operand. Bit 0 of the mask selects the least-significant word (word 0) to be copied, bit 1 selects the next-most significant word (word 1), and so forth. Bit 7 selects word 7 (the most-significant word for 128-bit operands).

For the 256-bit operands, the mask is reused to select words in the upper 128-bits of the source operands to be copied. Bit 0 of the mask selects word 8, bit 1 selects word 9, and so forth. Finally, bit 7 of the mask selects the word from position 15.

- When a mask bit = 0, the specified element of the first source is copied to the corresponding position in the destination.
- When a mask bit = 1, the specified element of the second source is copied to the corresponding position in the destination.

There are legacy and extended forms of the instruction:

PBLENDW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPBLENDW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PBLENDW	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPBLENDW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPBLENDW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PBLENDW *xmm1*, *xmm2/mem128, imm8*	66 0F 3A 0E /r ib	Selects word values from *xmm1* or *xmm2/mem128*, as specified by *imm8*. Writes the selected values to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPBLENDW *xmm1, xmm2, xmm3/mem128, imm8*	C4	$\overline{\text{RXB}}$.03	X.$\overline{src1}$.0.01	0E /r /ib
VPBLENDW *ymm1, ymm2, ymm3/mem256, imm8*	C4	$\overline{\text{RXB}}$.03	X.$\overline{src1}$.1.01	0E /r /ib

Related Instructions

(V)BLENDPD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PCLMULQDQ Carry-less Multiply Quadwords
VPCLMULQDQ

Performs a carry-less multiplication of a selected quadword element of the first source operand by a selected quadword element of the second source operand and writes the product to the destination.

Carry-less multiplication, also known as *binary polynomial multiplication*, is the mathematical operation of computing the product of two operands without generating or propagating carries. It is an essential component of cryptographic processing, and typically requires a large number of cycles.

The instruction provides an efficient means of performing the operation and is particularly useful in implementing the Galois counter mode used in the Advanced Encryption Standard (AES). See Appendix A on page 975 for additional information.

Bits 4 and 0 of an 8-bit immediate byte operand specify which quadword of each source operand to multiply, as follows.

Mnemonic	Imm[0]	Imm[4]	Quadword Operands Selected
(V)PCLMULLQLQDQ	0	0	SRC1[63:0], SRC2[63:0]
(V)PCLMULHQLQDQ	1	0	SRC1[127:64], SRC2[63:0]
(V)PCLMULLQHQDQ	0	1	SRC1[63:0], SRC2[127:64]
(V)PCLMULHQHQDQ	1	1	SRC1[127:64], SRC2[127:64]

Alias mnemonics are provided for the various immediate byte combinations.

There are legacy and extended forms of the instruction:

PCLMULQDQ

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPCLMULQDQ

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PCLMULQDQ	PCLMULQDQ	CPUID Fn0000_0001_ECX[PCLMULQDQ] (bit 1)
VPCLMULQDQ 128	AVX or PCLMULQDQ	CPUID Fn0000_0001_ECX[PCLMULQDQ] (bit 1) or CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPCLMULQDQ 256	VPCLMULQDQ	CPUID Fn0000_0007_ECX[VPCLMULQD]_x0 (bit 10)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCLMULQDQ *xmm1*, *xmm2/mem128, imm8*	66 0F 3A 44 /r ib	Performs carry-less multiplication of a selected quadword element of *xmm1* by a selected quadword element of *xmm2* or *mem128.* Elements are selected by bits 4 and 0 of *imm8.* Writes the product to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCLMULQDQ *xmm1*, *xmm2*, *xmm3/mem128*, *imm8*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.0.01	44 /r ib
VPCLMULQDQ *ymm1*, *ymm2*, *ymm3/mem256*, *imm8*	C4	$\overline{\text{RXB}}$.00011	X.$\overline{src}$.1.01	44 /r ib

Related Instructions

(V)PMULDQ, (V)PMULUDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PCMPEQB VPCMPEQB — Packed Compare Equal Bytes

Compares packed byte values in the first source operand to corresponding values in the second source operand and writes a comparison result to the corresponding byte of the destination.

When values are equal, the result is FFh; when values are not equal, the result is 00h.

There are legacy and extended forms of the instruction:

PCMPEQB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPCMPEQB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PCMPEQB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPCMPEQB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPCMPEQB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCMPEQB *xmm1*, *xmm2/mem128*	66 0F 74 /r	Compares packed bytes in *xmm1* to packed bytes in *xmm2* or *mem128*. Writes results to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMPEQB *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	74 /r
VPCMPEQB *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	74 /r

Related Instructions

(V)PCMPEQD, (V)PCMPEQW, (V)PCMPGTB, (V)PCMPGTD, (V)PCMPGTW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PCMPEQD VPCMPEQD — Packed Compare Equal Doublewords

Compares packed doubleword values in the first source operand to corresponding values in the second source operand and writes a comparison result to the corresponding doubleword of the destination.

When values are equal, the result is FFFFFFFFh; when values are not equal, the result is 00000000h.

There are legacy and extended forms of the instruction:

PCMPEQD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPCMPEQD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PCMPEQD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPCMPEQD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPCMPEQD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCMPEQD *xmm1*, *xmm2/mem128*	66 0F 76 /r	Compares packed doublewords in *xmm1* to packed doublewords in *xmm2* or *mem128*. Writes results to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMPEQD *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	76 /r
VPCMPEQD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	76 /r

Related Instructions

(V)PCMPEQB, (V)PCMPEQW, (V)PCMPGTB, (V)PCMPGTD, (V)PCMPGTW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PCMPEQQ VPCMPEQQ — Packed Compare Equal Quadwords

Compares packed quadword values in the first source operand to corresponding values in the second source operand and writes a comparison result to the corresponding quadword of the destination.

When values are equal, the result is FFFFFFFFFFFFFFFFh; when values are not equal, the result is 0000000000000000h.

There are legacy and extended forms of the instruction:

PCMPEQQ

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPCMPEQQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PCMPEQQ	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPCMPEQQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPCMPEQQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCMPEQQ *xmm1*, *xmm2/mem128*	66 0F 38 29 /r	Compares packed quadwords in *xmm1* to packed quadwords in *xmm2* or *mem128*. Writes results to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMPEQQ *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	29 /r
VPCMPEQQ *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	29 /r

Related Instructions

(V)PCMPEQB, (V)PCMPEQW, (V)PCMPGTB, (V)PCMPGTD, (V)PCMPGTW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PCMPEQW VPCMPEQW — Packed Compare Equal Words

Compares packed word values in the first source operand to corresponding values in the second source operand and writes a comparison result to the corresponding word of the destination.

When values are equal, the result is FFFFh; when values are not equal, the result is 0000h.

There are legacy and extended forms of the instruction:

PCMPEQW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPCMPEQW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PCMPEQW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPCMPEQW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPCMPEQW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCMPEQW *xmm1*, *xmm2/mem128*	66 0F 75 /r	Compares packed words in *xmm1* to packed words in *xmm2* or *mem128*. Writes results to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMPEQW *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	75 /r
VPCMPEQW *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	75 /r

Related Instructions

(V)PCMPEQB, (V)PCMPEQD, (V)PCMPGTB, (V)PCMPGTD, (V)PCMPGTW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PCMPESTRI Packed Compare
VPCMPESTRI Explicit Length Strings Return Index

Compares character string data in the first and second source operands. Comparison operations are carried out as specified by values encoded in the immediate operand. Writes an index to the ECX register.

Source operands are formatted as a packed characters in one of two supported widths: 8 or 16 bits. Characters may be treated as either signed or unsigned values. Each operand has associated with it a separate integer value specifying the length of the string.

The absolute value of the data in the EAX/RAX register represents the length of the character string in the first source operand; the absolute value of the data in the EDX/RDX register represents the length of the character string in the second source operand.

If the absolute value of the data in either register is greater than the maximum string length that fits in 128 bits, the length is set to the maximum: 8, for 16-bit characters, or 16, for 8-bit characters.

The comparison operations between the two operand strings are summarized in an intermediate result—a comparison summary bit vector that is post-processed to produce the final output. Data fields within the immediate byte specify the source data format, comparison type, comparison summary bit vector post-processing, and output option selection.

The index of either the most significant or least significant set bit of the post-processed comparison summary bit vector is returned in ECX. If no bits are set in the post-processed comparison summary bit vector, ECX is set to 16 for source operand strings composed of 8-bit characters or 8 for 16-bit character strings.

See Section 1.5, “String Compare Instructions” for information about source string data format, comparison operations, comparison summary bit vector generation, post-processing, and output selection options.

The rFLAGS are set to indicate the following conditions:

Flag	Condition
CF	Cleared if the comparison summary bit vector is zero; otherwise set.
PF	cleared.
AF	cleared.
ZF	Set if the specified length of the second string is less than the maximum; otherwise cleared.
SF	Set if the specified length of the first string is less than the maximum; otherwise cleared.
OF	Equal to the value of the lsb of the post-processed comparison summary bit vector.

There are legacy and extended forms of the instruction:

PCMPESTRI

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. A result index is written to the ECX register.

VPCMPESTRI

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. A result index is written to the ECX register.

Instruction Support

Form	Subset	Feature Flag
PCMPESTRI	SSE4.2	CPUID Fn0000_0001_ECX[SSE42] (bit 20)
VPCMPESTRI	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCMPESTRI *xmm1*, *xmm2/mem128, imm8*	66 0F 3A 61 /r ib	Compares packed string data in *xmm1* and *xmm2* or *mem128*. Writes a result index to the ECX register.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMPESTRI *xmm1, xmm2/mem128,* imm8	C4	$\overline{\text{RXB}}$.00011	X.1111.0.01	61 /r ib

Related Instructions

(V)PCMPESTRM, (V)PCMPISTRI, (V)PCMPISTRM

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL		OF	DF	IF	TF	SF	ZF	AF	PF	CF
									M				M	M	0	0	M
21	20	19	18	17	16	14	13	12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag that is set or cleared is M (modified). Unaffected flags are blank. Undefined flags are U.*

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PCMPESTRM Packed Compare
VPCMPESTRM Explicit Length Strings Return Mask

Compares character string data in the first and second source operands. Comparison operations are carried out as specified by values encoded in the immediate operand. Writes a mask value to the YMM0/XMM0 register.

Source operands are formatted as a packed characters in one of two supported widths: 8 or 16 bits. Characters may be treated as either signed or unsigned values. Each operand has associated with it a separate integer value specifying the length of the string.

The absolute value of the data in the EAX/RAX register represents the length of the character string in the first source operand; the absolute value of the data in the EDX/RDX register represents the length of the character string in the second source operand.

If the absolute value of the data in either register is greater than the maximum string length that fits in 128 bits, the length is set to the maximum: 8, for 16-bit characters, or 16, for 8-bit characters.

The comparison operations between the two operand strings are summarized in an intermediate result—a comparison summary bit vector that is post-processed to produce the final output. Data fields within the immediate byte specify the source data format, comparison type, comparison summary bit vector post-processing, and output option selection.

Depending on the output option selected, the post-processed comparison summary bit vector is either zero-extended to 128 bits or expanded into a byte/word-mask and then written to XMM0.

See Section 1.5, “String Compare Instructions” for information about source string data format, comparison operations, comparison summary bit vector generation, post-processing, and output selection options.

The rFLAGS are set to indicate the following conditions:

Flag	Condition
CF	Cleared if the comparison summary bit vector is zero; otherwise set.
PF	cleared.
AF	cleared.
ZF	Set if the specified length of the second string is less than the maximum; otherwise cleared.
SF	Set if the specified length of the first string is less than the maximum; otherwise cleared.
OF	Equal to the value of the lsb of the post-processed summary bit vector.

There are legacy and extended forms of the instruction:

PCMPESTRM

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The mask result is written to the XMM0 register.

VPCMPESTRM

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The mask result is written to the XMM0 register. Bits [255:128] of the YMM0 register are cleared.

Instruction Support

Form	Subset	Feature Flag
PCMPESTRM	SSE4.2	CPUID Fn0000_0001_ECX[SSE42] (bit 20)
VPCMPESTRM	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCMPESTRM*xmm1*, *xmm2/mem128, imm8*	66 0F 3A 60 /r ib	Compares packed string data in *xmm1* and *xmm2* or *mem128*. Writes a mask value to the XMM0 register.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMPESTRM *xmm1, xmm2/mem128,* imm8	C4	$\overline{\text{RXB}}$.00011	X.1111.0.01	60 /r ib

Related Instructions

(V)PCMPESTRI, (V)PCMPISTRI, (V)PCMPISTRM

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL		OF	DF	IF	TF	SF	ZF	AF	PF	CF
									M				M	M	0	0	M
21	20	19	18	17	16	14	13	12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PCMPGTB
VPCMPGTB
Packed Compare Greater Than Signed Bytes

Compares packed signed byte values in the first source operand to corresponding values in the second source operand and writes a comparison result to the corresponding byte of the destination.

When a value in the first operand is greater than a value in the second source operand, the result is FFh; when a value in the first operand is less than or equal to a value in the second operand, the result is 00h.

There are legacy and extended forms of the instruction:

PCMPGTB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPCMPGTB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PCMPGTB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPCMPGTB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPCMPGTB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCMPGTB *xmm1*, *xmm2/mem128*	66 0F 64 /r	Compares packed bytes in *xmm1* to packed bytes in *xmm2* or *mem128*. Writes results to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMPGTB *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	64 /r
VPCMPGTB *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	64 /r

Related Instructions

(V)PCMPEQB, (V)PCMPEQD, (V)PCMPEQW, (V)PCMPGTD, (V)PCMPGTW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PCMPGTD VPCMPGTD Packed Compare Greater Than Signed Doublewords

Compares packed signed doubleword values in the first source operand to corresponding values in the second source operand and writes a comparison result to the corresponding doubleword of the destination.

When a value in the first operand is greater than a value in the second operand, the result is FFFFFFFFh; when a value in the first operand is less than or equal to a value in the second operand, the result is 00000000h.

There are legacy and extended forms of the instruction:

PCMPGTD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPCMPGTD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PCMPGTD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPCMPGTD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPCMPGTD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCMPGTD *xmm1*, *xmm2/mem128*	66 0F 66 /r	Compares packed bytes in *xmm1* to packed bytes in *xmm2* or *mem128*. Writes results to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMPGTD *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	66 /r
VPCMPGTD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	66 /r

Related Instructions

(V)PCMPEQB, (V)PCMPEQD, (V)PCMPEQW, (V)PCMPGTB, (V)PCMPGTW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PCMPGTQ VPCMPGTQ — Packed Compare Greater Than Signed Quadwords

Compares packed signed quadword values in the first source operand to corresponding values in the second source operand and writes a comparison result to the corresponding quadword of the destination.

When a value in the first operand is greater than a value in the second operand, the result is FFFFFFFFFFFFFFFFh; when a value in the first operand is less than or equal to a value in the second operand, the result is 0000000000000000h.

There are legacy and extended forms of the instruction:

PCMPGTQ

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPCMPGTQ

The extended form of the instruction has 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PCMPGTQ	SSE4.2	CPUID Fn0000_0001_ECX[SSE42] (bit 20)
VPCMPGTQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPCMPGTQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCMPGTQ *xmm1*, *xmm2/mem128*	66 0F 38 37 /r	Compares packed bytes in *xmm1* to packed bytes in *xmm2* or *mem128*. Writes results to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMPGTQ *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	37 /r
VPCMPGTQ *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	37 /r

Related Instructions

(V)PCMPEQB, (V)PCMPEQD, (V)PCMPEQW, (V)PCMPGTB, (V)PCMPGTW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PCMPGTW Packed Compare Greater Than Signed Words
VPCMPGTW

Compares packed signed word values in the first source operand to corresponding values in the second source operand and writes a comparison result to the corresponding word of the destination.

When a value in the first operand is greater than a value in the second operand, the result is FFFFh; when a value in the first operand is less than or equal to a value in the second operand, the result is 0000h.

There are legacy and extended forms of the instruction:

PCMPGTW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPCMPGTW

The extended form of the instruction has 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PCMPGTW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPCMPGTW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPCMPGTW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCMPGTW *xmm1*, *xmm2/mem128*	66 0F 65 /r	Compares packed bytes in *xmm1* to packed bytes in *xmm2* or *mem128*. Writes results to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMPGTW *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	65 /r
VPCMPGTW *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	65 /r

Related Instructions

(V)PCMPEQB, (V)PCMPEQD, (V)PCMPEQW, (V)PCMPGTB, (V)PCMPGTD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PCMPISTRI Packed Compare
VPCMPISTRI Implicit Length Strings Return Index

Compares character string data in the first and second source operands. Comparison operations are carried out as specified by values encoded in the immediate operand. Writes an index to the ECX register.

Source operands are formatted as a packed characters in one of two supported widths: 8 or 16 bits. Characters may be treated as either signed or unsigned values.

Source operand strings shorter than the maximum that can be packed into a 128-bit value are terminated by a null character (value of 0). The characters prior to the null character constitute the string. If the first (lowest indexed) character is null, the string length is 0.

The comparison operations between the two operand strings are summarized in an intermediate result—a comparison summary bit vector that is post-processed to produce the final output. Data fields within the immediate byte specify the source data format, comparison type, comparison summary bit vector post-processing, and output option selection.

The index of either the most significant or least significant set bit of the post-processed comparison summary bit vector is returned in ECX. If no bits are set in the post-processed comparison summary bit vector, ECX is set to 16 for source operand strings composed of 8-bit characters or 8 for 16-bit character strings.

See Section 1.5, “String Compare Instructions” for information about source string data format, comparison operations, comparison summary bit vector generation, post-processing, and output selection options.

The rFLAGS are set to indicate the following conditions:

Flag	Condition
CF	Cleared if the comparison summary bit vector is zero; otherwise set.
PF	cleared.
AF	cleared.
ZF	Set if any byte (word) in the second operand is null; otherwise cleared.
SF	Set if any byte (word) in the first operand is null; otherwise cleared
OF	Equal to the value of the lsb of the post-processed summary bit vector.

There are legacy and extended forms of the instruction:

PCMPISTRI

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. A result index is written to the ECX register.

VPCMPISTRI

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. A result index is written to the ECX register.

Instruction Support

Form	Subset	Feature Flag
PCMPISTRI	SSE4.2	CPUID Fn0000_0001_ECX[SSE42] (bit 20)
VPCMPISTRI	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCMPISTRI *xmm1*, *xmm2/mem128, imm8*	66 0F 3A 63 /r ib	Compares packed string data in *xmm1* and *xmm2* or *mem128*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMPISTRI *xmm1, xmm2/mem128,* imm8	C4	$\overline{\text{RXB}}$.03	X.1111.0.01	63 /r ib

Related Instructions

(V)PCMPESTRI, (V)PCMPESTRM, (V)PCMPISTRM

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL		OF	DF	IF	TF	SF	ZF	AF	PF	CF
									M				M	M	0	0	M
21	20	19	18	17	16	14	13	12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag that is set or cleared is M (modified). Unaffected flags are blank. Undefined flags are U.*

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PCMPISTRM VPCMPISTRM — Packed Compare Implicit Length Strings Return Mask

Compares character string data in the first and second source operands. Comparison operations are carried out as specified by values encoded in the immediate operand. Writes a mask value to the YMM0/XMM0 register

Source operands are formatted as a packed characters in one of two supported widths: 8 or 16 bits. Characters may be treated as either signed or unsigned values.

Source operand strings shorter than the maximum that can be packed into a 128-bit value are terminated by a null character (value of 0). The characters prior to the null character constitute the string. If the first (lowest indexed) character is null, the string length is 0.

The comparison operations between the two operand strings are summarized in an intermediate result—a comparison summary bit vector that is post-processed to produce the final output. Data fields within the immediate byte specify the source data format, comparison type, comparison summary bit vector post-processing, and output option selection.

Depending on the output option selected, the post-processed comparison summary bit vector is either zero-extended to 128 bits or expanded into a byte/word-mask and then written to XMM0.

See Section 1.5, “String Compare Instructions” for information about source string data format, comparison operations, comparison summary bit vector generation, post-processing, and output selection options.

The rFLAGS are set to indicate the following conditions:

Flag	Condition
CF	Cleared if the comparison summary bit vector is zero; otherwise set.
PF	cleared.
AF	cleared.
ZF	Set if any byte (word) in the second operand is null; otherwise cleared.
SF	Set if any byte (word) in the first operand is null; otherwise cleared.
OF	Equal to the value of the lsb of the post-processed summary bit vector.

There are legacy and extended forms of the instruction:

PCMPISTRM

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The mask result is written to the XMM0 register.

VPCMPISTRM

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The mask result is written to the XMM0 register. Bits [255:128] of the YMM0 register are cleared.

Instruction Support

Form	Subset	Feature Flag
PCMPISTRM	SSE4.2	CPUID Fn0000_0001_ECX[SSE42] (bit 20)
VPCMPISTRM	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PCMPISTRM *xmm1*, *xmm2/mem128, imm8*	66 0F 3A 62 /r ib	Compares packed string data in *xmm1* and *xmm2* or *mem128*. Writes a result or mask to the XMM0 register.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMPISTRM *xmm1, xmm2/mem128,* imm8	C4	$\overline{\text{RXB}}$.03	X.1111.0.01	62 /r ib

Related Instructions

(V)PCMPESTRI, (V)PCMPESTRM, (V)PCMPISTRI

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL		OF	DF	IF	TF	SF	ZF	AF	PF	CF
									M				M	M	0	0	M
21	20	19	18	17	16	14	13	12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag that is set or cleared is M (modified). Unaffected flags are blank. Undefined flags are U.*

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PEXTRB Extract
VPEXTRB Packed Byte

Extracts a byte from a source register and writes it to an 8-bit memory location or to the low-order byte of a general-purpose register, with zero-extension to 32 or 64 bits. Bits [3:0] of an immediate byte operand select the byte to be extracted:

Value of imm8 [3:0]	Source Bits Extracted
0000	[7:0]
0001	[15:8]
0010	[23:16]
0011	[31:24]
0100	[39:32]
0101	[47:40]
0110	[55:48]
0111	[63:56]
1000	[71:64]
1001	[79:72]
1010	[87:80]
1011	[95:88]
1100	[103:96]
1101	[111:104]
1110	[119:112]
1111	[127:120]

There are legacy and extended forms of the instruction:

PEXTRB

The source operand is an XMM register and the destination is either an 8-bit memory location or the low-order byte of a general-purpose register. When the destination is a general-purpose register, the extracted byte is zero-extended to 32 or 64 bits.

VPEXTRB

The extended form of the instruction has a 128-bit encoding only.

The source operand is an XMM register and the destination is either an 8-bit memory location or the low-order byte of a general-purpose register. When the destination is a general-purpose register, the extracted byte is zero-extended to 32 or 64 bits.

Instruction Support

Form	Subset	Feature Flag
PEXTRB	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPEXTRB	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PEXTRB *reg/m8*, *xmm*, *imm8*	66 0F 3A 14 /r ib	Extracts an 8-bit value specified by *imm8* from *xmm* and writes it to *m8* or the low-order byte of a general-purpose register, with zero-extension.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPEXTRB *reg/mem8*, *xmm, imm8*	C4	$\overline{\text{RXB}}$.03	X.1111.0.01	14 /r ib

Related Instructions

(V)PEXTRD, (V)PEXTRW, (V)PEXTRQ, (V)PINSRB, (V)PINSRD, (V)PINSRW, (V)PINSRQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PEXTRD Extract
VPEXTRD Packed Doubleword

Extracts a doubleword from a source register and writes it to an 32-bit memory location or a 32-bit general-purpose register. Bits [1:0] of an immediate byte operand select the doubleword to be extracted:

Value of imm8 [1:0]	Source Bits Extracted
00	[31:0]
01	[63:32]
10	[95:64]
11	[127:96]

There are legacy and extended forms of the instruction:

PEXTRD

The encoding is the same as PEXTRQ, with REX.W = 0.

The source operand is an XMM register and the destination is either an 32-bit memory location or a 32-bit general-purpose register.

VPEXTRD

The extended form of the instruction has a 128-bit encoding only.

The encoding is the same as VPEXTRQ, with VEX.W = 0.

The source operand is an XMM register and the destination is either an 32-bit memory location or a 32-bit general-purpose register.

Instruction Support

Form	Subset	Feature Flag
PEXTRD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPEXTRD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PEXTRD *reg32/mem32*, *xmm*, *imm8*	66 (W0) 0F 3A 16 /r ib	Extracts a 32-bit value specified by *imm8* from *xmm* and writes it to *mem32* or *reg32*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPEXTRD *reg32/mem32*, *xmm, imm8*	C4	$\overline{\text{RXB}}$.03	0.1111.0.01	16 /r ib

Related Instructions

(V)PEXTRB, (V)PEXTRW, (V)PEXTRQ, (V)PINSRB, (V)PINSRD, (V)PINSRW, (V)PINSRQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PEXTRQ Extract
VPEXTRQ Packed Quadword

Extracts a quadword from a source register and writes it to an 64-bit memory location or to a 64-bit general-purpose register. Bit [0] of an immediate byte operand selects the quadword to be extracted:

Value of imm8 [0]	Source Bits Extracted
0	[63:0]
1	[127:64]

There are legacy and extended forms of the instruction:

PEXTRQ

The encoding is the same as PEXTRD, with REX.W = 1.

The source operand is an XMM register and the destination is either an 64-bit memory location or a 64-bit general-purpose register.

VPEXTRQ

The extended form of the instruction has a 128-bit encoding only.

The encoding is the same as VPEXTRD, with VEX.W = 1.

The source operand is an XMM register and the destination is either an 64-bit memory location or a 64-bit general-purpose register.

Instruction Support

Form	Subset	Feature Flag
PEXTRD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPEXTRD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PEXTRQ *reg64/mem64*, *xmm*, *imm8*	66 (W1) 0F 3A 16 /r ib	Extracts a 64-bit value specified by *imm8* from *xmm* and writes it to *mem64* or *reg64*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPEXTRQ *reg64/mem64*, *xmm, imm8*	C4	$\overline{\text{RXB}}$.03	1.1111.0.01	16 /r ib

Related Instructions

(V)PEXTRB, (V)PEXTRD, (V)PEXTRW, (V)PINSRB, (V)PINSRD, (V)PINSRW, (V)PINSRQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PEXTRW
VPEXTRW
Extract Packed Word

Extracts a word from a source register and writes it to a 16-bit memory location or to the low-order word of a general-purpose register, with zero-extension to 32 or 64 bits. Bits [3:0] of an immediate byte operand select the word to be extracted:

Value of imm8 [2:0]	Source Bits Extracted
000	[15:0]
001	[31:16]
010	[47:32
011	[63:48]
100	[79:64]
101	[95:80]
110	[111:96]
111	[127:112]

There are legacy and extended forms of the instruction:

PEXTRW

The legacy form of the instruction has SSE2 and SSE4.1 encodings.

The source operand is an XMM register and the destination is the low-order word of a general-purpose register. The extracted word is zero-extended to 32 or 64 bits.

The source operand is an XMM register and the destination is either an 16-bit memory location or the low-order word of a general-purpose register. When the destination is a general-purpose register, the extracted word is zero-extended to 32 or 64 bits.

VPEXTRW

The extended form of the instruction has two 128-bit encodings that correspond to the two legacy encodings.

The source operand is an XMM register and the destination is the low-order word of a general-purpose register. The extracted word is zero-extended to 32 or 64 bits.

The source operand is an XMM register and the destination is either an 16-bit memory location or the low-order word of a general-purpose register. When the destination is a general-purpose register, the extracted word is zero-extended to 32 or 64 bits.

Instruction Support

Form	Subset	Feature Flag
PEXTRW reg	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
PEXTRW reg/mem16	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPEXTRW	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PEXTRW *reg*, *xmm*, *imm8*	66 0F C5 /r ib	Extracts a 16-bit value specified by *imm8* from *xmm* and writes it to the low-order byte of a general-purpose register, with zero-extension.
PEXTRW *reg/m16*, *xmm*, *imm8*	66 0F 3A 15 /r ib	Extracts a 16-bit value specified by *imm8* from *xmm* and writes it to *m16* or the low-order byte of a general-purpose register, with zero-extension.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPEXTRW *reg*, *xmm, imm8*	C4	$\overline{\text{RXB}}$.01	X.1111.0.01	C5 /r ib
VPEXTRW *reg/mem16*, *xmm, imm8*	C4	$\overline{\text{RXB}}$.03	X.1111.0.01	15 /r ib

Related Instructions

(V)PEXTRB, (V)PEXTRD, (V)PEXTRQ, (V)PINSRB, (V)PINSRD, (V)PINSRW, (V)PINSRQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PHADDD Packed Horizontal Add
VPHADDD Doubleword

Adds adjacent 32-bit signed integers in each of two source operands and packs the sums into the destination. If a sum overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set) and only the low-order 32 bits of the sum are written in the destination.

Adds the 32-bit signed integer values in bits [63:32] and bits [31:0] of the first source operand and packs the sum into bits [31:0] of the destination; adds the 32-bit signed integer values in bits [127:96] and bits [95:64] of the first source operand and packs the sum into bits [63:32] of the destination. Adds the corresponding values in the second source operand and packs the sums into bits [95:64] and [127:96] of the destination.

Additionally, for the 256-bit form, adds the 32-bit signed integer values in bits [191:160] and bits [159:128] of the first source operand and packs the sum into bits [159:128] of the destination; adds the 32-bit signed integer values in bits [255:224] and bits [223:192] of the first source operand and packs the sum into bits [191:160] of the destination. Adds the corresponding values in the second source operand and packs the sums into bits [223:192] and [255:224] of the destination.

There are legacy and extended forms of the instruction:

PHADDD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination register. Bits [255:128] of the YMM register that corresponds to the destination not affected.

VPHADDD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PHADDD	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPHADDD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPHADDD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PHADDD *xmm1*, *xmm2/mem128*	66 0F 38 02 /r	Adds adjacent pairs of signed integers in *xmm1* and *xmm2* or *mem128*. Writes packed sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	02 /r
VPHADDD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	02 /r

Related Instructions

(V)PHADDW, (V)PHADDSW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PHADDSW VPHADDSW Packed Horizontal Add with Saturation Word

Adds adjacent 16-bit signed integers in each of two source operands, with saturation, and packs the 16-bit signed sums into the destination.

Positive sums greater than 7FFFh are saturated to 7FFFh; negative sums less than 8000h are saturated to 8000h.

For the 128-bit form of the instruction, the following operations are performed:

```
dest is the destination register – either an XMM register or the corresponding YMM register.
src1 is the first source operand. src2 is the second source operand.
Ssum() is a function that returns the saturated 16-bit signed sum of its arguments.

dest[15:0] = Ssum(src1[31:16], src1[15:0])
dest[31:16] = Ssum(src1[63:48], src1[47:32])
dest[47:32] = Ssum(src1[95:80], src1[79:64])
dest[63:48] = Ssum(src1[127:112], src1[111:96])
dest[79:64] = Ssum(src2[31:16], src2[15:0])
dest[95:80] = Ssum(src2[63:48], src2[47:32])
dest[111:96] = Ssum(src2[95:80], src2[79:64])
dest[127:112] = Ssum(src2[127:112], src2[111:96])
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[143:128] = Ssum(src1[159:144], src1[143:128])
dest[159:144] = Ssum(src1[191:176], src1[175:160])
dest[175:160] = Ssum(src1[223:208], src1[207:192])
dest[191:176] = Ssum(src1[255:240], src1[239:224])
dest[207:192] = Ssum(src2[159:144], src2[143:128])
dest[223:208] = Ssum(src2[191:176], src2[175:160])
dest[239:224] = Ssum(src2[223:208], src2[207:192])
dest[255:240] = Ssum(src2[255:240], src2[239:224])
```

There are legacy and extended forms of the instruction:

PHADDSW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPHADDSW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PHADDSW	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPHADDSW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPHADDSW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PHADDSW *xmm1*, *xmm2/mem128*	66 0F 38 03 /r	Adds adjacent pairs of signed integers in *xmm1* and *xmm2* or *mem128*, with saturation. Writes packed sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDSW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	03 /r
VPHADDSW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	03 /r

Related Instructions

(V)PHADDD, (V)PHADDW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PHADDW VPHADDW — Packed Horizontal Add Word

Adds adjacent 16-bit signed integers in each of two source operands and packs the 16-bit sums into the destination. If a sum overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set).

For the 128-bit form of the instruction, the following operations are performed:

dest is the destination register – either an XMM register or the corresponding YMM register.
src1 is the first source operand. src2 is the second source operand.

```
dest[15:0] = src1[31:16] + src1[15:0]
dest[31:16] = src1[63:48] + src1[47:32]
dest[47:32] = src1[95:80] + src1[79:64]
dest[63:48] = src1[127:112] + src1[111:96]
dest[79:64] = src2[31:16] + src2[15:0]
dest[95:80] = src2[63:48] + src2[47:32]
dest[111:96] = src2[95:80] + src2[79:64]
dest[127:112] = src2[127:112] + src2[111:96]
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[143:128] = src1[159:144] + src1[143:128]
dest[159:144] = src1[191:176] + src1[175:160]
dest[175:160] = src1[223:208] + src1[207:192]
dest[191:176] = src1[255:240] + src1[239:224]
dest[207:192] = src2[159:144] + src2[143:128]
dest[223:208] = src2[191:176] + src2[175:160]
dest[239:224] = src2[223:208] + src2[207:192]
dest[255:240] = src2[255:240] + src2[239:224]
```

There are legacy and extended forms of the instruction:

PHADDW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPHADDW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PHADDW	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPHADDW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPHADDW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PHADDW *xmm1*, *xmm2/mem128*	66 0F 38 01 /r	Adds adjacent pairs of signed integers in *xmm1* and *xmm2* or *mem128*. Writes packed sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.0.01	01 /r
VPHADDW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.1.01	01 /r

Related Instructions

(V)PHADDD, (V)PHADDSW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PHMINPOSUW VPHMINPOSUW Horizontal Minimum and Position

Finds the minimum unsigned 16-bit value in the source operand and copies it to the low order word element of the destination. Writes the source position index of the value to bits [18:16] of the destination and clears bits[127:19] of the destination.

There are legacy and extended forms of the instruction:

PHMINPOSUW

The source operand is an XMM register or 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPHMINPOSUW

The extended form of the instruction has a 128-bit encoding only.

The source operand is an XMM register or 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
PHMINPOSUW	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPHMINPOSUW	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PHMINPOSUW *xmm1*, *xmm2/mem128*	66 0F 38 41 /r	Finds the minimum unsigned word element in *xmm2* or *mem128*, copies it to *xmm1[15:0]*; writes its position index to *xmm1[18:16]*, and clears *xmm1[127:19]*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPHMINPOSUW *xmm1, xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	41 /r

Related Instructions

(V)PMINSB, (V)PMINSD, (V)PMINSW, (V)PMINUB, (V)PMINUD, (V)PMINUW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PHSUBD VPHSUBD Packed Horizontal Subtract Doubleword

Subtracts adjacent 32-bit signed integers in each of two source operands and packs the differences into the destination. The higher-order doubleword of each pair is subtracted from the lower-order doubleword.

Subtracts the 32-bit signed integer value in bits [63:32] of the first source operand from the 32-bit signed integer value in bits [31:0] of the first source operand and packs the difference into bits [31:0] of the destination; subtracts the 32-bit signed integer value in bits [127:96] of the first source operand from the 32-bit signed integer value in bits [95:64] of the first source operand and packs the difference into bits [63:32] of the destination. Performs the corresponding operations on pairs of 32-bit signed integer values in the second source operand and packs the differences into bits [95:64] and [127:96] of the destination.

Additionally, for the 256-bit form, subtracts the 32-bit signed integer value in bits [191:160] of the first source operand from the 32-bit signed integer value in bits [159:128] of the first source operand and packs the difference into bits [159:128] of the destination; subtracts the 32-bit signed integer value in bits [255:224] of the first source operand from the 32-bit integer value in bits [223:192] of the first source operand and packs the difference into bits [191:160] of the destination. Performs the corresponding operations on pairs of 32-bit signed integer values in the second source operand and packs the differences into bits [223:192] and [255:224] of the destination.

There are legacy and extended forms of the instruction:

PHSUBD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPHSUBD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PHSUBD	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPHSUBD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPHSUBD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PHSUBD *xmm1*, *xmm2/mem128*	66 0F 38 06 /r	Subtracts adjacent pairs of signed integers in *xmm1* and *xmm2* or *mem128*. Writes packed differences to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPHSUBD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.0.01	06 /r
VPHSUBD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.1.01	06 /r

Related Instructions

(V)PHSUBW, (V)PHSUBSW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PHSUBSW VPHSUBSW Packed Horizontal Subtract with Saturation Word

Subtracts adjacent 16-bit signed integers in each of two source operands, with saturation, and packs the differences into the destination. The higher-order word of each pair is subtracted from the lower-order word.

Positive differences greater than 7FFFh are saturated to 7FFFh; negative differences less than 8000h are saturated to 8000h.

For the 128-bit form of the instruction, the following operations are performed:

dest is the destination register – either an XMM register or the corresponding YMM register.
src1 is the first source operand. src2 is the second source operand.
Sdiff(A,B) is a function that returns the saturated 16-bit signed difference A − B.

```
dest[15:0] = Sdiff(src1[15:0], src1[31:16])
dest[31:16] = Sdiff(src1[47:32], src1[63:48])
dest[47:32] = Sdiff(src1[79:64], src1[95:80])
dest[63:48] = Sdiff(src1[111:96], src1[127:112])
dest[79:64] = Sdiff(src2[15:0], src2[31:16])
dest[95:80] = Sdiff(src2[47:32], src2[63:48])
dest[111:96] = Sdiff(src2[79:64], src2[95:80])
dest[127:112] = Sdiff(src2[111:96], src2[127:112])
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[143:128] = Sdiff(src1[143:128], src1[159:144])
dest[159:144] = Sdiff(src1[175:160], src1[191:176])
dest[175:160] = Sdiff(src1[207:192], src1[223:208])
dest[191:176] = Sdiff(src1[239:224], src1[255:240])
dest[207:192] = Sdiff(src2[143:128], src2[159:144])
dest[223:208] = Sdiff(src2[175:160], src2[191:176])
dest[239:224] = Sdiff(src2[207:192], src2[223:208])
dest[255:240] = Sdiff(src2[239:224], src2[255:240])
```

There are legacy and extended forms of the instruction:

PHSUBSW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPHSUBSW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PHSUBSW	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPHSUBSW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPHSUBSW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PHSUBSW *xmm1*, *xmm2/mem128*	66 0F 38 07 /r	Subtracts adjacent pairs of signed integers in *xmm1* and *xmm2* or *mem128*, with saturation. Writes packed differences to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPHSUBSW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.0.01	07 /r
VPHSUBSW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.1.01	07 /r

Related Instructions

(V)PHSUBD, (V)PHSUBW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PHSUBW VPHSUBW — Packed Horizontal Subtract Word

Subtracts adjacent 16-bit signed integers in each of two source operands and packs the differences into a destination. The higher-order word of each pair is subtracted from the lower-order word.

For the 128-bit form of the instruction, the following operations are performed:

dest is the destination register – either an XMM register or the corresponding YMM register.
src1 is the first source operand. src2 is the second source operand.

```
dest[15:0] = src1[15:0] − src1[31:16
dest[31:16] = src1[47:32] − src1[63:48]
dest[47:32] = src1[79:64] − src1[95:80]
dest[63:48] = src1[111:96] − src1[127:112]
dest[79:64] = src2[15:0] − src2[31:16]
dest[95:80] = src2[47:32] − src2[63:48]
dest[111:96] = src2[79:64] − src2[95:80]
dest[127:112] = src2[111:96] − src2[127:112]
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[143:128] = src1[143:128] − src1[159:144]
dest[159:144] = src1[175:160] − src1[191:176]
dest[175:160] = src1[207:192] − src1[223:208]
dest[191:176] = src1[239:224] − src1[255:240]
dest[207:192] = src2[143:128] − src2[159:144]
dest[223:208] = src2[175:160] − src2[191:176]
dest[239:224] = src2[207:192] − src2[223:208]
dest[255:240] = src2[239:224] − src2[255:240]
```

There are legacy and extended forms of the instruction:

PHSUBW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPHSUBW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PHSUBW	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPHSUBW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPHSUBW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PHSUBW *xmm1*, *xmm2/mem128*	66 0F 38 05 /r	Subtracts adjacent pairs of signed integers in *xmm1* and *xmm2* or *mem128*. Writes packed differences to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPHSUBW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	05 /r
VPHSUBW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	05 /r

Related Instructions

(V)PHSUBD, (V)PHSUBW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PINSRB VPINSRB Packed Insert Byte

Inserts a byte from an 8-bit memory location or the low-order byte of a 32-bit general-purpose register into a destination register. Bits [3:0] of an immediate byte operand select the location where the byte is to be inserted:

Value of imm8 [3:0]	Insertion Location
0000	[7:0]
0001	[15:8]
0010	[23:16]
0011	[31:24]
0100	[39:32]
0101	[47:40]
0110	[55:48]
0111	[63:56]
1000	[71:64]
1001	[79:72]
1010	[87:80]
1011	[95:88]
1100	[103:96]
1101	[111:104]
1110	[119:112]
1111	[127:120]

There are legacy and extended forms of the instruction:

PINSRB

The source operand is either an 8-bit memory location or the low-order byte of a 32-bit general-purpose register and the destination an XMM register. The other bytes of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPINSRB

The extended form of the instruction has a 128-bit encoding only.

There are two source operands. The first source operand is either an 8-bit memory location or the low-order byte of a 32-bit general-purpose register and the second source operand is an XMM register. The destination is a second XMM register. All the bytes of the second source other than the byte that corresponds to the location of the inserted byte are copied to the destination. Bits [255:128] of the YMM register that corresponds to destination are cleared.

Instruction Support

Form	Subset	Feature Flag
PINSRB	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPINSRB	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PINSRB *xmm*, *reg32*/*mem8*, *imm8*	66 0F 3A 20 /r ib	Inserts an 8-bit value selected by *imm8* from the low-order byte of *reg32* or from *mem8* into *xmm*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPINSRB *xmm*, *reg/mem8*, *xmm, imm8*	C4	$\overline{\text{RXB}}$.03	X.1111.0.01	20 /r ib

Related Instructions

(V)PEXTRB, (V)PEXTRD, (V)PEXTRQ, (V)PEXTRW, (V)PINSRD, (V)PINSRQ, (V)PINSRW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PINSRD VPINSRD — Packed Insert Doubleword

Inserts a doubleword from a 32-bit memory location or a 32-bit general-purpose register into a destination register. Bits [1:0] of an immediate byte operand select the location where the doubleword is to be inserted:

Value of imm8 [1:0]	Insertion Location
00	[31:0]
01	[63:32]
10	[95:64]
11	[127:96]

There are legacy and extended forms of the instruction:

PINSRD

The encoding is the same as PINSRQ, with REX.W = 0.

The source operand is either a 32-bit memory location or a 32-bit general-purpose register and the destination an XMM register. The other doublewords of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPINSRD

The extended form of the instruction has a 128-bit encoding only.

The encoding is the same as VPINSRQ, with VEX.W = 0.

There are two source operands. The first source operand is either a 32-bit memory location or a 32-bit general-purpose register and the second source operand is an XMM register. The destination is a second XMM register. All the doublewords of the second source other than the doubleword that corresponds to the location of the inserted doubleword are copied to the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
PINSRD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPINSRD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PINSRD *xmm*, *reg32/mem32*, *imm8*	66 (W0) 0F 3A 22 /r ib	Inserts a 32-bit value selected by *imm8* from *reg32* or *mem32* into *xmm*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPINSRD *xmm*, *reg32/mem32*, *xmm, imm8*	C4	$\overline{\text{RXB}}$.03	0.1111.0.01	22 /r ib

Related Instructions

(V)PEXTRB, (V)PEXTRD, (V)PEXTRQ, (V)PEXTRW, (V)PINSRB, (V)PINSRQ, (V)PINSRW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PINSRQ VPINSRQ — Packed Insert Quadword

Inserts a quadword from a 64-bit memory location or a 64-bit general-purpose register into a destination register. Bit [0] of an immediate byte operand selects the location where the doubleword is to be inserted:

Value of imm8 [0]	Insertion Location
0	[63:0]
1	[127:64]

There are legacy and extended forms of the instruction:

PINSRQ

The encoding is the same as PINSRD, with REX.W = 1.

The source operand is either a 64-bit memory location or a 64-bit general-purpose register and the destination an XMM register. The other quadwords of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPINSRQ

The extended form of the instruction has a 128-bit encoding only.

The encoding is the same as VPINSRD, with VEX.W = 1.

There are two source operands. The first source operand is either a 64-bit memory location or a 64-bit general-purpose register and the second source operand is an XMM register. The destination is a second XMM register. All the quadwords of the second source other than the quadword that corresponds to the location of the inserted quadword are copied to the destination. Bits [255:128] of the YMM register that corresponds to the destination XMM registers are cleared.

Instruction Support

Form	Subset	Feature Flag
PINSRQ	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPINSRQ	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PINSRQ *xmm*, *reg64*/*mem64*, *imm8*	66 (W1) 0F 3A 22 /r ib	Inserts a 64-bit value selected by *imm8* from *reg64* or *mem64* into *xmm*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPINSRQ *xmm*, *reg64/mem64*, *xmm, imm8*	C4	$\overline{\text{RXB}}$.03	1.1111.0.01	22 /r ib

Related Instructions

(V)PEXTRB, (V)PEXTRD, (V)PEXTRQ, (V)PEXTRW, (V)PINSRB, (V)PINSRD, (V)PINSRW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PINSRW VPINSRW — Packed Insert Word

Inserts a word from a 16-bit memory location or the low-order word of a 32-bit general-purpose register into a destination register. Bits [2:0] of an immediate byte operand select the location where the byte is to be inserted:

Value of imm8 [2:0]	Insertion Location
000	[15:0]
001	[31:16]
010	[47:32
011	[63:48]
100	[79:64]
101	[95:80]
110	[111:96]
111	[127:112]

There are legacy and extended forms of the instruction:

PINSRW

The source operand is either a 16-bit memory location or the low-order word of a 32-bit general-purpose register and the destination an XMM register. The other words of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPINSRW

The extended form of the instruction has a 128-bit encoding only.

There are two source operands. The first source operand is either a 16-bit memory location or the low-order word of a 32-bit general-purpose register and the second source operand is an XMM register. The destination is an XMM register. All the words of the second source other than the word that corresponds to the location of the inserted word are copied to the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
PINSRW	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VPINSRW	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PINSRW *xmm*, *reg32*/*mem16*, *imm8*	66 0F C4 /r ib	Inserts a 16-bit value selected by *imm8* from the low-order word of *reg32* or from *mem16* into *xmm*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPINSRW *xmm*, *reg32/mem16*, *xmm, imm8*	C4	$\overline{\text{RXB}}$.01	X.1111.0.01	C4 /r ib

Related Instructions

(V)PEXTRB, (V)PEXTRD, (V)PEXTRQ, (V)PEXTRW, (V)PINSRB, (V)PINSRD, (V)PINSRQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PMADDUBSW VPMADDUBSW — Packed Multiply and Add Unsigned Byte to Signed Word

Multiplies and adds sets of two packed 8-bit unsigned values from the first source operand and two packed 8-bit signed values from the second source operand, with signed saturation; writes eight 16-bit sums to the destination.

For the 128-bit form of the instruction, the following operations are performed:

```
dest is the destination register – either an XMM register or the corresponding YMM register.
src1 is the first source operand. src2 is the second source operand.
Ssum() is a function that returns the saturated 16-bit signed sum of its arguments.

dest[15:0] = Ssum(src1[7:0] * src2[7:0], src1[15:8] * src2[15:8])
dest[31:16] = Ssum(src1[23:16] * src2[23:16], src1[31:24] * src2[31:24])
dest[47:32] = Ssum(src1[39:32] * src2[39:32], src1[47:40] * src2[47:40])
dest[63:48] = Ssum(src1[55:48] * src2[55:48], src1[63:56] * src2[63:56])
dest[79:64] = Ssum(src1[71:64] * src2[71:64], src1[79:72] * src2[79:72])
dest[95:80] = Ssum(src1[87:80] * src2[87:80], src1[95:88] * src2[95:88])
dest[111:96] = Ssum(src1[103:96] * src2[103:96]], src1[111:104] * src2[111:104])
dest[127:112] = Ssum(src1[119:112] * src2[119:112], src1[127:120] * src2[127:120])
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[143:128] = Ssum(src1[135:128] * src2[135:128], src1[143:136] * src2[143:136])
dest[159:144] = Ssum(src1[151:144] * src2[151:144], src1[159:152] * src2[159:152])
dest[175:160] = Ssum(src1[167:160] * src2[167:160], src1[175:168] * src2[175:168])
dest[191:176] = Ssum(src1[183:176] * src2[183:176], src1[191:184] * src2[191:184])
dest[207:192] = Ssum(src1[199:192] * src2[199:192], src1[207:200] * src2[207:200])
dest[223:208] = Ssum(src1[215:208] * src2[215:208], src1[223:216] * src2[223:216])
dest[239:224] = Ssum(src1[231:224] * src2[231:224], src1[239:232] * src2[239:232])
dest[255:240] = Ssum(src1[247:240] * src2[247:240], src1[255:248] * src2[255:248])
```

There are legacy and extended forms of the instruction:

PMADDUBSW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMADDUBSW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMADDUBSW	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPMADDUBSW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMADDUBSW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMADDUBSW *xmm1*, *xmm2/mem128*	66 0F 38 04 /r	Multiplies packed 8-bit unsigned values in *xmm1* and packed 8-bit signed values *xmm2* / *mem128*, adds the products, and writes saturated sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMADDUBSW *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.0.01	04 /r
VPMADDUBSW *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.1.01	04 /r

Related Instructions

(V)PMADDWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMADDWD VPMADDWD Packed Multiply and Add Word to Doubleword

Multiplies and adds sets of four packed 16-bit signed values from two source registers; writes four 32-bit sums to the destination.

For the 128-bit form of the instruction, the following operations are performed:

dest is the destination register – either an XMM register or the corresponding YMM register.
src1 is the first source operand. src2 is the second source operand.

```
dest[31:0] = (src1[15:0] * src2[15:0]) + (src1[31:16] * src2[31:16])
dest[63:32] = (src1[47:32] * src2[47:32]) + (src1[63:48] * src2[63:48])
dest[95:64] = (src1[79:64] * src2[79:64]) + (src1[95:80] * src2[95:80])
dest[127:96] = (src1[111:96] * src2[111:96]) + (src1[127:112] * src2[127:112])
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[159:128] = (src1[143:128] * src2[143:128]) + (src1[159:144] * src2[159:144])
dest[191:160] = (src1[175:160] * src2[175:160]) + (src1[191:176] * src2[191:176])
dest[223:192] = (src1[207:192] * src2[207:192]) + (src1[223:208] * src2[223:208])
dest[255:224] = (src1[239:224] * src2[239:224]) + (src1[255:240] * src2[255:240])
```

When all four of the signed 16-bit source operands in a set have the value 8000h, the 32-bit overflow wraps around to 8000_0000h. There are no other overflow cases.

There are legacy and extended forms of the instruction:

PMADDWD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMADDWD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMADDWD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPMADDWD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMADDWD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMADDWD *xmm1*, *xmm2/mem128*	66 0F F5 /r	Multiplies packed 16-bit signed values in *xmm1* and *xmm2* or *mem128*, adds the products, and writes the sums to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMADDWD *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	F5 /r
VPMADDWD *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	F5 /r

Related Instructions

(V)PMADDUBSW, (V)PMULHUW, (V)PMULHW, (V)PMULLW, (V)PMULUDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMAXSB
VPMAXSB

Packed Maximum Signed Bytes

Compares each packed 8-bit signed integer value of the first source operand to the corresponding value of the second source operand and writes the numerically greater value into the corresponding byte of the destination.

The 128-bit form of the instruction compares 16 pairs of 8-bit signed integer values; the 256-bit form compares 32 pairs.

There are legacy and extended forms of the instruction:

PMAXSB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMAXSB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMAXSB	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMAXSB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMAXSB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMAXSB *xmm1*, *xmm2/mem128*	66 0F 38 3C /r	Compares 16 pairs of packed 8-bit values in *xmm1* and *xmm2* or *mem128* and writes the greater values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMAXSB *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	3C /r
VPMAXSB *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	3C /r

Related Instructions

(V)PMAXSD, (V)PMAXSW, (V)PMAXUB, (V)PMAXUD, (V)PMAXUW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMAXSD
VPMAXSD

Packed Maximum Signed Doublewords

Compares each packed 32-bit signed integer value of the first source operand to the corresponding value of the second source operand and writes the numerically greater value into the corresponding doubleword of the destination.

The 128-bit form of the instruction compares four pairs of 32-bit signed integer values; the 256-bit form compares eight.

There are legacy and extended forms of the instruction:

PMAXSD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMAXSD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMAXSD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMAXSD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMAXSD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMAXSD *xmm1*, *xmm2/mem128*	66 0F 38 3D /r	Compares four pairs of packed 32-bit values in *xmm1* and *xmm2* or *mem128* and writes the greater values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMAXSD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	3D /r
VPMAXSD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	3D /r

Related Instructions

(V)PMAXSB, (V)PMAXSW, (V)PMAXUB, (V)PMAXUD, (V)PMAXUW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMAXSW VPMAXSW Packed Maximum Signed Words

Compares each packed 16-bit signed integer value of the first source operand to the corresponding value of the second source operand and writes the numerically greater value into the corresponding word of the destination.

The 128-bit form of the instruction compares eight pairs of 16-bit signed integer values; the 256-bit form compares 16 pairs.

There are legacy and extended forms of the instruction:

PMAXSW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMAXSW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMAXSW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPMAXSW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMAXSW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMAXSW *xmm1*, *xmm2/mem128*	66 0F EE /r	Compares eight pairs of packed 16-bit values in *xmm1* and *xmm2* or *mem128* and writes the greater values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMAXSW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	EE /r
VPMAXSW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	EE /r

Related Instructions

(V)PMAXSB, (V)PMAXSD, (V)PMAXUB, (V)PMAXUD, (V)PMAXUW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMAXUB VPMAXUB — Packed Maximum Unsigned Bytes

Compares each packed 8-bit unsigned integer value of the first source operand to the corresponding value of the second source operand and writes the numerically greater value into the corresponding byte of the destination.

The 128-bit form of the instruction compares 16 pairs of 8-bit unsigned integer values; the 256-bit form compares 32 pairs.

There are legacy and extended forms of the instruction:

PMAXUB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMAXUB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMAXUB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPMAXUB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMAXUB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMAXUB *xmm1*, *xmm2/mem128*	66 0F DE /r	Compares 16 pairs of packed unsigned 8-bit values in *xmm1* and *xmm2* or *mem128* and writes the greater values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMAXUB *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	DE /r
VPMAXUB *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	DE /r

Related Instructions

(V)PMAXSB, (V)PMAXSD, (V)PMAXSW, (V)PMAXUD, (V)PMAXUW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

None

PMAXUD VPMAXUD — Packed Maximum Unsigned Doublewords

Compares each packed 32-bit unsigned integer value of the first source operand to the corresponding value of the second source operand and writes the numerically greater value into the corresponding doubleword of the destination.

The 128-bit form of the instruction compares four pairs of 32-bit unsigned integer values; the 256-bit form compares eight.

There are legacy and extended forms of the instruction:

PMAXUD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMAXUD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMAXUD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMAXUD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMAXUD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMAXUD *xmm1*, *xmm2/mem128*	66 0F 38 3F /r	Compares four pairs of packed unsigned 32-bit values in *xmm1* and *xmm2* or *mem128* and writes the greater values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMAXUD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	3F /r
VPMAXUD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	3F /r

Related Instructions

(V)PMAXSB, (V)PMAXSD, (V)PMAXSW, (V)PMAXUB, (V)PMAXUW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMAXUW VPMAXUW — Packed Maximum Unsigned Words

Compares each packed 16-bit unsigned integer value of the first source operand to the corresponding value of the second source operand and writes the numerically greater value into the corresponding word of the destination.

The 128-bit form of the instruction compares eight pairs of 16-bit unsigned integer values; the 256-bit form compares 16 pairs.

There are legacy and extended forms of the instruction:

PMAXUW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMAXUW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMAXUW	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMAXUW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMAXUW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMAXUW *xmm1*, *xmm2/mem128*	66 0F 38 3E /r	Compares eight pairs of packed unsigned 16-bit values in *xmm1* and *xmm2* or *mem128* and writes the greater values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMAXUW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	3E /r
VPMAXUW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	3E /r

Related Instructions

(V)PMAXSB, (V)PMAXSD, (V)PMAXSW, (V)PMAXUB, (V)PMAXUD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMINSB VPMINSB — Packed Minimum Signed Bytes

Compares each packed 8-bit signed integer value of the first source operand to the corresponding value of the second source operand and writes the numerically lesser value into the corresponding byte of the destination.

The 128-bit form of the instruction compares 16 pairs of 8-bit signed integer values; the 256-bit form compares 32 pairs.

There are legacy and extended forms of the instruction:

PMINSB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMINSB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMINSB	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMINSB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMINSB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMINSB *xmm1*, *xmm2/mem128*	66 0F 38 38 /r	Compares 16 pairs of packed 8-bit values in *xmm1* and *xmm2* or *mem128* and writes the lesser values to the corresponding positions in *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMINSB *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	38 /r
VPMINSB *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	38 /r

Related Instructions

(V)PMINSD, (V)PMINSW, (V)PMINUB, (V)PMINUD, (V)PMINUW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMINSD VPMINSD — Packed Minimum Signed Doublewords

Compares each packed 32-bit signed integer value of the first source operand to the corresponding value of the second source operand and writes the numerically lesser value into the corresponding doubleword of the destination.

The 128-bit form of the instruction compares four pairs of 32-bit signed integer values; the 256-bit form compares eight.

There are legacy and extended forms of the instruction:

PMINSD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMINSD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMINSD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMINSD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMINSD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMINSD *xmm1*, *xmm2/mem128*	66 0F 38 39 /r	Compares four pairs of packed 32-bit values in *xmm1* and *xmm2* or *mem128* and writes the lesser values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMINSD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	39 /r
VPMINSD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	39 /r

Related Instructions

(V)PMINSB, (V)PMINSW, (V)PMINUB, (V)PMINUD, (V)PMINUW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMINSW VPMINSW Packed Minimum Signed Words

Compares each packed 16-bit signed integer value of the first source operand to the corresponding value of the second source operand and writes the numerically lesser value into the corresponding word of the destination.

The 128-bit form of the instruction compares eight pairs of 16-bit signed integer values; the 256-bit form compares 16 pairs.

There are legacy and extended forms of the instruction:

PMINSW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMINSW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMINSW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPMINSW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMINSW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMINSW *xmm1*, *xmm2/mem128*	66 0F EA /r	Compares eight pairs of packed 16-bit values in *xmm1* and *xmm2* or *mem128* and writes the lesser values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMINSW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	EA /r
VPMINSW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	EA /r

Related Instructions

(V)PMINSB, (V)PMINSD, (V)PMINUB, (V)PMINUD, (V)PMINUW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMINUB VPMINUB — Packed Minimum Unsigned Bytes

Compares each packed 8-bit unsigned integer value of the first source operand to the corresponding value of the second source operand and writes the numerically lesser value into the corresponding byte of the destination.

The 128-bit form of the instruction compares 16 pairs of 8-bit unsigned integer values; the 256-bit form compares 32 pairs.

There are legacy and extended forms of the instruction:

PMINUB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMINUB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMINUB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPMINUB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMINUB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMINUB *xmm1*, *xmm2/mem128*	66 0F DA /r	Compares 16 pairs of packed unsigned 8-bit values in *xmm1* and *xmm2* or *mem128* and writes the lesser values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMINUB *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	DA /r
VPMINUB *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	DA /r

Related Instructions

(V)PMINSB, (V)PMINSD, (V)PMINSW, (V)PMINUD, (V)PMINUW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMINUD Packed Minimum
VPMINUD Unsigned Doublewords

Compares each packed 32-bit unsigned integer value of the first source operand to the corresponding value of the second source operand and writes the numerically lesser value into the corresponding doubleword of the destination.

The 128-bit form of the instruction compares four pairs of 32-bit unsigned integer values; the 256-bit form compares eight.

There are legacy and extended forms of the instruction:

PMINUD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMINUD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMINUD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMINUD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMINUD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMINUD *xmm1*, *xmm2/mem128*	66 0F 38 3B /r	Compares four pairs of packed unsigned 32-bit values in *xmm1* and *xmm2* or *mem128* and writes the lesser values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMINUD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	3B /r
VPMINUD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	3B /r

Related Instructions

(V)PMINSB, (V)PMINSD, (V)PMINSW, (V)PMINUB, (V)PMINUW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMINUW VPMINUW — Packed Minimum Unsigned Words

Compares each packed 16-bit unsigned integer value of the first source operand to the corresponding value of the second source operand and writes the numerically lesser value into the corresponding word of the destination.

The 128-bit form of the instruction compares eight pairs of 16-bit unsigned integer values; the 256-bit form compares 16 pairs.

There are legacy and extended forms of the instruction:

PMINUW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMINUW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMINUW	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMINUW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMINUW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMINUW *xmm1*, *xmm2/mem128*	66 0F 38 3A /r	Compares eight pairs of packed unsigned 16-bit values in *xmm1* and *xmm2* or *mem128* and writes the lesser values to the corresponding positions in *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMINUW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	3A /r
VPMINUW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	3A /r

Related Instructions

(V)PMINSB, (V)PMINSD, (V)PMINSW, (V)PMINUB, (V)PMINUD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMOVMSKB VPMOVMSKB — Packed Move Mask Byte

Copies the value of the most-significant bit of each byte element of the source operand to create a 16 or 32 bit mask value, zero-extends the value, and writes it to the destination.

There are legacy and extended forms of the instruction:

PMOVMSKB

The source operand is an XMM register. The destination is a 32-bit general purpose register. The mask is zero-extended to fill the destination register, the mask occupies bits [15:0].

VPMOVMSKB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is an XMM register. The destination is a 64-bit general purpose register. The mask is zero-extended to fill the destination register, the mask occupies bits [15:0].

YMM Encoding

The source operand is a YMM register. The destination is a 64-bit general purpose register. The mask is zero-extended to fill the destination register, the mask occupies bits [31:0].

Instruction Support

Form	Subset	Feature Flag
PMOVMSKB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPMOVMSKB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVMSKB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVMSKB *reg32*, *xmm1*	66 0F D7 /r	Moves a zero-extended mask consisting of the most-significant bit of each byte in *xmm1* to a 32-bit general-purpose register.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VMOVMSKB *reg64*, *xmm1*	C4	$\overline{\text{RXB}}$.01	X.1111.0.01	D7 /r
VMOVMSKB *reg64*, *ymm1*	C4	$\overline{\text{RXB}}$.01	X.1111.1.01	D7 /r

Related Instructions

(V)MOVMSKPD, (V)MOVMSKPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv field ! = 1111b.
			A	VEX.L field = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — SSE, AVX and AVX2 exception
A — AVX, AVX2exception
S — SSE exception

PMOVSXBD VPMOVSXBD — Packed Move with Sign-Extension Byte to Doubleword

Sign-extends four or eight packed 8-bit signed integers in the source operand to 32 bits and writes the packed doubleword signed integers to the destination.

If the source operand is a register, the 8-bit signed integers are taken from the least-significant bytes of the register.

There are legacy and extended forms of the instruction:

PMOVSXBD

The source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMOVSXBD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either an XMM register or a 64-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PMOVSXBD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMOVSXBD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVSXBD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVSXBD *xmm1*, *xmm2/mem32*	66 0F 38 21 /r	Sign-extends four packed signed 8-bit integers in the four low bytes of *xmm2* or *mem32* and writes four packed signed 32-bit integers to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMOVSXBD *xmm1*, *xmm2/mem32*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	21 /r
VPMOVSXBD *ymm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	21 /r

Related Instructions

(V)PMOVSXBQ, (V)PMOVSXBW, (V)PMOVSXDQ, (V)PMOVSXWD, (V)PMOVSXW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMOVSXBQ VPMOVSXBQ — Packed Move with Sign Extension Byte to Quadword

Sign-extends two or four packed 8-bit signed integers in the source operand to 64 bits and writes the packed quadword signed integers to the destination.

If the source operand is a register, the 8-bit signed integers are taken from the least-significant bytes of the register.

There are legacy and extended forms of the instruction:

PMOVSXBQ

The source operand is either an XMM register or a 16-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMOVSXBQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 16-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either an XMM register or a 32-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PMOVSXBQ	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMOVSXBQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVSXBQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVSXBQ *xmm1, xmm2/mem16*	66 0F 38 22 /r	Sign-extends two packed signed 8-bit integers in the two low bytes of *xmm2* or *mem16* and writes two packed signed 64-bit integers to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMOVSXBQ *xmm1*, *xmm2/mem16*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	22 /r
VPMOVSXBQ *ymm1*, *xmm2/mem32*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	22 /r

Related Instructions

(V)PMOVSXBD, (V)PMOVSXBW, (V)PMOVSXDQ, (V)PMOVSXWD, (V)PMOVSXW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMOVSXBW VPMOVSXBW — Packed Move with Sign Extension Byte to Word

Sign-extends eight or sixteen packed 8-bit signed integers in the source operand to 16 bits and writes the packed word signed integers to the destination.

If the source operand is a register, the eight 8-bit signed integers are taken from the lower half of the register.

There are legacy and extended forms of the instruction:

PMOVSXBW

The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMOVSXBW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either an XMM register or a 128-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PMOVSXBW	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMOVSXBW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVSXBW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVSXBW *xmm1*, *xmm2/mem64*	66 0F 38 20 /r	Sign-extends eight packed signed 8-bit integers in the eight low bytes of *xmm2* or *mem64* and writes eight packed signed 16-bit integers to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMOVSXBW *xmm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	20 /r
VPMOVSXBW *ymm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	20 /r

Related Instructions

(V)PMOVSXBD, (V)PMOVSXBQ, (V)PMOVSXDQ, (V)PMOVSXWD, (V)PMOVSXW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMOVSXDQ VPMOVSXDQ — Packed Move with Sign-Extension Doubleword to Quadword

Sign-extends two or four packed 32-bit signed integers in the source operand to 64 bits and writes the packed quadword signed integers to the destination.

If the source operand is a register, the two 32-bit signed integers are taken from the lower half of the register.

There are legacy and extended forms of the instruction:

PMOVSXDQ

The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMOVSXDQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either an XMM register or a 128-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PMOVSXDQ	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMOVSXDQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVSXDQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVSXDQ *xmm1*, *xmm2/mem64*	66 0F 38 25 /r	Sign-extends two packed signed 32-bit integers in the two low doublewords of *xmm2* or *mem64* and writes two packed signed 64-bit integers to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMOVSXDQ *xmm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	25 /r
VPMOVSXDQ *ymm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	25 /r

Related Instructions

(V)PMOVSXBD, (V)PMOVSXBQ, (V)PMOVSXBW, (V)PMOVSXWD, (V)PMOVSXWQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMOVSXWD VPMOVSXWD — Packed Move with Sign-Extension Word to Doubleword

Sign-extends four or eight packed 16-bit signed integers in the source operand to 32 bits and writes the packed doubleword signed integers to the destination.

If the source operand is a register, the four 16-bit signed integers are taken from the lower half of the register.

There are legacy and extended forms of the instruction:

PMOVSXWD

The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMOVSXWD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either an XMM register or a 128-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PMOVSXWD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMOVSXWD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVSXWD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVSXWD *xmm1*, *xmm2/mem64*	66 0F 38 23 /r	Sign-extends four packed signed 16-bit integers in the four low words of *xmm2* or *mem64* and writes four packed signed 32-bit integers to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMOVSXWD *xmm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	23 /r
VPMOVSXWD *ymm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	23 /r

Related Instructions

(V)PMOVSXBD, (V)PMOVSXBQ, (V)PMOVSXBW, (V)PMOVSXDQ, (V)PMOVSXWQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMOVSXWQ VPMOVSXWQ Packed Move with Sign-Extension Word to Quadword

Sign-extends two or four packed 16-bit signed integers to 64 bits and writes the packed quadword signed integers to the destination.

If the source operand is a register, the 16-bit signed integers are taken from least-significant words of the register.

There are legacy and extended forms of the instruction:

PMOVSXWQ

The source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMOVSXWQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either an XMM register or a 64-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PMOVSXWQ	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMOVSXWQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVSXWQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVSXWQ *xmm1*, *xmm2/mem32*	66 0F 38 24 /r	Sign-extends two packed signed 16-bit integers in the two low words of *xmm2* or *mem32* and writes two packed signed 64-bit integers to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMOVSXWQ *xmm1*, *xmm2/mem32*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	24 /r
VPMOVSXWQ *ymm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	24 /r

Related Instructions

(V)PMOVSXBD, (V)PMOVSXBQ, (V)PMOVSXBW, (V)PMOVSXDQ, (V)PMOVSXWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMOVZXBD VPMOVZXBD — Packed Move with Zero-Extension Byte to Doubleword

Zero-extends four or eight packed 8-bit unsigned integers in the source operand to 32 bits and writes the packed doubleword positive-signed integers to the destination.

If the source operand is a register, the 8-bit signed integers are taken from the least-significant bytes of the register.

There are legacy and extended forms of the instruction:

PMOVZXBD

The source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMOVZXBD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either an XMM register or a 64-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PMOVZXBD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMOVZXBD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVZXBD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVZXBD *xmm1*, *xmm2/mem32*	66 0F 38 31 /r	Zero-extends four packed unsigned 8-bit integers in the four low bytes of *xmm2* or *mem32* and writes four packed positive-signed 32-bit integers to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMOVZXBD *xmm1*, *xmm2/mem32*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	31 /r
VPMOVZXBD *ymm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	31 /r

Related Instructions

(V)PMOVZXBQ, (V)PMOVZXBW, (V)PMOVZXDQ, (V)PMOVZXWD, (V)PMOVZXW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMOVZXBQ VPMOVZXBQ — Packed Move Byte to Quadword with Zero-Extension

Zero-extends two or four packed 8-bit unsigned integers in the source operand to 64 bits and writes the packed quadword positive-signed integers to the destination.

If the source operand is a register, the 8-bit signed integers are taken from the least-significant bytes of the register.

There are legacy and extended forms of the instruction:

PMOVZXBQ

The source operand is either an XMM register or a 16-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMOVZXBQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 16-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either an XMM register or a 32-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PMOVZXBQ	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMOVZXBQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVZXBQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVZXBQ *xmm1, xmm2/mem16*	66 0F 38 32 /r	Zero-extends two packed unsigned 8-bit integers in the two low bytes of *xmm2* or *mem16* and writes two packed positive-signed 64-bit integers to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMOVZXBQ *xmm1, xmm2/mem16*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	32 /r
VPMOVZXBQ *ymm1, xmm2/mem32*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	32 /r

Related Instructions

(V)PMOVZXBD, (V)PMOVZXBW, (V)PMOVZXDQ, (V)PMOVZXWD, (V)PMOVZXW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMOVZXBW Packed Move Byte to Word with Zero-Extension VPMOVZXBW

Zero-extends eight or sixteen packed 8-bit unsigned integers in the source operand to 16 bits and writes the packed word positive-signed integers to the destination.

If the source operand is a register, the eight 8-bit signed integers are taken from the lower half of the register.

There are legacy and extended forms of the instruction:

PMOVZXBW

The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMOVZXBW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either an XMM register or a 128-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PMOVZXBW	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMOVZXBW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVZXBW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVZXBW *xmm1*, *xmm2/mem64*	66 0F 38 30 /r	Zero-extends eight packed unsigned 8-bit integers in the eight low bytes of *xmm2* or *mem64* and writes eight packed positive-signed 16-bit integers to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMOVZXBW *xmm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	30 /r
VPMOVZXBW *ymm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	30 /r

Related Instructions

(V)PMOVZXBD, (V)PMOVZXBQ, (V)PMOVZXDQ, (V)PMOVZXWD, (V)PMOVZXW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMOVZXDQ VPMOVZXDQ Packed Move with Zero-Extension Doubleword to Quadword

Zero-extends two or four packed 32-bit unsigned integers in the source operand to 64 bits and writes the packed quadword positive-signed integers to the destination.

If the source operand is a register, the two 32-bit signed integers are taken from the lower half of the register.

There are legacy and extended forms of the instruction:

PMOVZXDQ

The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMOVZXDQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either an XMM register or a 128-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PMOVZXDQ	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMOVZXDQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVZXDQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVZXDQ *xmm1*, *xmm2/mem64*	66 0F 38 35 /r	Zero-extends two packed unsigned 32-bit integers in the two low doublewords of *xmm2* or *mem64* and writes two packed positive-signed 64-bit integers to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMOVZXDQ *xmm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	35 /r
VPMOVZXDQ *ymm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	35 /r

Related Instructions

(V)PMOVZXBD, (V)PMOVZXBQ, (V)PMOVZXBW, (V)PMOVZXWD, (V)PMOVZXWQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMOVZXWD VPMOVZXWD — Packed Move Word to Doubleword with Zero-Extension

Zero-extends four or eight packed 16-bit unsigned integers in the source operand to 32 bits and writes the packed doubleword positive-signed integers to the destination.

If the source operand is a register, the four 16-bit signed integers are taken from the lower half of the register.

There are legacy and extended forms of the instruction:

PMOVZXWD

The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMOVZXWD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either an XMM register or a 128-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PMOVZXWD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMOVZXWD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVZXWD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVZXWD *xmm1*, *xmm2/mem64*	66 0F 38 33 /r	Zero-extends four packed unsigned 16-bit integers in the four low words of *xmm2* or *mem64* and writes four packed positive-signed 32-bit integers to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMOVZXWD *xmm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	33 /r
VPMOVZXWD *ymm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	33 /r

Related Instructions

(V)PMOVZXBD, (V)PMOVZXBQ, (V)PMOVZXBW, (V)PMOVZXDQ, (V)PMOVZXWQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMOVZXWQ VPMOVZXWQ — Packed Move with Zero-Extension Word to Quadword

Zero-extends two or four packed 16-bit unsigned integers to 64 bits and writes the packed quadword positive signed integers to the destination.

If the source operand is a register, the 16-bit signed integers are taken from least-significant words of the register.

There are legacy and extended forms of the instruction:

PMOVZXWQ

The source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMOVZXWQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either an XMM register or a 64-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PMOVZXWQ	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMOVZXWQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMOVZXWQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMOVZXWQ *xmm1*, *xmm2/mem32*	66 0F 38 34 /r	Zero-extends two packed unsigned 16-bit integers in the two low words of *xmm2* or *mem32* and writes two packed positive-signed 64-bit integers to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMOVZXWQ *xmm1*, *xmm2/mem32*	C4	$\overline{\text{RXB}}$.02	X.1111.0.01	34 /r
VPMOVZXWQ *ymm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	X.1111.1.01	34 /r

Related Instructions

(V)PMOVZXBD, (V)PMOVZXBQ, (V)PMOVZXBW, (V)PMOVZXDQ, (V)PMOVZXWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMULDQ Packed Multiply
VPMULDQ Signed Doubleword to Quadword

Multiplies two or four pairs of 32-bit signed integers in the first and second source operands and writes two or four packed quadword signed integer products to the destination.

For the 128-bit form of the instruction, the following operations are performed:

dest is the destination register – either an XMM register or the corresponding YMM register.
src1 is the first source operand. src2 is the second source operand.

```
dest[63:0] = (src1[31:0] * src2[31:0])
dest[127:64] = (src1[95:64] * src2[95:64])
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[191:128] = (src1[159:128] * src2[159:128])
dest[255:192] = (src1[223:192] * src2[223:192])
```

There are legacy and extended forms of the instruction:

PMULDQ

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMULDQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMULDQ	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMULDQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMULDQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMULDQ *xmm1*, *xmm2/mem128*	66 0F 38 28 /r	Multiplies two packed 32-bit signed integers in *xmm1[31:0]* and *xmm1[95:64]* by the corresponding values in *xmm2* or *mem128*. Writes packed 64-bit signed integer products to *xmm1[63:0]* and *xmm1[127:64]*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMULDQ *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	28 /r
VPMULDQ *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	28 /r

Related Instructions

(V)PMULLD, (V)PMULHW, (V)PMULHUW,(V)PMULUDQ, (V)PMULLW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMULHRSW VPMULHRSW Packed Multiply High with Round and Scale Words

Multiplies each packed 16-bit signed value in the first source operand by the corresponding value in the second source operand, truncates the 32-bit product to the 18 most significant bits by right-shifting, then rounds the truncated value by adding 1 to its least-significant bit. Writes bits [16:1] of the sum to the corresponding word of the destination.

There are legacy and extended forms of the instruction:

PMULHRSW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMULHRSW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMULHRSW	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPMULHRSW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMULHRSW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMULHRSW *xmm1*, *xmm2*/*mem128*	66 0F 38 0B /r	Multiplies each packed 16-bit signed value in *xmm1* by the corresponding value in *xmm2* or *mem128*, truncates product to 18 bits, rounds by adding 1. Writes bits [16:1] of the sum to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMULHRSW *xmm1*, *xmm2, xmm3*/*mem128*	C4	$\overline{\text{RXB}}$.2	X.$\overline{src1}$.0.01	0B /r
VPMULHRSW *ymm1*, *ymm2, ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.2	X.$\overline{src1}$.1.01	0B /r

Related Instructions

None

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMULHUW VPMULHUW — Packed Multiply High Unsigned Word

Multiplies each packed 16-bit unsigned value in the first source operand by the corresponding value in the second source operand; writes the high-order 16 bits of each 32-bit product to the corresponding word of the destination.

There are legacy and extended forms of the instruction:

PMULHUW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMULHUW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMULHUW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPMULHUW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMULHUW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMULHUW *xmm1*, *xmm2/mem128*	66 0F E4 /r	Multiplies packed 16-bit unsigned values in *xmm1* by the corresponding values in *xmm2* or *mem128.* Writes bits [31:16] of each product to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMULHUW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	E4 /r
VPMULHUW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	E4 /r

Related Instructions

(V)PMULDQ, (V)PMULHW, (V)PMULLD, (V)PMULLW, (V)PMULUDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMULHW
VPMULHW
Packed Multiply High Signed Word

Multiplies each packed 16-bit signed value in the first source operand by the corresponding value in the second source operand; writes the high-order 16 bits of each 32-bit product to the corresponding word of the destination.

There are legacy and extended forms of the instruction:

PMULHW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMULHW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMULHW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPMULHW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMULHW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMULHW *xmm1*, *xmm2/mem128*	66 0F E5 /r	Multiplies packed 16-bit signed values in *xmm1* by the corresponding values in *xmm2* or *mem128.* Writes bits [31:16] of each product to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMULHW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	E5 /r
VPMULHW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	E5 /r

Related Instructions

(V)PMULDQ, (V)PMULHUW, (V)PMULLD, (V)PMULLW, (V)PMULUDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMULLD VPMULLD — Packed Multiply and Store Low Signed Doubleword

Multiplies four packed 32-bit signed integers in the first source operand by the corresponding values in the second source operand and writes bits [31:0] of each 64-bit product to the corresponding 32-bit element of the destination.

There are legacy and extended forms of the instruction:

PMULLD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMULLD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMULLD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPMULLD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMULLD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMULLD *xmm1*, *xmm2/mem128*	66 0F 38 40 /r	Multiplies four packed 32-bit signed integers in *xmm1* by corresponding values in *xmm2* or *m128.* Writes bits [31:0] of each 64-bit product to the corresponding 32-bit element of *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMULLD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	40 /r
VPMULLD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	40 /r

Related Instructions

(V)PMULDQ, (V)PMULHUW, (V)PMULHW, (V)PMULLW, (V)PMULUDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMULLW
VPMULLW

Packed Multiply Low Signed Word

Multiplies eight packed 16-bit signed integers in the first source operand by the corresponding values in the second source operand and writes bits [15:0] of each 32-bit product to the corresponding 16-bit element of the destination.

There are legacy and extended forms of the instruction:

PMULLW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMULLW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMULLW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPMULLW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMULLW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMULLW *xmm1*, *xmm2/mem128*	66 0F D5 /r	Multiplies eight packed 16-bit signed integers in *xmm1* by corresponding values in *xmm2* or *m128.* Writes bits [15:0] of each 32-bit product to the corresponding 16-bit element of *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMULLW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	D5 /r
VPMULLW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	D5 /r

Related Instructions

(V)PMULDQ, (V)PMULHUW, (V)PMULHW, (V)PMULLD, (V)PMULUDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PMULUDQ Packed Multiply
VPMULUDQ Unsigned Doubleword to Quadword

Multiplies two or four pairs of 32-bit unsigned integers in the first and second source operands and writes two or four packed quadword unsigned integer products to the destination.

For the 128-bit form of the instruction, the following operations are performed:

dest is the destination register – either an XMM register or the corresponding YMM register.
src1 is the first source operand. src2 is the second source operand.

```
dest[63:0] = (src1[31:0] * src2[31:0])
dest[127:64] = (src1[95:64] * src2[95:64])
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[191:128] = (src1[159:128] * src2[159:128])
dest[255:192] = (src1[223:192] * src2[223:192])
```

There are legacy and extended forms of the instruction:

PMULUDQ

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPMULUDQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PMULUDQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPMULUDQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPMULUDQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PMULUDQ *xmm1*, *xmm2/mem128*	66 0F F4 /r	Multiplies two packed 32-bit unsigned integers in *xmm1[31:0]* and *xmm1[95:64]* by the corresponding values in *xmm2* or *mem128*. Writes packed 64-bit unsigned integer products to *xmm1[63:0]* and *xmm1[127:64]*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPMULUDQ *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	F4 /r
VPMULUDQ *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	F4 /r

Related Instructions

(V)PMULDQ, (V)PMULHUW, (V)PMULHW, (V)PMULLD, (V)PMULLW, (V)PMULUDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

POR Packed OR
VPOR

Performs a bitwise OR of the first and second source operands and writes the result to the destination. When one or both of a pair of corresponding bits in the first and second operands are set, the corresponding bit of the destination is set; when neither source bit is set, the destination bit is cleared.

There are legacy and extended forms of the instruction:

POR

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPOR

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
POR	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPOR 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPOR 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
POR *xmm1*, *xmm2/mem128*	66 0F EB /r	Performs bitwise OR of values in *xmm1* and *xmm2* or *mem128*. Writes results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPOR *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	EB /r
VPOR *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	EB /r

Related Instructions

(V)PAND, (V)PANDN, (V)PXOR

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSADBW VPSADBW — Packed Sum of Absolute Differences Bytes to Words

Subtracts the 16 or 32 packed 8-bit unsigned integers in the second source operand from the corresponding values in the first source operand and computes the absolute value of the differences. Computes two or four unsigned 16-bit integer sums of groups of eight absolute differences and writes the sums to specific words of the destination.

For the 128-bit form of the instruction:

- The unsigned 16-bit integer sum of absolute differences of the eight bytes [7:0] of the source operands is written to bits [15:0] of the destination; bits [63:16] are cleared.
- The unsigned 16-bit integer sum of absolute differences of the eight bytes [15:8] of the source operands is written to bits [79:64] of the destination; bits [127:80] are cleared.

Additionally, for the 256-bit form of the instruction:

- The unsigned 16-bit integer sum of absolute differences of the eight bytes [23:16] of the source operands is written to bits [143:128] of the destination; bits [191:144] are cleared.
- The unsigned 16-bit integer sum of absolute differences of the eight bytes [24:31] of the source operands is written to bits [207:192] of the destination; bits [255:208] are cleared.

There are legacy and extended forms of the instruction:

PSADBW

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSADBW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSADBW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSADBW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSADBW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSADBW *xmm1*, *xmm2*/*mem128*	66 0F F6 /r	Compute the sum of the absolute differences of two sets of packed 8-bit unsigned integer values in *xmm1* and *xmm2* or *mem128*. Writes 16-bit unsigned integer sums to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSADBW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	F6 /r
VPSADBW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	F6 /r

Related Instructions

(V)MPSADBW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

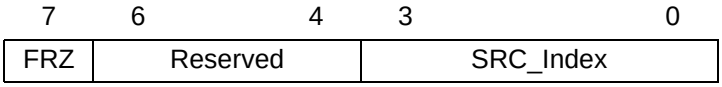
Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSHUFB VPSHUFB — Packed Shuffle Byte

Copies bytes from the first source operand to the destination or clears bytes in the destination, as specified by control bytes in the second source operand.

The control bytes occupy positions in the source operand that correspond to positions in the destination. Each control byte has the following fields.



Bits	Description
[7]	Set the bit to clear the corresponding byte of the destination. Clear the bit to copy the selected source byte to the corresponding byte of the destination.
[6:4]	Reserved
[3:0]	Binary value selects the source byte.

For the 256-bit form of the instruction, the SRC_Index fields in the upper 16 bytes of the second source operand select bytes in the upper 16 bytes of the first source operand to be copied.

There are legacy and extended forms of the instruction:

PSHUFB

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSHUFB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSHUFB	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPSHUFB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSHUFB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSHUFB *xmm1*, *xmm2/mem128*	66 0F 38 00 /r	Moves bytes in *xmm1* as specified by control bytes in *xmm2* or *mem128*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSHUFB *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.0.01	00 /r
VPSHUFB *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.1.01	00 /r

Related Instructions

(V)PSHUFD, (V)PSHUFW, (V)PSHUHW, (V)PSHUFLW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSHUFD
VPSHUFD
Packed Shuffle Doublewords

Copies packed doubleword values from a source to a doubleword in the destination, as specified by bit fields of an immediate byte operand. A source doubleword can be copied more than once.

Source doublewords are selected by two-bit fields in the immediate-byte operand. Each field corresponds to a destination doubleword, as shown:

Destination Doubleword	Immediate-Byte Bit Field	Value of Bit Field	Source Doubleword
[31:0]	[1:0]	00	[31:0]
		01	[63:32]
		10	[95:64]
		11	[127:96]
[63:32]	[3:2]	00	[31:0]
		01	[63:32]
		10	[95:64]
		11	[127:96]
[95:64]	[5:4]	00	[31:0]
		01	[63:32]
		10	[95:64]
		11	[127:96]
[127:96]	[7:6]	00	[31:0]
		01	[63:32]
		10	[95:64]
		11	[127:96]

For the 256-bit form of the instruction, the same immediate byte selects doublewords in the upper 128-bits of the source operand to be copied to the destination.

Destination Doubleword	Immediate-Byte Bit Field	Value of Bit Field	Source Doubleword
[159:128]	[1:0]	00	[159:128]
		01	[191:160]
		10	[223:192]
		11	[225:224]
[191:160]	[3:2]	00	[159:128]
		01	[191:160]
		10	[223:192]
		11	[225:224]

Destination Doubleword	Immediate-Byte Bit Field	Value of Bit Field	Source Doubleword
[223:192]	[5:4]	00	[159:128]
		01	[191:160]
		10	[223:192]
		11	[225:224]
[255:224]	[7:6]	00	[159:128]
		01	[191:160]
		10	[223:192]
		11	[225:224]

There are legacy and extended forms of the instruction:

PSHUFD

The source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSHUFD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PSHUFD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSHUFD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSHUFD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSHUFD *xmm1*, *xmm2/mem128*, *imm8*	66 0F 70 /r ib	Copies packed 32-bit values from *xmm2* or *mem128* to *xmm1*, as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSHUFD *xmm1*, *xmm2/mem128*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.1111.0.01	70 /r ib
VPSHUFD *ymm1*, *ymm2/mem256*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.1111.1.01	70 /r ib

Related Instructions

(V)PSHUFHW, (V)PSHUFLW, (V)PSHUFW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSHUFHW VPSHUFHW — Packed Shuffle High Words

Copies packed word values from the high quadword of the source operand or the upper quadwords of two halves of the source operand to a word in the high quadword of the destination or the upper quadwords of two halves of the destination, as specified by bit fields of an immediate byte operand. A source word can be copied more than once.

Source words are selected by two-bit fields in the immediate-byte operand. Each field corresponds to a destination word, as shown:

Destination Word	Immediate-Byte Bit Field	Value of Bit Field	Source Word
[79:64]	[1:0]	00	[79:64]
		01	[95:80]
		10	[111:96]
		11	[127:112]
[95:80]	[3:2]	00	[79:64]
		01	[95:80]
		10	[111:96]
		11	[127:112]
[111:96]	[5:4]	00	[79:64]
		01	[95:80]
		10	[111:96]
		11	[127:112]
[127:112]	[7:6]	00	[79:64]
		01	[95:80]
		10	[111:96]
		11	[127:112]

The least-significant quadword of the source is copied to the corresponding quadword of the destination.

For the 256-bit form of the instruction, the same immediate byte selects words in the most-significant quadword of the source operand to be copied to the destination:

Destination Word	Immediate-Byte Bit Field	Value of Bit Field	Source Word
[207:192]	[1:0]	00	[207:192]
		01	[223:208]
		10	[239:224]
		11	[255:240]

Destination Word	Immediate-Byte Bit Field	Value of Bit Field	Source Word
[223:208]	[3:2]	00	[207:192]
		01	[223:208]
		10	[239:224]
		11	[255:240]
[239:224]	[5:4]	00	[207:192]
		01	[223:208]
		10	[239:224]
		11	[255:240]
[255:240]	[7:6]	00	[207:192]
		01	[223:208]
		10	[239:224]
		11	[255:240]

The least-significant quadword of the upper 128 bits of the source is copied to the corresponding quadword of the destination.

There are legacy and extended forms of the instruction:

PSHUFHW

The source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSHUFHW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PSHUFHW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSHUFHW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSHUFHW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSHUFHW *xmm1*, *xmm2*/*mem128*, *imm8*	F3 0F 70 /r ib	Copies packed 16-bit values from the high-order quadword of *xmm2* or *mem128* to the high-order quadword of *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSHUFHW *xmm1*, *xmm2/mem128*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.1111.0.10	70 /r ib
VPSHUFHW *ymm1*, *ymm2/mem256*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.1111.1.10	70 /r ib

Related Instructions

(V)PSHUFD, (V)PSHUFLW, (V)PSHUFW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSHUFLW VPSHUFLW — Packed Shuffle Low Words

Copies packed word values from the low quadword of the source operand or the lower quadwords of two halves of the source operand to a word in the low quadword of the destination or the lower quadwords of two halves of the destination, as specified by bit fields of an immediate byte operand. A source word can be copied more than once.

Source words are selected by two-bit fields in the immediate-byte operand. Each bit field corresponds to a destination word, as shown:

Destination Word	Immediate-Byte Bit Field	Value of Bit Field	Source Word
[15:0]	[1:0]	00	[15:0]
		01	[31:16]
		10	[47:32]
		11	[63:48]
[31:16]	[3:2]	00	[15:0]
		01	[31:16]
		10	[47:32]
		11	[63:48]
[47:32]	[5:4]	00	[15:0]
		01	[31:16]
		10	[47:32]
		11	[63:48]
[63:48]	[7:6]	00	[15:0]
		01	[31:16]
		10	[47:32]
		11	[63:48]

The most-significant quadword of the source is copied to the corresponding quadword of the destination.

For the 256-bit form of the instruction, the same immediate byte selects words in the lower quadword of the upper 128 bits of the source operand to be copied to the destination:

Destination Word	Immediate-Byte Bit Field	Value of Bit Field	Source Word
[143:128]	[1:0]	00	[143:128]
		01	[159:144]
		10	[175:160]
		11	[191:176]

Destination Word	Immediate-Byte Bit Field	Value of Bit Field	Source Word
[159:144]	[3:2]	00	[143:128]
		01	[159:144]
		10	[175:160]
		11	[191:176]
[175:160]	[5:4]	00	[143:128]
		01	[159:144]
		10	[175:160]
		11	[191:176]
[191:176]	[7:6]	00	[143:128]
		01	[159:144]
		10	[175:160]
		11	[191:176]

The most-significant quadword of the upper 128 bits of the source is copied to the corresponding quadword of the destination.

There are legacy and extended forms of the instruction:

PSHUFLW

The source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSHUFLW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
PSHUFLW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSHUFLW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSHUFLW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSHUFLW *xmm1*, *xmm2*/*mem128*, *imm8*	F2 0F 70 /r ib	Copies packed 16-bit values from the low-order quadword of *xmm2* or *mem128* to the low-order quadword of *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSHUFLW *xmm1*, *xmm2/mem128*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.1111.0.11	70 /r ib
VPSHUFLW *ymm1*, *ymm2/mem256*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.1111.1.11	70 /r ib

Related Instructions

(V)PSHUFD, (V)PSHUFHW, (V)PSHUFW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSIGNB VPSIGNB — Packed Sign Byte

For each packed signed byte in the first source operand, evaluate the corresponding byte of the second source operand and perform one of the following operations.

- When a byte of the second source is negative, write the two's-complement of the corresponding byte of the first source to the destination.
- When a byte of the second source is positive, copy the corresponding byte of the first source to the destination.
- When a byte of the second source is zero, clear the corresponding byte of the destination.

There are legacy and extended forms of the instruction:

PSIGNB

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSIGNB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSIGNB	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPSIGNB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSIGNB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSIGNB *xmm1*, *xmm2*/*mem128*	66 0F 38 08 /r	Perform operation based on evaluation of each packed 8-bit signed integer value in *xmm2* or *mem128*. Write 8-bit signed results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSIGNB *xmm1*, xmm2*, xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.0.01	08 /r
VPSIGNB *ymm1*, ymm2*, ymm2/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{src1}$.1.01	08 /r

Related Instructions

(V)PSIGNW, (V)PSIGND

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSIGND VPSIGND — Packed Sign Doubleword

For each packed signed doubleword in the first source operand, evaluate the corresponding doubleword of the second source operand and perform one of the following operations.

- When a doubleword of the second source is negative, write the two's-complement of the corresponding doubleword of the first source to the destination.
- When a doubleword of the second source is positive, copy the corresponding doubleword of the first source to the destination.
- When a doubleword of the second source is zero, clear the corresponding doubleword of the destination.

There are legacy and extended forms of the instruction:

PSIGND

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSIGND

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSIGND	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPSIGND 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSIGND 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSIGND *xmm1*, *xmm2*/*mem128*	66 0F 38 0A /r	Perform operation based on evaluation of each packed 32-bit signed integer value in *xmm2* or *mem128*. Write 32-bit signed results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSIGND *xmm1*, xmm2*, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.0.01	0A /r
VPSIGND *ymm1*, ymm2*, ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.1.01	0A /r

Related Instructions

(V)PSIGNB, (V)PSIGNW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSIGNW VPSIGNW — Packed Sign Word

For each packed signed word in the first source operand, evaluate the corresponding word of the second source operand and perform one of the following operations.

- When a word of the second source is negative, write the two's-complement of the corresponding word of the first source to the destination.
- When a word of the second source is positive, copy the corresponding word of the first source to the destination.
- When a word of the second source is zero, clear the corresponding word of the destination.

There are legacy and extended forms of the instruction:

PSIGNW

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSIGNW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSIGNW	SSSE3	CPUID Fn0000_0001_ECX[SSSE3] (bit 9)
VPSIGNW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSIGNW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSIGNW *xmm1*, *xmm2*/*mem128*	66 0F 38 09 /r	Perform operation based on evaluation of each packed 16-bit signed integer value in *xmm2* or *mem128*. Write 16-bit signed results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSIGNW *xmm1*, xmm2*, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.0.01	09 /r
VPSIGNW *ymm1*, ymm2*, ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	X.$\overline{\textit{src1}}$.1.01	09 /r

Related Instructions

(V)PSIGNB, (V)PSIGND

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSLLD VPSLLD Packed Shift Left Logical Doublewords

Left-shifts each packed 32-bit value in the source operand as specified by a shift-count operand and writes the shifted values to the destination.

The shift-count operand can be an immediate byte, a second register, or a memory location. The shift count is treated as an unsigned integer. When the shift count is provided by a register or memory location, only bits [63:0] of the value are considered.

Low-order bits emptied by shifting are cleared. When the shift count is greater than 31, the destination is cleared.

There are legacy and extended forms of the instruction:

PSLLD

There are two forms of the instruction, based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSLLD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

There are two 128-bit encodings. These differ based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is an XMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

There are two 256-bit encodings. These differ based on the type of count operand.

The first source operand is a YMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is a YMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv.

Instruction Support

Form	Subset	Feature Flag
PSLLD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSLLD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSLLD 256-bit	AVX2	CPUID Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSLLD *xmm1*, *xmm2/mem128*	66 0F F2 /r	Left-shifts packed doublewords in *xmm1* as specified by *xmm2[63:0]* or *mem128[63:0]*.
PSLLD *xmm*, *imm8*	66 0F 72 /6 ib	Left-shifts packed doublewords in *xmm* as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSLLD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	F2 /r
VPSLLD *xmm1*, *xmm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.0.01	72 /6 ib
VPSLLD *ymm1*, *ymm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	F2 /r
VPSLLD *ymm1*, *ymm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.1.01	72 /6 ib

Related Instructions

(V)PSLLDQ, (V)PSLLQ, (V)PSLLW, (V)PSRAD, (V)PSRAW, (V)PSRLD, (V)PSRLDQ, (V)PSRLQ, (V)PSRLW, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	When alignment checking enabled: • 128-bit memory operand not 16-byte aligned. • 256-bit memory operand not 32-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.

X — AVX, AVX2, and SSE exception
A — AVX and AVX2 exception
S — SSE exception

PSLLDQ VPSLLDQ — Packed Shift Left Logical Double Quadword

Left-shifts the one or each of the two double quadword values in the source operand the number of bytes specified by an immediate byte operand and writes the shifted values to the destination.

The immediate byte operand supplies an unsigned shift count. Low-order bytes emptied by shifting are cleared. When the shift value is greater than 15, the destination is cleared. For the 256-bit form of the instruction, the shift count is applied to both the upper and the lower double quadword. Bytes shifted out of the lower 128 bits are not shifted into the upper.

There are legacy and extended forms of the instruction:

PSLLDQ

The source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSLLDQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is an XMM register. The destination is an XMM register specified by VEX.vvvv. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is a YMM register. The destination is a YMM register specified by VEX.vvvv.

Instruction Support

Form	Subset	Feature Flag
PSLLDQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSLLDQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSLLDQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSLLDQ *xmm*, *imm8*	66 0F 73 /7 ib	Left-shifts double quadword value in *xmm1* as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSLLDQ *xmm1*, xmm2, *imm8*	C4	$\overline{\text{RXB}}$.01	0.$\overline{\text{dest}}$.0.01	73 /7 ib
VPSLLDQ *ymm1*, ymm2, *imm8*	C4	$\overline{\text{RXB}}$.01	0.$\overline{\text{dest}}$.1.01	73 /7 ib

Related Instructions

(V)PSLLD, (V)PSLLQ, (V)PSLLW, (V)PSRAD, (V)PSRAW, (V)PSRLD, (V)PSRLDQ, (V)PSRLQ, (V)PSRLW, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSLLQ VPSLLQ Packed Shift Left Logical Quadwords

Left-shifts each packed 64-bit value in the source operand as specified by a shift-count operand and writes the shifted values to the destination.

The shift-count operand can be an immediate byte, a second register, or a memory location. The shift count is treated as an unsigned integer. When the shift count is provided by a register or memory location, only bits [63:0] of the value are considered.

Low-order bits emptied by shifting are cleared. When the shift value is greater than 63, the destination is cleared.

There are legacy and extended forms of the instruction:

PSLLQ

There are two forms of the instruction, based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSLLQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

There are two 128-bit encodings. These differ based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is an XMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

There are two 256-bit encodings. These differ based on the type of count operand.

The first source operand is a YMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is a YMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv.

Instruction Support

Form	Subset	Feature Flag
PSLLQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSLLQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSLLQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSLLQ *xmm1*, *xmm2/mem128*	66 0F F3 /r	Left-shifts packed quadwords in *xmm1* as specified by *xmm2[63:0]* or *mem128[63:0]*.
PSLLQ *xmm*, *imm8*	66 0F 73 /6 ib	Left-shifts packed quadwords in *xmm* as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSLLQ *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	F3 /r
VPSLLQ *xmm1*, *xmm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.0.01	73 /6 ib
VPSLLQ *ymm1*, *ymm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	F3 /r
VPSLLQ *ymm1*, *ymm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.1.01	73 /6 ib

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLW, (V)PSRAD, (V)PSRAW, (V)PSRLD, (V)PSRLDQ, (V)PSRLQ, (V)PSRLW, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQLLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	When alignment checking enabled: • 128-bit memory operand not 16-byte aligned. • 256-bit memory operand not 32-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.

X — AVX, AVX2, and SSE exception
A — AVX and AVX2 exception
S — SSE exception

PSLLW VPSLLW — Packed Shift Left Logical Words

Left-shifts each packed 16-bit value in the source operand as specified by a shift-count operand and writes the shifted values to the destination.

The shift-count operand can be an immediate byte, a second register, or a memory location. The shift count is treated as an unsigned integer. When the shift count is provided by a register or memory location, only bits [63:0] of the value are considered.

Low-order bits emptied by shifting are cleared. When the shift count is greater than 15, the destination is cleared.

There are legacy and extended forms of the instruction:

PSLLW

There are two forms of the instruction, based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSLLW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

There are two 128-bit encodings. These differ based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is an XMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

There are two 256-bit encodings. These differ based on the type of count operand.

The first source operand is a YMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is a YMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv.

Instruction Support

Form	Subset	Feature Flag
PSLLW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSLLW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSLLW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSLLW *xmm1*, *xmm2/mem128*	66 0F F1 /r	Left-shifts packed words in *xmm1* as specified by *xmm2[63:0]* or *mem128[63:0]*.
PSLLW *xmm*, *imm8*	66 0F 71 /6 ib	Left-shifts packed words in *xmm* as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSLLW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	F1 /r
VPSLLW *xmm1*, *xmm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.0.01	71 /6 ib
VPSLLW *ymm1*, *ymm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	F1 /r
VPSLLW *ymm1*, *ymm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.1.01	71 /6 ib

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLQ, (V)PSRAD, (V)PSRAW, (V)PSRLD, (V)PSRLDQ, (V)PSRLQ, (V)PSRLW, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	When alignment checking enabled: • 128-bit memory operand not 16-byte aligned. • 256-bit memory operand not 32-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.

X — AVX, AVX2, and SSE exception
A — AVX and AVX2 exception
S — SSE exception

PSRAD VPSRAD Packed Shift Right Arithmetic Doublewords

Right-shifts each packed 32-bit value in the source operand as specified by a shift-count operand and writes the shifted values to the destination.

The shift-count operand can be an immediate byte, a second register, or a memory location. The shift count is treated as an unsigned integer. When the shift count is provided by a register or memory location, only bits [63:0] of the value are considered.

High-order bits emptied by shifting are filled with the sign bit of the initial value. When the shift value is greater than 31, each doubleword of the destination is filled with the sign bit of its initial value.

There are legacy and extended forms of the instruction:

PSRAD

There are two forms of the instruction, based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSRAD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

There are two 128-bit encodings. These differ based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is an XMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

There are two 256-bit encodings. These differ based on the type of count operand.

The first source operand is a YMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is a YMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv.

Instruction Support

Form	Subset	Feature Flag
PSRAD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSRAD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSRAD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSRAD *xmm1*, *xmm2*/*mem128*	66 0F E2 /r	Right-shifts packed doublewords in *xmm1* as specified by *xmm2[63:0]* or *mem128[63:0]*.
PSRAD *xmm*, *imm8*	66 0F 72 /4 ib	Right-shifts packed doublewords in *xmm* as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSRAD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	E2 /r
VPSRAD *xmm1*, *xmm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.0.01	72 /4 ib
VPSRAD *ymm1*, *ymm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	E2 /r
VPSRAD *ymm1*, *ymm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.1.01	72 /4 ib

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLQ, (V)PSLLW, (V)PSRAW, (V)PSRLD, (V)PSRLDQ, (V)PSRLQ, (V)PSRLW, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	When alignment checking enabled: • 128-bit memory operand not 16-byte aligned. • 256-bit memory operand not 32-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.

X — AVX, AVX2, and SSE exception
A — AVX and AVX2 exception
S — SSE exception

PSRAW VPSRAW — Packed Shift Right Arithmetic Words

Right-shifts each packed 16-bit value in the source operand as specified by a shift-count operand and writes the shifted values to the destination.

The shift-count operand can be an immediate byte, a second register, or a memory location. The shift count is treated as an unsigned integer. When the shift count is provided by a register or memory location, only bits [63:0] of the value are considered.

High-order bits emptied by shifting are filled with the sign bit of the initial value. When the shift value is greater than 16, each doubleword of the destination is filled with the sign bit of its initial value.

There are legacy and extended forms of the instruction:

PSRAW

There are two forms of the instruction, based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSRAW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

There are two 128-bit encodings. These differ based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is an XMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

There are two 256-bit encodings. These differ based on the type of count operand.

The first source operand is a YMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is a YMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv.

Instruction Support

Form	Subset	Feature Flag
PSRAW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSRAW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSRAW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSRAW *xmm1*, *xmm2/mem128*	66 0F E1 /r	Right-shifts packed words in *xmm1* as specified by *xmm2[63:0]* or *mem128[63:0]*.
PSRAW *xmm*, *imm8*	66 0F 71 /4 ib	Right-shifts packed words in *xmm* as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSRAW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	E1 /r
VPSRAW *xmm1*, *xmm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.0.01	71 /4 ib
VPSRAW *ymm1*, *ymm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	E1 /r
VPSRAW *ymm1*, *ymm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.1.01	71 /4 ib

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLQ, (V)PSLLW, (V)PSRAD, (V)PSRLD, (V)PSRLDQ, (V)PSRLQ, (V)PSRLW, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	When alignment checking enabled: • 128-bit memory operand not 16-byte aligned. • 256-bit memory operand not 32-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.

X — AVX, AVX2, and SSE exception
A — AVX and AVX2 exception
S — SSE exception

PSRLD VPSRLD — Packed Shift Right Logical Doublewords

Right-shifts each packed 32-bit value in the source operand as specified by a shift-count operand and writes the shifted values to the destination.

The shift-count operand can be an immediate byte, a second register, or a memory location. The shift count is treated as an unsigned integer. When the shift count is provided by a register or memory location, only bits [63:0] of the value are considered.

High-order bits emptied by shifting are cleared. When the shift value is greater than 31, the destination is cleared.

There are legacy and extended forms of the instruction:

PSRLD

There are two forms of the instruction, based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSRLD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

There are two 128-bit encodings. These differ based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is an XMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

There are two 256-bit encodings. These differ based on the type of count operand.

The first source operand is a YMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is a YMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv.

Instruction Support

Form	Subset	Feature Flag
PSRLD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSRLD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSRLD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSRLD *xmm1*, *xmm2/mem128*	66 0F D2 /r	Right-shifts packed doublewords in *xmm1* as specified by *xmm2[63:0]* or *mem128[63:0]*.
PSRLD *xmm*, *imm8*	66 0F 72 /2 ib	Right-shifts packed doublewords in *xmm* as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSRLD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	D2 /r
VPSRLD *xmm1*, *xmm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.0.01	72 /2 ib
VPSRLD *ymm1*, *ymm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	D2 /r
VPSRLD *ymm1*, *ymm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.1.01	72 /2 ib

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLQ, (V)PSLLW, (V)PSRAD, (V)PSRAW, (V)PSRLDQ, (V)PSRLQ, (V)PSRLW, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	When alignment checking enabled: • 128-bit memory operand not 16-byte aligned. • 256-bit memory operand not 32-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.

X — AVX, AVX2, and SSE exception
A — AVX and AVX2 exception
S — SSE exception

PSRLDQ VPSRLDQ — Packed Shift Right Logical Double Quadword

Right-shifts one or each of two double quadword values in the source operand the number of bytes specified by an immediate byte operand and writes the shifted values to the destination.

The immediate byte operand supplies an unsigned shift count. High-order bytes emptied by shifting are cleared. When the shift value is greater than 15, the destination is cleared. For the 256-bit form of the instruction, the shift count is applied to both the upper and the lower double quadword. Bytes shifted out of the upper 128 bits are not shifted into the lower.

There are legacy and extended forms of the instruction:

PSRLDQ

The source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSRLDQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The source operand is an XMM register. The destination is an XMM register specified by VEX.vvvv. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The source operand is a YMM register. The destination is a YMM register specified by VEX.vvvv.

Instruction Support

Form	Subset	Feature Flag
PSRLDQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSRLDQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSRLDQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSRLDQ *xmm*, *imm8*	66 0F 73 /3 ib	Right-shifts double quadword value in *xmm1* as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSRLDQ *xmm1*, *xmm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.0.01	73 /3 ib
VPSRLDQ *ymm1*, *ymm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.1.01	73 /3 ib

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLQ, (V)PSLLW, (V)PSRAD, (V)PSRAW, (V)PSRLD, (V)PSRLQ, (V)PSRLW, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSRLQ VPSRLQ Packed Shift Right Logical Quadwords

Right-shifts each packed 64-bit value in the source operand as specified by a shift-count operand and writes the shifted values to the destination.

The shift-count operand can be an immediate byte, a second register, or a memory location. The shift count is treated as an unsigned integer. When the shift count is provided by a register or memory location, only bits [63:0] of the value are considered.

High-order bits emptied by shifting are cleared. When the shift value is greater than 63, the destination is cleared.

There are legacy and extended forms of the instruction:

PSRLQ

There are two forms of the instruction, based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSRLQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

There are two 128-bit encodings. These differ based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is an XMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

There are two 256-bit encodings. These differ based on the type of count operand.

The first source operand is a YMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is a YMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv.

Instruction Support

Form	Subset	Feature Flag
PSRLQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSRLQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSRLQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSRLQ *xmm1*, *xmm2*/*mem128*	66 0F D3 /r	Right-shifts packed quadwords in *xmm1* as specified by *xmm2[63:0]* or *mem128[63:0]*.
PSRLQ *xmm*, *imm8*	66 0F 73 /2 ib	Right-shifts packed quadwords in *xmm* as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSRLQ *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	D3 /r
VPSRLQ *xmm1*, *xmm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.0.01	73 /2 ib
VPSRLQ *ymm1*, *ymm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	D3 /r
VPSRLQ *ymm1*, *ymm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.1.01	73 /2 ib

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLQ, (V)PSLLW, (V)PSRAD, (V)PSRAW, (V)PSRLD, (V)PSRLDQ, (V)PSRLW, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	When alignment checking enabled: • 128-bit memory operand not 16-byte aligned. • 256-bit memory operand not 32-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.

X — AVX, AVX2, and SSE exception
A — AVX and AVX2 exception
S — SSE exception

PSRLW VPSRLW — Packed Shift Right Logical Words

Right-shifts each packed 16-bit value in the source operand as specified by a shift-count operand and writes the shifted values to the destination.

The shift-count operand can be an immediate byte, a second register, or a memory location. The shift count is treated as an unsigned integer. When the shift count is provided by a register or memory location, only bits [63:0] of the value are considered.

High-order bits emptied by shifting are cleared. When the shift value is greater than 15, the destination is cleared.

There are legacy and extended forms of the instruction:

PSRLW

There are two forms of the instruction, based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSRLW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

There are two 128-bit encodings. These differ based on the type of count operand.

The first source operand is an XMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is an XMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

There are two 256-bit encodings. These differ based on the type of count operand.

The first source operand is a YMM register. The shift count is specified by either a second XMM register or a 128-bit memory location, or by an immediate 8-bit operand. The destination is a YMM register. For the immediate operand encoding, the destination is specified by VEX.vvvv.

Instruction Support

Form	Subset	Feature Flag
PSRLW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSRLW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSRLW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSRLW *xmm1*, *xmm2*/*mem128*	66 0F D1 /r	Right-shifts packed words in *xmm1* as specified by *xmm2[63:0]* or *mem128[63:0]*.
PSRLW *xmm*, *imm8*	66 0F 71 /2 ib	Right-shifts packed words in *xmm* as specified by *imm8*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSRLW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	D1 /r
VPSRLW *xmm1*, *xmm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.0.01	71 /2 ib
VPSRLW *ymm1*, *ymm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	D1 /r
VPSRLW *ymm1*, *ymm2*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\text{dest}}$.1.01	71 /2 ib

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLQ, (V)PSLLW, (V)PSRAD, (V)PSRAW, (V)PSRLD, (V)PSRLDQ, (V)PSRLQ, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	When alignment checking enabled: • 128-bit memory operand not 16-byte aligned. • 256-bit memory operand not 32-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.

X — AVX, AVX2, and SSE exception
A — AVX and AVX2 exception
S — SSE exception

PSUBB VPSUBB — Packed Subtract Bytes

Subtracts 16 or 32 packed 8-bit integer values in the second source operand from the corresponding values in the first source operand and writes the integer differences to the corresponding bytes of the destination.

This instruction operates on both signed and unsigned integers. When a result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 8 bits of each result are written to the destination.

There are legacy and extended forms of the instruction:

PSUBB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSUBB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSUBB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSUBB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSUBB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSUBB *xmm1*, *xmm2*/*mem128*	66 0F F8 /r	Subtracts 8-bit signed integer values in *xmm2* or *mem128* from corresponding values in *xmm1*. Writes differences to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSUBB *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	F8 /r
VPSUBB *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	F8 /r

Related Instructions

(V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PSUBUSB, (V)PSUBUSW, (V)PSUBW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSUBD VPSUBD — Packed Subtract Doublewords

Subtracts four or eight packed 32-bit integer values in the second source operand from the corresponding values in the first source operand and writes the integer differences to the corresponding doubleword of the destination.

This instruction operates on both signed and unsigned integers. When a result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 8 bits of each result are written to the destination.

There are legacy and extended forms of the instruction:

PSUBD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VSUBD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSUBD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSUBD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSUBD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSUBD *xmm1*, *xmm2*/*mem128*	66 0F FA /r	Subtracts packed 32-bit integer values in *xmm2* or *mem128* from corresponding values in *xmm1*. Writes the differences to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSUBD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	FA /r
VPSUBD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	FA /r

Related Instructions

(V)PSUBB, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PSUBUSB, (V)PSUBUSW, (V)PSUBW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSUBQ VPSUBQ Packed Subtract Quadword

Subtracts two or four packed 64-bit integer values in the second source operand from the corresponding values in the first source operand and writes the differences to the corresponding quadword of the destination.

This instruction operates on both signed and unsigned integers. When a result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 8 bits of each result are written to the destination.

There are legacy and extended forms of the instruction:

PSUBQ

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VSUBQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSUBQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSUBQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSUBQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSUBQ *xmm1*, *xmm2*/*mem128*	66 0F FB /r	Subtracts packed 64-bit integer values in *xmm2* or *mem128* from corresponding values in *xmm1*. Writes the differences to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSUBQ *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	FB /r
VPSUBQ *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	FB /r

Related Instructions

(V)PSUBB, (V)PSUBD, (V)PSUBSB, (V)PSUBSW, (V)PSUBUSB, (V)PSUBUSW, (V)PSUBW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSUBSB VPSUBSB Packed Subtract Signed With Saturation Bytes

Subtracts 16 or 32 packed 8-bit signed integer value in the second source operand from the corresponding values in the first source operand and writes the signed integer differences to the corresponding byte of the destination.

For each packed value in the destination, if the value is larger than the largest signed 8-bit integer, it is saturated to 7Fh, and if the value is smaller than the smallest signed 8-bit integer, it is saturated to 80h.

There are legacy and extended forms of the instruction:

PSUBSB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSUBSB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSUBSB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSUBSB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSUBSB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSUBSB *xmm1*, *xmm2/mem128*	66 0F E8 /r	Subtracts packed 8-bit signed integer values in *xmm2* or *mem128* from corresponding values in *xmm1*. Writes the differences to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSUBSB *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	E8 /r
VPSUBSB *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	E8 /r

Related Instructions

(V)PSUBB, (V)PSUBD, (V)PSUBQ, (V)PSUBSW, (V)PSUBUSB, (V)PSUBUSW, (V)PSUBW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSUBSW VPSUBSW Packed Subtract Signed With Saturation Words

Subtracts eight or sixteen packed 16-bit signed integer values in the second source operand from the corresponding values in the first source operand and writes the signed integer differences to the corresponding word of the destination.

Positive differences greater than 7FFFh are saturated to 7FFFh; negative differences less than 8000h are saturated to 8000h.

There are legacy and extended forms of the instruction:

PSUBSW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSUBSW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSUBSW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSUBSW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSUBSW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSUBSW *xmm1*, *xmm2*/*mem128*	66 0F E9 /r	Subtracts packed 16-bit signed integer values in *xmm2* or *mem128* from corresponding values in *xmm1*. Writes the differences to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSUBSW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	E9 /r
VPSUBSW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	E9 /r

Related Instructions

(V)PSUBB, (V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBUSB, (V)PSUBUSW, (V)PSUBW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSUBUSB VPSUBUSB Packed Subtract Unsigned With Saturation Bytes

Subtracts 16 or 32 packed 8-bit unsigned integer value in the second source operand from the corresponding values in the first source operand and writes the unsigned integer difference to the corresponding byte of the destination.

Differences less than 00h are saturated to 00h.

There are legacy and extended forms of the instruction:

PSUBUSB

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSUBUSB

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSUBUSB	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSUBUSB 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSUBUSB 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSUBUSB *xmm1*, *xmm2*/*mem128*	66 0F D8 /r	Subtracts packed byte unsigned integer values in *xmm2* or *mem128* from corresponding values in *xmm1*. Writes the differences to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSUBUSB *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	D8 /r
VPSUBUSB *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	D8 /r

Related Instructions

(V)PSUBB, (V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PSUBUSW, (V)PSUBW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSUBUSW VPSUBUSW Packed Subtract Unsigned With Saturation Words

Subtracts eight or sixteen packed 16-bit unsigned integer value in the second source operand from the corresponding values in the first source operand and writes the unsigned integer differences to the corresponding word of the destination.

Differences less than 0000h are saturated to 0000h.

There are legacy and extended forms of the instruction:

PSUBUSW

The first source operand is an XMM register and the second source operand is an XMM register or 128-bit memory location. The first source operand is also the destination register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSUBUSW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSUBUSW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSUBUSW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSUBUSW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSUBUSW *xmm1*, *xmm2*/*mem128*	66 0F D9 /r	Subtracts packed 16-bit unsigned integer values in *xmm2* or *mem128* from corresponding values in *xmm1*. Writes the differences to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSUBUSW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	D9 /r
VPSUBUSW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	D9 /r

Related Instructions

(V)PSUBB, (V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PSUBUSB, (V)PSUBW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PSUBW VPSUBW — Packed Subtract Words

Subtracts eight or sixteen packed 16-bit integer values in the second source operand from the corresponding values in the first source operand and writes the integer differences to the corresponding word of the destination.

This instruction operates on both signed and unsigned integers. When a result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 8 bits of each result are written to the destination.

There are legacy and extended forms of the instruction:

PSUBW

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPSUBW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PSUBW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPSUBW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPSUBW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PSUBW *xmm1*, *xmm2/mem128*	66 0F F9 /r	Subtracts packed 16-bit integer values in *xmm2* or *mem128* from corresponding values in *xmm1*. Writes the differences to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSUBW *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	F9 /r
VPSUBW *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	F9 /r

Related Instructions

(V)PSUBB, (V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PSUBUSB, (V)PSUBUSW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PTEST Packed Bit Test
VPTEST

First, performs a bitwise AND of the first source operand with the second source operand. Sets rFLAGS.ZF when all bit operations = 0; else, clears ZF.

Second. performs a bitwise AND of the second source operand with the logical complement (NOT) of the first source operand. Sets rFLAGS.CF when all bit operations = 0; else, clears CF.

Neither source operand is modified.

There are legacy and extended forms of the instruction:

PTEST

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location.

VPTEST

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location.

YMM Encoding

The first source operand is a YMM register. The second source operand is a YMM register or 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
PTEST	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPTEST	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PTEST *xmm1*, *xmm2/mem128*	66 0F 38 17 /r	Set ZF if bitwise AND of *xmm2/m128* with *xmm1* = 0; else, clear ZF. Set CF if bitwise AND of *xmm2/m128* with NOT*xmm1* = 0; else, clear CF.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPTEST *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.00010	X.1111.0.01	17 /r
VPTEST *ymm1*, *ymm2*/*mem256*	C4	$\overline{\text{RXB}}$.00010	X.1111.1.01	17 /r

Related Instructions

VTESTPD, VTESTPS

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	M	0	0	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3 and 1 are reserved. A flag set or cleared is M (modified). Unaffected flags are blank. Undefined flags are U.*

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

PUNPCKHBW VPUNPCKHBW — Unpack and Interleave High Bytes

Unpacks the 8 high-order bytes of each octword the first and second source operands and interleaves the bytes as they are copied to the destination. The low-order bytes of each octword of the source operands are ignored.

Bytes are interleaved in ascending order from the least-significant byte of the upper 8 bytes of each octword of the source operands with bytes from the first source operand occupying the lower byte of each pair copied to the destination.

For the 128-bit form of the instruction, the following operations are performed:

```
dest[7:0] = src1[71:64]
dest[15:8] = src2[71:64]
dest[23:16] = src1[79:72]
dest[31:24] = src2[79:72]
dest[39:32] = src1[87:80]
dest[47:40] = src2[87:80]
dest[55:48] = src1[95:88]
dest[63:56] = src2[95:88]
dest[71:64] = src1[103:96]
dest[79:72] = src2[103:96]
dest[87:80] = src1[111:104]
dest[95:88] = src2[111:104]
dest[103:96] = src1[119:112]
dest[111:104] = src2[119:112]
dest[119:112] = src1[127:120]
dest[127:120] = src2[127:120]
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[135:128] = src1[199:192]
dest[143:136] = src2[199:192]
dest[151:144] = src1[207:200]
dest[159:152] = src2[207:200]
dest[167:160] = src1[215:208]
dest[175:168] = src2[215:208]
dest[183:176] = src1[223:216]
dest[191:184] = src2[223:216]
dest[199:192] = src1[231:224]
dest[207:200] = src2[231:224]
dest[215:208] = src1[239:232]
dest[223:216] = src2[239:232]
dest[231:224] = src1[247:240]
dest[239:232] = src2[247:240]
dest[247:240] = src1[255:248]
dest[255:248] = src2[255:248]
```

When the second source operand is all 0s, the destination effectively contains the 8 high-order bytes from the first source operand or the 8 high-order bytes from both octwords of the first source operand zero-extended to 16 bits. This operation is useful for expanding unsigned 8-bit values to unsigned 16-bit operands for subsequent processing that requires higher precision.

There are legacy and extended forms of the instruction:

PUNPCKHBW

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPUNPCKHBW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PUNPCKHBW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPUNPCKHBW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPUNPCKHBW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PUNPCKHBW *xmm1*, *xmm2*/*mem128*	66 0F 68 /r	Unpacks and interleaves the high-order bytes of *xmm1* and *xmm2* or *mem128*. Writes the bytes to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPUNPCKHBW *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	68 /r
VPUNPCKHBW *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	68 /r

Related Instructions

(V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKHWD, (V)PUNPCKLBW, (V)PUNPCKLDQ, (V)PUNPCKLQDQ, (V)PUNPCKLWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PUNPCKHDQ VPUNPCKHDQ — Unpack and Interleave High Doublewords

Unpacks the two high-order doublewords of each octword of the first and second source operands and interleaves the doublewords as they are copied to the destination. The low-order doublewords of each octword of the source operands are ignored.

Doublewords are interleaved in ascending order from the least-significant doubleword of the high quadword of each octword with doublewords from the first source operand occupying the lower doubleword of each pair copied to the destination.

For the 128-bit form of the instruction, the following operations are performed:

```
dest[31:0] = src1[95:64]
dest[63:32] = src2[95:64]
dest[95:64] = src1[127:96]
dest[127:96] = src2[127:96]
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[159:128] = src1[223:192]
dest[191:160] = src2[223:192]
dest[223:192] = src1[255:224]
dest[255:224] = src2[255:224]
```

When the second source operand is all 0s, the destination effectively receives the 2 high-order doublewords from the first source operand or the 2 high-order doublewords from both octwords of the first source operand zero-extended to 64 bits. This operation is useful for expanding unsigned 32-bit values to unsigned 64-bit operands for subsequent processing that requires higher precision.

There are legacy and extended forms of the instruction:

PUNPCKHDQ

The first source operand is an XMM register and the second source operand is an XMM register or 128-bit memory location. The first source operand is also the destination register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPUNPCKHDQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PUNPCKHDQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPUNPCKHDQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPUNPCKHDQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PUNPCKHDQ *xmm1*, *xmm2*/*mem128*	66 0F 6A /r	Unpacks and interleaves the high-order doublewords of *xmm1* and *xmm2* or *mem128*. Writes the doublewords to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPUNPCKHDQ *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	6A /r
VPUNPCKHDQ *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	6A /r

Related Instructions

(V)PUNPCKHBW, (V)PUNPCKHQDQ, (V)PUNPCKHWD, (V)PUNPCKLBW, (V)PUNPCKLDQ, (V)PUNPCKLQDQ, (V)PUNPCKLWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PUNPCKHQDQ VPUNPCKHQDQ Unpack and Interleave High Quadwords

Unpacks the high-order quadword of each octword of the first and second source operands and interleaves the quadwords as they are copied to the destination. The low-order quadword of each octword of the source operands is ignored.

Quadwords are interleaved in ascending order with the high-order quadword from the first source operand or each octword of the first source operand occupying the lower quadword of corresponding octword of the destination.

For the 128-bit form of the instruction, the following operations are performed:

```
dest[63:0] = src1[127:64]
dest[127:64] = src2[127:64]
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[191:128] = src1[255:192]
dest[255:192] = src2[255:192]
```

When the second source operand is all 0s, the destination effectively receives the quadword from upper half of the first source operand or the high-order quadwords from each octword of the first source operand zero-extended to 128 bits. This operation is useful for expanding unsigned 64-bit values to unsigned 128-bit operands for subsequent processing that requires higher precision.

There are legacy and extended forms of the instruction:

PUNPCKHQDQ

The first source operand is an XMM register and the second source operand is an XMM register or 128-bit memory location. The first source operand is also the destination register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPUNPCKHQDQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PUNPCKHQDQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPUNPCKHQDQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPUNPCKHQDQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PUNPCKHQDQ *xmm1*, *xmm2*/*mem128*	66 0F 6D /r	Unpacks and interleaves the high-order quadwords of *xmm1* and *xmm2* or *mem128*. Writes the bytes to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPUNPCKHQDQ *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	6D /r
VPUNPCKHQDQ *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	6D /r

Related Instructions

(V)PUNPCKHBW, (V)PUNPCKHDQ, (V)PUNPCKHWD, (V)PUNPCKLBW, (V)PUNPCKLDQ, (V)PUNPCKLQDQ, (V)PUNPCKLWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PUNPCKHWD VPUNPCKHWD — Unpack and Interleave High Words

Unpacks the 4 high-order words of each octword of the first and second source operands and interleaves the words as they are copied to the destination. The low-order words of each octword of the source operands are ignored.

Words are interleaved in ascending order from the least-significant word of the high quadword of each octword with words from the first source operand occupying the lower word of each pair copied to the destination.

For the 128-bit form of the instruction, the following operations are performed:

```
dest[15:0] = src1[79:64]
dest[31:16] = src2[79:64]
dest[47:32] = src1[95:80]
dest[63:48] = src2[95:80]
dest[79:64] = src1[111:96]
dest[95:80] = src2[111:96]
dest[111:96] = src1[127:112]
dest[127:112] = src2[127:112]
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[143:128] = src1[207:192]
dest[159:144] = src2[207:192]
dest[175:160] = src1[223:208]
dest[191:176] = src2[223:208]
dest[207:192] = src1[239:224]
dest[223:208] = src2[239:224]
dest[239:224] = src1[255:240]
dest[255:240] = src2[255:240]
```

When the second source operand is all 0s, the destination effectively receives the 4 high-order words from the first source operand or the 4 high-order words from both octwords of the first source operand and zero-extended to 32 bits. This operation is useful for expanding unsigned 16-bit values to unsigned 32-bit operands for subsequent processing that requires higher precision.

There are legacy and extended forms of the instruction:

PUNPCKHWD

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPUNPCKHWD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PUNPCKHWD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPUNPCKHWD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPUNPCKHWD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PUNPCKHWD *xmm1*, *xmm2*/*mem128*	66 0F 69 /r	Unpacks and interleaves the high-order words of *xmm1* and *xmm2* or *mem128*. Writes the words to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPUNPCKHWD *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	69 /r
VPUNPCKHWD *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	69 /r

Related Instructions

(V)PUNPCKHBW, (V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKLBW, (V)PUNPCKLDQ, (V)PUNPCKLQDQ, (V)PUNPCKLWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PUNPCKLBW VPUNPCKLBW — Unpack and Interleave Low Bytes

Unpacks the 8 low-order bytes of each octword of the first and second source operands and interleaves the bytes as they are copied to the destination. The high-order bytes of each octword are ignored.

Bytes are interleaved in ascending order from the least-significant byte of source operands with bytes from the first source operand occupying the lower byte of each pair copied to the destination.

For the 128-bit form of the instruction, the following operations are performed:

```
dest[7:0] = src1[7:0]
dest[15:8] = src2[7:0]
dest[23:16] = src1[15:8]
dest[31:24] = src2[15:8]
dest[39:32] = src1[23:16]
dest[47:40] = src2[23:16]
dest[55:48] = src1[31:24]
dest[63:56] = src2[31:24]
dest[71:64] = src1[39:32]
dest[79:72] = src2[39:32]
dest[87:80] = src1[47:40]
dest[95:88] = src2[47:40]
dest[103:96] = src1[55:48]
dest[111:104] = src2[55:48]
dest[119:112] = src1[63:56]
dest[127:120] = src2[63:56]
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[135:128] = src1[135:128]
dest[143:136] = src2[135:128]
dest[151:144] = src1[143:136]
dest[159:152] = src2[143:136]
dest[167:160] = src1[151:144]
dest[175:168] = src2[151:144]
dest[183:176] = src1[159:152]
dest[191:184] = src2[159:152]
dest[199:192] = src1[167:160]
dest[207:200] = src2[167:160]
dest[215:208] = src1[175:168]
dest[223:216] = src2[175:168]
dest[231:224] = src1[183:176]
dest[239:232] = src2[183:176]
dest[247:240] = src1[191:184]
dest[255:248] = src2[191:184]
```

When the second source operand is all 0s, the destination effectively receives the eight low-order bytes from the first source operand or the eight low-order bytes from both octwords of the first source operand zero-extended to 16 bits. This operation is useful for expanding unsigned 8-bit values to unsigned 16-bit operands for subsequent processing that requires higher precision.

There are legacy and extended forms of the instruction:

PUNPCKLBW

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPUNPCKLBW

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PUNPCKLBW	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPUNPCKLBW 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPUNPCKLBW 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PUNPCKLBW *xmm1*, *xmm2/mem128*	66 0F 60 /r	Unpacks and interleaves the low-order bytes of *xmm1* and *xmm2* or *mem128*. Writes the bytes to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPUNPCKLBW *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	60 /r
VPUNPCKLBW *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	60 /r

Related Instructions

(V)PUNPCKHBW, (V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKHWD, (V)PUNPCKLDQ, (V)PUNPCKLQDQ, (V)PUNPCKLWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PUNPCKLDQ VPUNPCKLDQ — Unpack and Interleave Low Doublewords

Unpacks the two low-order doublewords of each octword of the first and second source operands and interleaves the doublewords as they are copied to the destination. The high-order doublewords of each octword of the source operands are ignored.

Doublewords are interleaved in ascending order from the least-significant doubleword of the sources with doublewords from the first source operand occupying the lower doubleword of each pair copied to the destination.

For the 128-bit form of the instruction, the following operations are performed:

```
dest[31:0] = src1[31:0]
dest[63:32] = src2[31:0]
dest[95:64] = src1[63:32]
dest[127:96] = src2[63:32]
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[159:128] = src1[159:128]
dest[191:160] = src2[159:128]
dest[223:192] = src1[191:160]
dest[255:224] = src2[191:160]
```

When the second source operand is all 0s, the destination effectively receives the two low-order doublewords from the first source operand or the two low-order doublewords from both octwords of the source operand zero-extended to 64 bits. This operation is useful for expanding unsigned 32-bit values to unsigned 64-bit operands for subsequent processing that requires higher precision.

There are legacy and extended forms of the instruction:

PUNPCKLDQ

The first source operand is an XMM register and the second source operand is an XMM register or 128-bit memory location. The first source operand is also the destination register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPUNPCKLDQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PUNPCKLDQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPUNPCKLDQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPUNPCKLDQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PUNPCKLDQ *xmm1*, *xmm2/mem128*	66 0F 62 /r	Unpacks and interleaves the low-order doublewords of *xmm1* and *xmm2* or *mem128*. Writes the doublewords to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPUNPCKLDQ *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	62 /r
VPUNPCKLDQ *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	62 /r

Related Instructions

(V)PUNPCKHW, (V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKHWD, (V)PUNPCKLBW, (V)PUNPCKLQDQ, (V)PUNPCKLWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PUNPCKLQDQ VPUNPCKLQDQ — Unpack and Interleave Low Quadwords

Unpacks the low-order quadword of each octword of the first and second source operands and interleaves the quadwords as they are copied to the destination. The high-order quadword of each octword of the source operands is ignored.

Quadwords are interleaved in ascending order from the least-significant quadword of the sources with quadwords from the first source operand occupying the lower quadword of each pair copied to the destination.

For the 128-bit form of the instruction, the following operations are performed:

```
dest[63:0] = src1[63:0]
dest[127:64] = src2[63:0]
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[191:128] = src1[191:128]
dest[255:192] = src2[191:128]
```

When the second source operand is all 0s, the destination effectively receives the low-order quadword from the first source operand or the low-order quadword of both octwords of the first source operand zero-extended to 128 bits. This operation is useful for expanding unsigned 64-bit values to unsigned 128-bit operands for subsequent processing that requires higher precision.

There are legacy and extended forms of the instruction:

PUNPCKLQDQ

The first source operand is an XMM register and the second source operand is an XMM register or 128-bit memory location. The first source operand is also the destination register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPUNPCKLQDQ

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PUNPCKLQDQ	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPUNPCKLQDQ 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPUNPCKLQDQ 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PUNPCKLQDQ *xmm1*, *xmm2*/*mem128*	66 0F 6C /r	Unpacks and interleaves the low-order quadwords of *xmm1* and *xmm2* or *mem128*. Writes the bytes to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPUNPCKLQDQ *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	6C /r
VPUNPCKLQDQ *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	6C /r

Related Instructions

(V)PUNPCKHBW, (V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKHWD, (V)PUNPCKLBW, (V)PUNPCKLDQ, (V)PUNPCKLWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PUNPCKLWD VPUNPCKLWD — Unpack and Interleave Low Words

Unpacks the four low-order words of each octword of the first and second source operands and interleaves the words as they are copied to the destination. The high-order words of each octword of the source operands are ignored.

Words are interleaved in ascending order from the least-significant word of the source operands with words from the first source operand occupying the lower word of each pair copied to the destination.

For the 128-bit form of the instruction, the following operations are performed:

```
dest[15:0] = src1[15:0]
dest[31:16] = src2[15:0]
dest[47:32] = src1[31:16]
dest[63:48] = src2[31:16]
dest[79:64] = src1[47:32]
dest[95:80] = src2[47:32]
dest[111:96] = src1[63:48]
dest[127:112] = src2[63:48]
```

Additionally, for the 256-bit form of the instruction, the following operations are performed:

```
dest[143:128] = src1[143:128]
dest[159:144] = src2[143:128]
dest[175:160] = src1[159:144]
dest[191:176] = src2[159:144]
dest[207:192] = src1[175:160]
dest[223:208] = src2[175:160]
dest[239:224] = src1[191:176]
dest[255:240] = src2[191:176]
```

When the second source operand is all 0s, the destination effectively receives the 4 low-order words from the first source operand or the 4 low-order words of each octword of the first source operand zero-extended to 32 bits. This operation is useful for expanding unsigned 16-bit values to unsigned 32-bit operands for subsequent processing that requires higher precision.

There are legacy and extended forms of the instruction:

PUNPCKLWD

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source operand is also the destination register. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

PUNPCKLWD

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PUNPCKLWD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPUNPCKLWD 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPUNPCKLWD 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PUNPCKLWD *xmm1*, *xmm2/mem128*	66 0F 61 /r	Unpacks and interleaves the low-order words of *xmm1* and *xmm2* or *mem128*. Writes the words to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPUNPCKLWD *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	61 /r
VPUNPCKLWD *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	61 /r

Related Instructions

(V)PUNPCKHBW, (V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKHWD, (V)PUNPCKLBW, (V)PUNPCKLDQ, (V)PUNPCKLQDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

PXOR Packed Exclusive OR
VPXOR

Performs a bitwise XOR of the first and second source operands and writes the result to the destination. When either of a pair of corresponding bits in the first and second operands are set, the corresponding bit of the destination is set; when both source bits are set or when both source bits are not set, the destination bit is cleared.

There are legacy and extended forms of the instruction:

PXOR

The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source XMM register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VPXOR

The extended form of the instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PXOR	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VPXOR 128-bit	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VPXOR 256-bit	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
PXOR *xmm1*, *xmm2*/*mem128*	66 0F EF /r	Performs bitwise XOR of values in *xmm1* and *xmm2* or *mem128*. Writes the result to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPXOR *xmm1*, *xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	EF /r
VPXOR *ymm1*, *ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	EF /r

Related Instructions

(V)PAND, (V)PANDN, (V)POR

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

RCPPS Reciprocal
VRCPPS Packed Single-Precision Floating-Point

Computes the approximate reciprocal of each packed single-precision floating-point value in the source operand and writes the results to the corresponding doubleword of the destination. MXCSR.RC as no effect on the result.

The maximum error is less than or equal to $1.5 * 2^{-12}$ times the true reciprocal. A source value that is ±zero or denormal returns an infinity of the source value sign. Results that underflow are changed to signed zero. For both SNaN and QNaN source operands, a QNaN is returned.

There are legacy and extended forms of the instruction:

RCPPS

Computes four reciprocals. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VRCPPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Computes four reciprocals. The source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Computes eight reciprocals. The source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
RCPPS	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VRCPPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
RCPPS *xmm1*, *xmm2/mem128*	0F 53 /r	Computes reciprocals of packed single-precision floating-point values in *xmm1* or *mem128*. Writes result to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VRCPPS *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.01	X.1111.0.00	53 /r
VRCPPS *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.01	X.1111.1.00	53 /r

Related Instructions

(V)RCPSS, (V)RSQRTPS, (V)RSQRTSS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

RCPSS Reciprocal
VRCPSS Scalar Single-Precision Floating-Point

Computes the approximate reciprocal of the scalar single-precision floating-point value in a source operand and writes the results to the low-order doubleword of the destination. MXCSR.RC as no effect on the result.

The maximum error is less than or equal to $1.5 * 2^{-12}$ times the true reciprocal. A source value that is ±zero or denormal returns an infinity of the source value sign. Results that underflow are changed to signed zero. For both SNaN and QNaN source operands, a QNaN is returned.

There are legacy and extended forms of the instruction:

RCPSS

The source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [127:32] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VRCPSS

The extended form of the instruction has a 128-bit encoding only.

The first source operand and the destination are XMM registers. The second source operand is either an XMM register or a 32-bit memory location. Bits [31:0] of the destination contain the reciprocal; bits [127:32] of the destination are copied from the first source register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
RCPSS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VRCPSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
RCPSS *xmm1*, *xmm2/mem32*	F3 0F 53 /r	Computes reciprocal of scalar single-precision floating-point value in *xmm1* or *mem32*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VRCPSS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.X.10	53 /r

Related Instructions

(V)RCPPS, (V)RSQRTPS, (V)RSQRTSS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

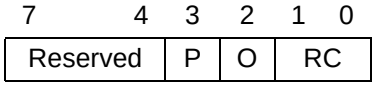
X — AVX and SSE exception
A — AVX exception
S — SSE exception

ROUNDPD Round
VROUNDPD Packed Double-Precision Floating-Point

Rounds two or four double-precision floating-point values as specified by an immediate byte operand. Source values are rounded to integral values and written to the destination as double-precision floating-point values.

SNaN source values are converted to QNaN. When DAZ =1, denormals are converted to zero before rounding.

The immediate byte operand is defined as follows.



Bits	Mnemonic	Description
[7:4]	—	Reserved
[3]	P	Precision Exception
[2]	O	Rounding Control Source
[1:0]	RC	Rounding Control

Precision exception definitions:

Value	Description
0	Normal PE exception
1	PE field is not updated. No precision exception is taken when unmasked.

Rounding control source definitions:

Value	Description
0	Use RC from immediate operand
1	Use RC from MXCSR

Rounding control definition:

Value	Description
00	Nearest
01	Downward (toward negative infinity)
10	Upward (toward positive infinity)
11	Truncated

There are legacy and extended forms of the instruction:

ROUNDPD

Rounds two source values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. There is a third 8-bit immediate operand. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VROUNDPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Rounds two source values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. There is a third 8-bit immediate operand. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Rounds four source values. The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. There is a third 8-bit immediate operand. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
PCMPEQQ	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VPCMPEQQ	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ROUNDPD *xmm1*, *xmm2*/*mem128, imm8*	66 0F 3A 09 /r ib	Rounds double-precision floating-point values in *xmm2 or mem128.* Writes rounded double-precision values to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VROUNDPD *xmm1*, *xmm2*/*mem128, imm8*	C4	$\overline{\text{RXB}}$.03	X.1111.0.01	09 /r ib
VROUNDPD *ymm1*, *xmm2*/*mem256, imm8*	C4	$\overline{\text{RXB}}$.03	X.1111.1.01	09 /r ib

Related Instructions

(V)ROUNDPS, (V)ROUNDSD, (V)ROUNDSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*

Exceptions

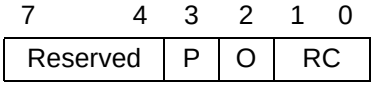
Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

ROUNDPS Round
VROUNDPS Packed Single-Precision Floating-Point

Rounds four or eight single-precision floating-point values as specified by an immediate byte operand. Source values are rounded to integral values and written to the destination as single-precision floating-point values.

SNaN source values are converted to QNaN. When DAZ =1, denormals are converted to zero before rounding.

The immediate byte operand is defined as follows.



Bits	Mnemonic	Description
[7:4]	—	Reserved
[3]	P	Precision Exception
[2]	O	Rounding Control Source
[1:0]	RC	Rounding Control

Precision exception definitions:

Value	Description
0	Normal PE exception
1	PE field is not updated. No precision exception is taken when unmasked.

Rounding control source definitions:

Value	Description
0	Use RC from immediate operand
1	Use RC from MXCSR

Rounding control definition:

Value	Description
00	Nearest
01	Downward (toward negative infinity)
10	Upward (toward positive infinity)
11	Truncated

There are legacy and extended forms of the instruction:

ROUNDPS

Rounds four source values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. There is a third 8-bit immediate operand. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VROUNDPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Rounds four source values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. There is a third 8-bit immediate operand. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Rounds eight source values. The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. There is a third 8-bit immediate operand. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
ROUNDPS	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VROUNDPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ROUNDPS *xmm1*, *xmm2*/*mem128, imm8*	66 0F 3A 08 /r ib	Rounds single-precision floating-point values in *xmm2 or mem128.* Writes rounded single-precision values to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VROUNDPS *xmm1*, *xmm2*/*mem128, imm8*	C4	$\overline{\text{RXB}}$.03	X.1111.0.01	08 /r ib
VROUNDPS *ymm1*, *xmm2*/*mem256, imm8*	C4	$\overline{\text{RXB}}$.03	X.1111.1.01	08 /r ib

Related Instructions

(V)ROUNDPD, (V)ROUNDSD, (V)ROUNDSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

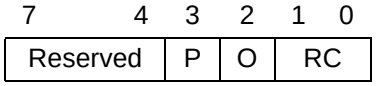
X — AVX and SSE exception
A — AVX exception
S — SSE exception

ROUNDSD Round
VROUNDSD Scalar Double-Precision

Rounds a scalar double-precision floating-point value as specified by an immediate byte operand. Source values are rounded to integral values and written to the destination as double-precision floating-point values.

SNaN source values are converted to QNaN. When DAZ =1, denormals are converted to zero before rounding.

The immediate byte operand is defined as follows.



Bits	Mnemonic	Description
[7:4]	—	Reserved
[3]	P	Precision Exception
[2]	O	Rounding Control Source
[1:0]	RC	Rounding Control

Precision exception definitions:

Value	Description
0	Normal PE exception
1	PE field is not updated. No precision exception is taken when unmasked.

Rounding control source definitions:

Value	Description
0	Use RC from immediate operand
1	Use RC from MXCSR

Rounding control definition:

Value	Description
00	Nearest
01	Downward (toward negative infinity)
10	Upward (toward positive infinity)
11	Truncated

There are legacy and extended forms of the instruction:

ROUNDSD

The source operand is either an XMM register or a 64-bit memory location. When the source is an XMM register, the value to be rounded must be in the low quadword. The destination is an XMM register. There is a third 8-bit immediate operand. Bits [127:64] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to destination XMM register are not affected.

VROUNDSD

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register. The second source operand is either an XMM register or a 64-bit memory location. The destination is a third XMM register. There is a fourth 8-bit immediate operand. Bits [127:64] of the destination are copied from the first source operand. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
ROUNDSD	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VROUNDSD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ROUNDSD *xmm1*, *xmm2*/*mem64, imm8*	66 0F 3A 0B /r ib	Rounds a double-precision floating-point value in *xmm2[63:0]* or *mem64.* Writes a rounded double-precision value to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VROUNDSD *xmm1*, *xmm2*, *xmm3*/*mem64, imm8*	C4	$\overline{\text{RXB}}$.03	X.$\overline{\textit{src1}}$.X.01	0B /r ib

Related Instructions

(V)ROUNDPD, (V)ROUNDPS, (V)ROUNDSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					*M*
17	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*

Exceptions

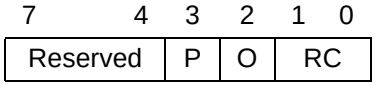
Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

ROUNDSS Round
VROUNDSS Scalar Single-Precision

Rounds a scalar single-precision floating-point value as specified by an immediate byte operand. Source values are rounded to integral values and written to the destination as single-precision floating-point values.

SNaN source values are converted to QNaN. When DAZ =1, denormals are converted to zero before rounding.

The immediate byte operand is defined as follows.



Bits	Mnemonic	Description
[7:4]	—	Reserved
[3]	P	Precision Exception
[2]	O	Rounding Control Source
[1:0]	RC	Rounding Control

Precision exception definitions:

Value	Description
0	Normal PE exception
1	PE field is not updated. No precision exception is taken when unmasked.

Rounding control source definitions:

Value	Description
0	Use RC from immediate operand
1	Use RC from MXCSR

Rounding control definition:

Value	Description
00	Nearest
01	Downward (toward negative infinity)
10	Upward (toward positive infinity)
11	Truncated

There are legacy and extended forms of the instruction:

ROUNDSS

The source operand is either an XMM register or a 32-bit memory location. When the source is an XMM register, the value to be rounded must be in the low doubleword. The destination is an XMM register. There is a third 8-bit immediate operand. Bits [127:32] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to destination XMM register are not affected.

VROUNDSS

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is a third XMM register. There is a fourth 8-bit immediate operand. Bits [127:32] of the destination are copied from the first source operand. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
ROUNDSS	SSE4.1	CPUID Fn0000_0001_ECX[SSE41] (bit 19)
VROUNDSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
ROUNDSS *xmm1*, *xmm2*/*mem64, imm8*	66 0F 3A 0A /r ib	Rounds a single-precision floating-point value in *xmm2[63:0]* or *mem64.* Writes a rounded single-precision value to *xmm1.*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VROUNDSS *xmm1*, *xmm2*, *xmm3*/*mem64, imm8*	C4	$\overline{\text{RXB}}$.03	X.$\overline{\textit{src1}}$.X.01	0A /r ib

Related Instructions

(V)ROUNDPD, (V)ROUNDPS, (V)ROUNDSD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					*M*
17	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

RSQRTPS Reciprocal Square Root
VRSQRTPS Packed Single-Precision Floating-Point

Computes the approximate reciprocal of the square root of each packed single-precision floating-point value in the source operand and writes the results to the corresponding doublewords of the destination. MXCSR.RC has no effect on the result.

The maximum error is less than or equal to $1.5 * 2^{-12}$ times the true reciprocal square root. A source value that is ±zero or denormal returns an infinity of the source value sign. Negative source values other than –zero and –denormal return a QNaN floating-point indefinite value. For both SNaN and QNaN source operands, a QNaN is returned.

There are legacy and extended forms of the instruction:

RSQRTPS

Computes four values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VRSQRTPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Computes four values. The destination is an XMM register. The source operand is either an XMM register or a 128-bit memory location. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Computes eight values. The destination is a YMM register. The source operand is either a YMM register or a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
RSQRTPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VRSQRTPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
RSQRTPS *xmm1*, *xmm2*/*mem128*	0F 52 /r	Computes reciprocals of square roots of packed single-precision floating-point values in *xmm1* or *mem128*. Writes result to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VRSQRTPS *xmm1*, *xmm2*/*mem128*	C4	$\overline{\text{RXB}}$.01	X.1111.0.00	52 /r
VRSQRTPS *ymm1*, *ymm2*/*mem256*	C4	$\overline{\text{RXB}}$.01	X.1111.1.00	52 /r

Related Instructions

(V)RSQRTSS, (V)SQRTPD, (V)SQRTPS, (V)SQRTSD, (V)SQRTSS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

RSQRTSS Reciprocal Square Root
VRSQRTSS Scalar Single-Precision Floating-Point

Computes the approximate reciprocal of the square root of the scalar single-precision floating-point value in a source operand and writes the result to the low-order doubleword of the destination. MXCSR.RC as no effect on the result.

The maximum error is less than or equal to $1.5 * 2^{-12}$ times the true reciprocal square root. A source value that is ±zero or denormal returns an infinity of the source value's sign. Negative source values other than –zero and –denormal return a QNaN floating-point indefinite value. For both SNaN and QNaN source operands, a QNaN is returned.

There are legacy and extended forms of the instruction:

RSQRTSS

The source operand is either an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [127:32] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VRSQRTSS

The extended form of the instruction has a 128-bit encoding only.

The first source operand and the destination are XMM registers. The second source operand is either an XMM register or a 32-bit memory location. Bits [31:0] of the destination contain the reciprocal square root of the single-precision floating-point value held in bits [31:0] of the second source operand; bits [127:32] of the destination are copied from the first source register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
RSQRTSS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VRSQRTSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
RSQRTSS *xmm1*, *xmm2/mem32*	F3 0F 52 /r	Computes reciprocal of square root of a scalar single-precision floating-point value in *xmm1* or *mem32*. Writes result to *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VRSQRTSS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.X.10	52 /r

Related Instructions

(V)RSQRTPS, (V)SQRTPD, (V)SQRTPS, (V)SQRTSD, (V)SQRTSS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

SHA1RNDS4 Four Rounds of SHA1

Execute 4 rounds of a SHA1 operation using the 4 double words (A, B, C, D) from the first source operand, and value E from the second operand. The lower two bits of the immediate are used to specify the function and constant appropriate for the current round of processing. The resulting (A, B, C, D) is placed in the destination register which is the same as the first source register.

The following function is performed:

```
A ← SRC1[127:96];
B ← SRC1[95:64];
C ← SRC1[63:32];
D ← SRC1[31:0];

W0E ← SRC2[127:96];
W1  ← SRC2[95:64];
W2  ← SRC2[63:32];
W3  ← SRC2[31:0];

i=imm[1:0]  which determines f_i and K_i

First Round operation:
A_1 ← f_ 0(B, C, D) + (A Rotate Left 5) +W0E +K_0;
B_1 ← A;
C_1 ← B Rotate Left 30;
D_1 ← C;
E_1 ← D;

FOR j = 1 to 3
{       A_(j +1) ← f_j(B_j, C_j, D_j) + (A_j Rotate Left 5) +Wj+ E_j +K_i;

        B_(j+1) <- A_j;

        C_(j +1) ← B_j Rotate Left 30;

        D_(j +1) ← C_j;
        E_(j +1) ← D_j;

}

DEST[127:96] ← A_4;
DEST[95:64]  ← B_4;
DEST[63:32]  ← C_4;
DEST[31:0]   ← D_4;
```

Mnemonic	Opcode	Description
SHA1RNDS4 xmm1, xmm2/m128, imm8	0F 3A CC /r ib	Executes 4 Rounds of SHA1

Related Instructions

SHA1NEXTE, SHA1MSG1, SHA1MSG2

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exceptions	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID
	A	A		AVX instructions are only recognized in protected mode
	S	S	S	CR0.EM=1 OR CR4.OSFXSR=0
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE]
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page Fault, #PF		S	X	A page fault resulted from the execution of the instruction

X - SSE, AVX, and AVX2 exception
A - AVX, AVX2 exception
S - SSE exception

SHA1NEXTE Calculate Next E SHA1

Calculate what the next E register values should be after 4 rounds of a SHA1 operation using the 4 double words from the second source operand, and value A from the first operand. The resulting E is placed in the destination register which is the same as the first source register.

```
DEST[127:96] ← SRC2[127:96] + (SRC1[127:96] rotated left 30)
DEST[95:0]   ← SRC2[95:0];
```

Mnemonic	Opcode	Description
SHA1NEXTE xmm1,xmm2/m128	0F 38 C8 /r	Calculate Next E of SHA1

Related Instructions

SHA1RNDS4, SHA1MSG1, SHA1MSG2

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exceptions	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID
	A	A		AVX instructions are only recognized in protected mode
	S	S	S	CR0.EM=1 OR CR4.OSFXSR=0
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE]
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory

Exceptions	Real	Virtual 8086	Protected	Cause of Exception
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page Fault, #PF		S	X	A page fault resulted from the execution of the instruction
X - SSE, AVX, and AVX2 exception A - AVX, AVX2 exception S - SSE exception				

SHA1MSG1 Message Intermediate 1

Performs the 1st of two intermediate calculations necessary before doing the next four rounds of the SHA1 message.

```
DEST[127:96] ← SRC1[63:32]  XOR SRC1[127:96]
DEST[95:64]  ← SRC1[31:0]   XOR SRC1[95:64]
DEST[63:32]  ← SRC2[127:96] XOR SRC1[63:32]
DEST[31:0]   ← SRC2[95:64]  XOR SRC1[31:0]
```

Mnemonic	Opcode	Description
SHA1MSG1 xmm1, xmm2/m128	0F 38 C9 /r	Calculate Message Intermediate 1

Related Instructions

SHA1RNDS4, SHA1NEXTE, SHA1MSG2

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exceptions	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID
	A	A		AVX instructions are only recognized in protected mode
	S	S	S	CR0.EM=1 OR CR4.OSFXSR=0
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE]
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory

Exceptions	Real	Virtual 8086	Protected	Cause of Exception
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page Fault, #PF		S	X	A page fault resulted from the execution of the instruction
X - SSE, AVX, and AVX2 exception A - AVX, AVX2 exception S - SSE exception				

SHA1MSG2 Message Calculation 2

Performs the 2nd of two intermediate calculations necessary before doing the next four rounds of the SHA1 message.

```
Temp[31:0] ← (SRC1[127:96]   XOR  SRC2[95:64]) Rotate Left 1

DEST[127:96] ← Temp[31:0]
DEST[95:64]  ← (SRC1[95:64]  XOR  SRC2[63:32]) Rotate Left 1
DEST[63:32]  ← (SRC1{63:32]  XOR  SRC2[31:0])  Rotate Left 1
DEST[31:0]   ← (SRC1[31:0]   XOR  Temp[31:0])  Rotate Left 1
```

Mnemonic	Opcode	Description
SHA1MSG2 xmm1, xmm2/m128	0F 38 CA /r	CCalculate Message Intermediate 2

Related Instructions

SHA1RNDS4, SHA1NEXTE, SHA1MSG1

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exceptions	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID
	A	A		AVX instructions are only recognized in protected mode
	S	S	S	CR0.EM=1 OR CR4.OSFXSR=0
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE]
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

Exceptions	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page Fault, #PF		S	X	A page fault resulted from the execution of the instruction
X - SSE, AVX, and AVX2 exception A - AVX, AVX2 exception S - SSE exception				

SHA256RNDS2 Two Rounds of SHA256

Performs 2 rounds of SHA256 operation with the first operand holding the initial SHA256 state (C, D, G, H), the second operand holding the initial SHA256 state (A, B, E, F), and the implicit operand xmm0 holding a pre-computed sum of the next two double word round 2 message as well as the corresponding round constants. The resulting SHA256 state (A, B, E, F) is placed in the destination register.

```
A_0 ← SRC2[127:96];
B_0 ← SRC2[95:64];
C_0 ← SRC1[127:96];
D_0 ← SRC1[95:64];
E_0 ← SRC2[63:32];
F_0 ← SRC2[31:0];
G_0 ← SRC1[63:32];
H_0 ← SRC1[31:0];
K0 ← XMM0[31: 0];
K1 ← XMM0[63: 32];

FOR i = 0 to 1
{       A_(i +1) ← Ch (E_i, F_i, G_i) + Perm1(E_i) +K_i + H_i + Ma(A_i , B_i, C_i) + Perm0(A_i);
        B_(i +1) ← A_i;
        C_(i +1) ← B_i ;
        D_(i +1) ← C_i;
        E_(i +1) ← Ch (E_i, F_i, G_i) + Perm1(E_i) + K_i + H_i + D_i;
        F_(i +1) ← E_i ;
        G_(i +1) ← F_i;
        H_(i +1) ← G_i;
}

DEST[127:96] ← A_2;
DEST[95:64] ← B_2;
DEST[63:32] ← E_2;
DEST[31:0] ← F_2;
```

Mnemonic	Opcode	Description
SHA256RNDS2xmm1, xmm2/m128, xmm0	0F 38 CB /r	Execute 2 rounds of SHA256

Related Instructions

SHA256MSG1, SHA256MSG2

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exceptions	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID
	A	A		AVX instructions are only recognized in protected mode
	S	S	S	CR0.EM=1 OR CR4.OSFXSR=0
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE]
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page Fault, #PF		S	X	A page fault resulted from the execution of the instruction

X - SSE, AVX, and AVX2 exception
A - AVX, AVX2 exception
S - SSE exception

SHA256MSG1 Message Intermediate 1

Performs the 1st of two intermediate calculations necessary for the next four SHA256 message dwords.

```
DEST[127:96] ← SRC1[127:96] + Perm2( SRC2[31:0])
DEST[95:64]  ← SRC1[95:64]  + Perm2( SRC1[127:96])
DEST[63:32]  ← SRC1[63:32]  + Perm2( SRC1[95:64]
DEST[31:0]   ← SRC1[31:0]   + Perm2( SRC1[63:62])
```

Mnemonic	Opcode	Description
SHA256MSG1xmm1, xmm2/m128	0F 38 CC /r	Calculate Message Intermediate 1

Related Instructions

SHA256RNDS2, SHA256MSG2

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exceptions	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID
	A	A		AVX instructions are only recognized in protected mode
	S	S	S	CR0.EM=1 OR CR4.OSFXSR=0
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE]
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.

Exceptions	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page Fault, #PF		S	X	A page fault resulted from the execution of the instruction
X - SSE, AVX, and AVX2 exception A - AVX, AVX2 exception S - SSE exception				

SHA256MSG2 Message Intermediate 2

Performs the 2nd of two intermediate calculations necessary for the next four SHA256 message dwords.

```
Temp0        ← SRC1[31:0]     + Perm3( SRC2[95:64])
Temp1        ← SRC1[63:32]    + Perm3( SRC2[127:96])

DEST[127:96] ← SRC1[127:96]   + Perm3( Temp1)
DEST[95:64]  ← SRC1[95:64]    + Perm3( Temp0)
DEST[63:32]  ← SRC1[63:32]    + Perm3( SRC2[127:96])
DEST[31:0]   ← SRC1[31:0]     + Perm3( SRC2[95:624])
```

Mnemonic	Opcode	Description
SHA256MSG1 xmm1, xmm2/m128	0F 38 CD /r	Calculate Message Intermediate 2

Related Instructions

SHA256RNDS2, SHA256MSG1

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exceptions	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	Instruction not supported by CPUID
	A	A		AVX instructions are only recognized in protected mode
	S	S	S	CR0.EM=1 OR CR4.OSFXSR=0
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE]
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.

Exceptions	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page Fault, #PF		S	X	A page fault resulted from the execution of the instruction
X - SSE, AVX, and AVX2 exception A - AVX, AVX2 exception S - SSE exception				

SHUFPD Shuffle
VSHUFPD Packed Double-Precision Floating-Point

Copies packed double-precision floating-point values from either of two sources to quadwords in the destination, as specified by bit fields of an immediate byte operand.

Each bit corresponds to a quadword destination. The 128-bit legacy and extended versions of the instruction use bits [1:0]; the 256-bit extended version uses bits [3:0], as shown.

Destination Quadword	Immediate-Byte Bit Field	Value of Bit Field	Source 1 Bits Copied	Source 2 Bits Copied
Used by 128-bit encoding and 256-bit encoding				
[63:0]	[0]	0	[63:0]	—
		1	[127:64]	—
[127:64]	[1]	0	—	[63:0]
		1	—	]127:64]
Used only by 256-bit encoding				
[191:128]	[2]	0	[191:128]	—
		1	[255:192]	—
[255:192]	[3]	0	—	[191:128]
		1	—	[255:192]

There are legacy and extended forms of the instruction:

SHUFPD

Selects from four source values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. There is a third 8-bit immediate operand. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VSHUFPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Selects from four source values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. There is a fourth 8-bit immediate operand. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Selects from eight source values. The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register. There is a fourth 8-bit immediate operand.

Instruction Support

Form	Subset	Feature Flag
SHUFPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VSHUFPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
SHUFPD *xmm1*, *xmm2*/*mem128*, *imm8*	66 0F C6 /r ib	Shuffles packed double-precision floating-point values in *xmm1* and *xmm2* or *mem128*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VSHUFPD *xmm1*, *xmm2, xmm3*/*mem128*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	C6 /r
VSHUFPD *ymm1*, *ymm2, ymm3*/*mem256*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	C6 /r

Related Instructions

(V)SHUFPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

SHUFPS Shuffle
VSHUFPS Packed Single-Precision Floating-Point

Copies packed single-precision floating-point values from either of two sources to doublewords in the destination, as specified by bit fields of an immediate byte operand.

Each bit field corresponds to a doubleword destination. The 128-bit legacy and extended versions of the instruction use a single 128-bit destination; the 256-bit extended version performs duplicate operations on bits [127:0] and bits [255:128] of the source and destination.

Destination Doubleword	Immediate-Byte Bit Field	Value of Bit Field	Source 1 Bits Copied	Source 2 Bits Copied
[31:0]	[1:0]	00	[31:0]	—
		01	[63:32]	—
		10	[95:64]	—
		11	[127:96]	—
[63:32]	[3:2]	00	[31:0]	—
		01	[63:32]	—
		10	[95:64]	—
		11	[127:96]	—
[95:64]	[5:4]	00	—	[31:0]
		01	—	[63:32]
		10	—	[95:64]
		11	—	[127:96]
[127:96]	[7:6]	00	—	[31:0]
		01	—	[63:32]
		10	—	[95:64]
		11	—	[127:96]
Upper 128 bits of 256-bit source and destination used by 256-bit encoding				
[159:128]	[1:0]	00	[159:128]	—
		01	[191:160]	—
		10	[223:192]	—
		11	[255:224]	—
[191:160]	[3:2]	00	[159:128]	—
		01	[191:160]	—
		10	[223:192]	—
		11	[255:224]	—
[223:192]	[5:4]	00	—	[159:128]
		01	—	[191:160]
		10	—	[223:192]
		11	—	[255:224]
[255:224]	[7:6]	00	—	[159:128]
		01	—	[191:160]
		10	—	[223:192]
		11	—	[255:224]

There are legacy and extended forms of the instruction:

SHUFPS

Selects from eight source values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. There is a third 8-bit immediate operand. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VSHUFPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Selects from eight source values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. There is a fourth 8-bit immediate operand. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Selects from 16 source values. The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register. There is a fourth 8-bit immediate operand.

Instruction Support

Form	Subset	Feature Flag
SHUFPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VSHUFPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
SHUFPS *xmm1*, *xmm2*/*mem128*, *imm8*	0F C6 /r ib	Shuffles packed single-precision floating-point values in *xmm1* and *xmm2* or *mem128*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VSHUFPS *xmm1*, *xmm2, xmm3*/*mem128*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.00	C6 /r
VSHUFPS *ymm1*, *ymm2, ymm3*/*mem256*, *imm8*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.00	C6 /r

Related Instructions

(V)SHUFPD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

SQRTPD Square Root
VSQRTPD Packed Double-Precision Floating-Point

Computes the square root of each packed double-precision floating-point value in a source operand and writes the result to the corresponding quadword of the destination.

Performing the square root of +infinity returns +infinity.

There are legacy and extended forms of the instruction:

SQRTPD

Computes two values. The destination is an XMM register. The source operand is either an XMM register or a 128-bit memory location. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VSQRTPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Computes two values. The source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Computes four values. The source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
SQRTPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VSQRTPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
SQRTPD *xmm1*, *xmm2/mem128*	66 0F 51 /r	Computes square roots of packed double-precision floating-point values in *xmm1* or *mem128*. Writes the results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VSQRTPD *xmm1, xmm2/mem128*	C4	$\overline{\text{RXB}}$.01	X.1111.0.01	51 /r
VSQRTPD *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.01	X.1111.1.01	51 /r

Related Instructions

(V)RSQRTPS, (V)RSQRTSS, (V)SQRTPS, (V)SQRTSD, (V)SQRTSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M				M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

SQRTPS VSQRTPS — Square Root Packed Single-Precision Floating-Point

Computes the square root of each packed single-precision floating-point value in a source operand and writes the result to the corresponding doubleword of the destination.

Performing the square root of +infinity returns +infinity.

There are legacy and extended forms of the instruction:

SQRTPS

Computes four values. The destination is an XMM register. The source operand is either an XMM register or a 128-bit memory location. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VSQRTPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Computes four values. The destination is an XMM register. The source operand is either an XMM register or a 128-bit memory location. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Computes eight values. The destination is a YMM register. The source operand is either a YMM register or a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
SQRTPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VSQRTPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
SQRTPS *xmm1*, *xmm2/mem128*	0F 51 /r	Computes square roots of packed single-precision floating-point values in *xmm1* or *mem128*. Writes the results to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VSQRTPS *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.01	X.1111.0.00	51 /r
VSQRTPS *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.01	X.1111.1.00	51 /r

Related Instructions

(V)RSQRTPS, (V)RSQRTSS, (V)SQRTPD, (V)SQRTSD, (V)SQRTSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M				M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

SQRTSD Square Root
VSQRTSD Scalar Double-Precision Floating-Point

Computes the square root of a double-precision floating-point value and writes the result to the low quadword of the destination. The three-operand form of the instruction also writes a copy of the upper quadword of a second source operand to the upper quadword of the destination.

Performing the square root of +infinity returns +infinity.

There are legacy and extended forms of the instruction:

SQRTSD

The source operand is either an XMM register or a 64-bit memory location. When the source is an XMM register, the source value must be in the low quadword. The destination is an XMM register. Bits [127:64] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to destination XMM register are not affected.

VSQRTSD

The extended form of the instruction has a single 128-bit encoding that requires three operands:

VSQRTSD *xmm1, xmm2, xmm3/mem64*

The first source operand is an XMM register. The second source operand is either an XMM register or a 64-bit memory location. When the second source is an XMM register, the source value must be in the low quadword. The destination is a third XMM register. The square root of the second source operand is written to bits [63:0] of the destination register. Bits [127:64] of the destination are copied from the corresponding bits of the first source operand. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
SQRTSD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VSQRTSD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
SQRTSD *xmm1*, *xmm2/mem64*	F2 0F 51 /r	Computes the square root of a double-precision floating-point value in *xmm1* or *mem64*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VSQRTSD *xmm1*, *xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.X.11	51 /r

Related Instructions

(V)RSQRTPS, (V)RSQRTSS, (V)SQRTPD, (V)SQRTPS, (V)SQRTSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M				M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

SQRTSS Square Root
VSQRTSS Scalar Single-Precision Floating-Point

Computes the square root of a single-precision floating-point value and writes the result to the low doubleword of the destination. The three-operand form of the instruction also writes a copy of the three most significant doublewords of a second source operand to the upper 96 bits of the destination.

Performing the square root of +infinity returns +infinity.

There are legacy and extended forms of the instruction:

SQRTSS

The source operand is either an XMM register or a 32-bit memory location. When the source is an XMM register, the source value must be in the low doubleword. The destination is an XMM register. Bits [127:32] of the destination are not affected. Bits [255:128] of the YMM register that corresponds to destination XMM register are not affected.

VSQRTSS

The extended form has a single 128-bit encoding that requires three operands:

VSQRTSS *xmm1, xmm2, xmm3/mem64*

The first source operand is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. When the second source is an XMM register, the source value must be in the low doubleword. The destination is a third XMM register. The square root of the second source operand is written to bits [31:0] of the destination register. Bits [127:32] of the destination are copied from the corresponding bits of the first source operand. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
SQRTSS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VSQRTSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
SQRTSS *xmm1*, *xmm2/mem32*	F3 0F 51 /r	Computes square root of a single-precision floating-point value in *xmm1* or *mem32*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VSQRTSS *xmm1*, *xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.X.10	51 /r

Related Instructions

(V)RSQRTPS, (V)RSQRTSS, (V)SQRTPD, (V)SQRTPS, (V)SQRTSD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M				M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

STMXCSR Store MXCSR
VSTMXCSR

Saves the content of the MXCSR extended control/status register to a 32-bit memory location. Reserved bits are stored as zeroes. The MXCSR is described in “Registers” in Volume 1.

For both legacy STMXCSR and extended VSTMXCSR forms of the instruction, the source operand is the MXCSR and the destination is a 32-bit memory location.

There is one encoding for each instruction form.

Instruction Support

Form	Subset	Feature Flag
STMXCSR	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VSTMXCSR	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
STMXCSR *mem32*	0F AE /3	Stores content of MXCSR in *mem32*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VSTMXCSR *mem32*	C4	$\overline{\text{RXB}}$.01	X.1111.0.00	AE /3

Related Instructions

(V)LDMXCSR

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
	S	S	S	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

SUBPD VSUBPD Subtract Packed Double-Precision Floating-Point

Subtracts each packed double-precision floating-point value of the second source operand from the corresponding value of the first source operand and writes the difference to the corresponding quadword of the destination.

There are legacy and extended forms of the instruction:

SUBPD

Subtracts two pairs of values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VSUBPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Subtracts two pairs of values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Subtracts four pairs of values. The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
SUBPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VSUBPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
SUBPD *xmm1*, *xmm2/mem128*	66 0F 5C /r	Subtracts packed double-precision floating-point values in *xmm2* or *mem128* from corresponding values of *xmm1*. Writes differences to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VSUBPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	5C /r
VSUBPD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	5C /r

Related Instructions

(V)SUBPS, (V)SUBSD, (V)SUBSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

SUBPS Subtract
VSUBPS Packed Single-Precision Floating-Point

Subtracts each packed single-precision floating-point value of the second source operand from the corresponding value of the first source operand and writes the difference to the corresponding quadword of the destination.

There are legacy and extended forms of the instruction:

SUBPS

Subtracts four pairs of values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VSUBPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Subtracts four pairs of values. The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Subtracts eight pairs of values. The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
SUBPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VSUBPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
SUBPS *xmm1*, *xmm2/mem128*	0F 5C /r	Subtracts packed single-precision floating-point values in *xmm2* or *mem128* from corresponding values of *xmm1*. Writes differences to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VSUBPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.00	5C /r
VSUBPS *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.00	5C /r

Related Instructions

(V)SUBPD, (V)SUBSD, (V)SUBSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

SUBSD Subtract
VSUBSD Scalar Double-Precision Floating-Point

Subtracts the double-precision floating-point value in the low-order quadword of the second source operand from the corresponding value in the first source operand and writes the result to the low-order quadword of the destination

There are legacy and extended forms of the instruction:

SUBSD

The first source operand is an XMM register and the second source operand is either an XMM register or a 64-bit memory location. The first source register is also the destination register. Bits [127:64] of the destination and bits [255:128] of the corresponding YMM register are not affected.

VSUBSD

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 64-bit memory location. The destination is a third XMM register. Bits [127:64] of the first source operand are copied to bits [127:64] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
SUBSD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VSUBSD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
SUBSD *xmm1*, *xmm2/mem64*	F2 0F 5C /r	Subtracts low-order double-precision floating-point value in *xmm2* or *mem64* from the corresponding value of *xmm1*. Writes the difference to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VSUBSD *xmm1*, *xmm2*, *xmm3*/*mem64*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.X.11	5C /r

Related Instructions

(V)SUBPD, (V)SUBPS, (V)SUBSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

SUBSS Subtract
VSUBSS Scalar Single-Precision Floating-Point

Subtracts the single-precision floating-point value in the low-order word of the second source operand from the corresponding value in the first source operand and writes the result to the low-order word of the destination

There are legacy and extended forms of the instruction:

SUBSS

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location. The first source register is also the destination register. Bits [127:32] of the destination and bits [255:128] of the corresponding YMM register are not affected.

VSUBSS

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location. The destination is a third XMM register. Bits [127:32] of the first source operand are copied to bits [127:32] of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
SUBSS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VSUBSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
SUBSS *xmm1*, *xmm2*/*mem32*	F3 0F 5C /r	Subtracts a low-order single-precision floating-point value in *xmm2* or *mem32* from the corresponding value of *xmm1*. Writes the difference to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VSUBSS *xmm1*, *xmm2*, *xmm3*/*mem32*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.X.10	5C /r

Related Instructions

(V)SUBPD, (V)SUBPS, (V)SUBSD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

UCOMISD Unordered Compare
VUCOMISD Scalar Double-Precision Floating-Point

Performs an unordered comparison of a double-precision floating-point value in the low-order 64 bits of an XMM register with a double-precision floating-point value in the low-order 64 bits of an XMM register or a 64-bit memory location.

The ZF, PF, and CF bits in the rFLAGS register reflect the result of the compare as follows.

Result of Compare	ZF	PF	CF
Unordered	1	1	1
Greater Than	0	0	0
Less Than	0	0	1
Equal	1	0	0

The OF, AF, and SF bits in rFLAGS are cleared. If the instruction causes an unmasked SIMD floating-point exception (#XF), the rFLAGS bits are not updated.

The result is unordered when one or both of the operand values is a NaN. UCOMISD signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN.

The legacy and extended forms of the instruction operate in the same way.

Instruction Support

Form	Subset	Feature Flag
UCOMISD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VUCOMISD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
UCOMISD *xmm1*, *xmm2/mem64*	66 0F 2E /r	Compares scalar double-precision floating-point values in *xmm1* and *xmm2* or *mem64*. Sets rFLAGS.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VUCOMISD *xmm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.00001	X.1111.X.01	2E /r

Related Instructions

(V)CMPPD, (V)CMPPS, (V)CMPSD, (V)CMPSS, (V)COMISD, (V)COMISS, (V)UCOMISS

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	M	0	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set or cleared is M (modified). Unaffected flags are blank.*
Note: *If the instruction causes an unmasked SIMD floating-point exception (#XF), the rFLAGS bits are not updated.*

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

UCOMISS Unordered Compare
VUCOMISS Scalar Single-Precision Floating-Point

Performs an unordered comparison of a single-precision floating-point value in the low-order 32 bits of an XMM register with a single-precision floating-point value in the low-order 32 bits of an XMM register or a 32-bit memory location.

The ZF, PF, and CF bits in the rFLAGS register reflect the result of the compare as follows.

Result of Compare	ZF	PF	CF
Unordered	1	1	1
Greater Than	0	0	0
Less Than	0	0	1
Equal	1	0	0

The OF, AF, and SF bits in rFLAGS are cleared. If the instruction causes an unmasked SIMD floating-point exception (#XF), the rFLAGS bits are not updated.

The result is unordered when one or both of the operand values is a NaN. UCOMISD signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN.

The legacy and extended forms of the instruction operate in the same way.

Instruction Support

Form	Subset	Feature Flag
UCOMISS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VUCOMISS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
UCOMISS *xmm1*, *xmm2/mem32*	0F 2E /r	Compares scalar single-precision floating-point values in *xmm1* and *xmm2* or *mem64*. Sets rFLAGS.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VUCOMISS *xmm1*, *xmm2/mem32*	C4	$\overline{\text{RXB}}$.01	X.1111.X.00	2E /r

Related Instructions

(V)CMPPD, (V)CMPPS, (V)CMPSD, (V)CMPSS, (V)COMISD, (V)COMISS, (V)UCOMISD

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				0	M	0	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set or cleared is M (modified). Unaffected flags are blank.*
Note: *If the instruction causes an unmasked SIMD floating-point exception (#XF), the rFLAGS bits are not updated.*

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
															M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

UNPCKHPD Unpack High
VUNPCKHPD Double-Precision Floating-Point

Unpacks the high-order double-precision floating-point values of the first and second source operands and interleaves the values into the destination. Bits [63:0] of the source operands are ignored.

Values are interleaved in ascending order from the lsb of the sources and the destination. Bits [127:64] of the first source are written to bits [63:0] of the destination; bits [127:64] of the second source are written to bits [127:64] of the destination. For the 256-bit encoding, the process is repeated for bits [255:192] of the sources and bits [255:128] of the destination.

There are legacy and extended forms of the instruction:

UNPCKHPD

Interleaves one pair of values. The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VUNPCKHPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Interleaves one pair of values. The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Interleaves two pairs of values. The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
UNPCKHPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VUNPCKHPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
UNPCKHPD *xmm1*, *xmm2/mem128*	66 0F 15 /r	Unpacks the high-order double-precision floating-point values in *xmm1* and *xmm2* or *mem128* and interleaves them into *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VUNPCKHPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.01	15 /r
VUNPCKHPD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.01	15 /r

Related Instructions

(V)UNPCKHPS, (V)UNPCKLPD, (V)UNPCKLPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

UNPCKHPS Unpack High
VUNPCKHPS Single-Precision Floating-Point

Unpacks the high-order single-precision floating-point values of the first and second source operands and interleaves the values into the destination. Bits [63:0] of the source operands are ignored.

Values are interleaved in ascending order from the lsb of the sources and the destination. Bits [95:64] of the first source are written to bits [31:0] of the destination; bits [95:64] of the second source are written to bits [63:32] of the destination and so on, ending with bits [127:96] of the second source in bits [127:96] of the destination. For the 256-bit encoding, the process continues for bits [255:192] of the sources and bits [255:128] of the destination.

There are legacy and extended forms of the instruction:

UNPCKHPS

Interleaves two pairs of values. The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VUNPCKHPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Interleaves two pairs of values. The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Interleaves four pairs of values. The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
UNPCKHPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VUNPCKHPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
UNPCKHPS *xmm1*, *xmm2*/*mem128*	0F 15 /r	Unpacks the high-order single-precision floating-point values in *xmm1* and *xmm2* or *mem128* and interleaves them into *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VUNPCKHPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.00	15 /r
VUNPCKHPS *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.00	15 /r

Related Instructions

(V)UNPCKHPD, (V)UNPCKLPD, (V)UNPCKLPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

UNPCKLPD Unpack Low
VUNPCKLPD Double-Precision Floating-Point

Unpacks the low-order double-precision floating-point values of the first and second source operands and interleaves the values into the destination. Bits [127:64] of the source operands are ignored.

Values are interleaved in ascending order from the lsb of the sources and the destination. Bits [63:0] of the first source are written to bits [63:0] of the destination; bits [63:0] of the second source are written to bits [127:64] of the destination. For the 256-bit encoding, the process is repeated for bits [191:128] of the sources and bits [255:128] of the destination.

There are legacy and extended forms of the instruction:

UNPCKLPD

Interleaves one pair of values. The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VUNPCKLPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Interleaves one pair of values. The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Interleaves two pairs of values. The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
UNPCKLPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VUNPCKLPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
UNPCKLPD *xmm1*, *xmm2*/*mem128*	66 0F 14 /r	Unpacks the low-order double-precision floating-point values in *xmm1* and *xmm2* or *mem128* and interleaves them into *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VUNPCKLPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	14 /r
VUNPCKLPD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	14 /r

Related Instructions

(V)UNPCKHPD, (V)UNPCKHPS, (V)UNPCKLPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

UNPCKLPS Unpack Low
VUNPCKLPS Single-Precision Floating-Point

Unpacks the low-order single-precision floating-point values of the first and second source operands and interleaves the values into the destination. Bits [127:64] of the source operands are ignored.

Values are interleaved in ascending order from the lsb of the sources and the destination. Bits [31:0] of the first source are written to bits [31:0] of the destination; bits [31:0] of the second source are written to bits [63:32] of the destination and so on, ending with bits [63:32] of the second source in bits [127:96] of the destination. For the 256-bit encoding, the process continues for bits [191:128] of the sources and bits [255:128] of the destination.

There are legacy and extended forms of the instruction:

UNPCKLPS

Interleaves two pairs of values. The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VUNPCKLPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Interleaves two pairs of values. The first source operand is an XMM register and the second source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Interleaves four pairs of values. The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
UNPCKLPS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VUNPCKLPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
UNPCKLPS *xmm1*, *xmm2*/*mem128*	0F 14 /r	Unpacks the high-order single-precision floating-point values in *xmm1* and *xmm2* or *mem128* and interleaves them into *xmm1*

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VUNPCKLPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.0.00	14 /r
VUNPCKLPS *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{\textit{src1}}$.1.00	14 /r

Related Instructions

(V)UNPCKHPD, (V)UNPCKHPS, (V)UNPCKLPD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

VBROADCASTF128 Load With Broadcast From 128-bit Memory Location

Loads double-precision floating-point data from a 128-bit memory location and writes it to the two 128-bit elements of a YMM register

This extended-form instruction has a single 256-bit encoding.

The source operand is a 128-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VBROADCASTF128	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VBROADCASTF128 *ymm1*, *mem128*	C4	$\overline{\text{RXB}}$.02	0.1111.1.01	1A /r

Related Instructions

VBROADCASTSD, VBROADCASTSS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

VBROADCASTI128 Load With Broadcast Integer From 128-bit Memory Location

Loads data from a 128-bit memory location and writes it to the two 128-bit elements of a YMM register

There is a single form of this instruction:

VBROADCASTI128 *dest, mem128*

There is a single VEX.L = 1 encoding of this instruction.

The source operand is a 128-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VBROADCASTI128	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

Instruction Encoding

	Encoding			
Mnemonic	**VEX**	**RXB.map_select**	**W.vvvv.L.pp**	**Opcode**
VBROADCASTI128 *ymm1*, *mem128*	C4	$\overline{\text{RXB}}$.02	0.1111.1.01	5A /r

Related Instructions

VBROADCASTF128, VEXTRACTF128, VEXTRACTI128, VINSERTF128, VINSERTI128

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 0.
			A	Register-based source operand specified (MODRM.mod = 11b)
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

VBROADCASTSD Load With Broadcast Scalar Double

Loads a double-precision floating-point value from a register or memory and writes it to the four 64-bit elements of a YMM register

This extended-form instruction has a single 256-bit encoding.

The source operand is the lower half of an XMM register or a 64-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VBROADCASTSD *ymm1*, *mem64*	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VBROADCASTSD *ymm1*, *xmm*	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VBROADCASTSD *ymm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	0.1111.1.01	19 /r

Related Instructions

VBROADCASTF128, VBROADCASTSS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 0.
			A	Register-based source operand specified when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX, AVX2 exception.				

VBROADCASTSS Load With Broadcast Scalar Single

Loads a single-precision floating-point value from a register or memory and writes it to all 4 or 8 doublewords of an XMM or YMM register.

This extended-form instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Copies the source operand to all four 32-bit elements of the destination.

The source operand is the least-significant 32 bits of an XMM register or a 32-bit memory location. The destination is an XMM register.

YMM Encoding

Copies the source operand to all eight 32-bit elements of the destination.

The source operand is the least-significant 32 bits of an XMM register or a 32-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VBROADCASTSS *mem32*	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)
VBROADCASTSS *xmm*	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VBROADCASTSS *xmm1*, *xmm2/mem32*	C4	$\overline{\text{RXB}}$.02	0.1111.0.01	18 /r
VBROADCASTSS *ymm1*, *xmm2/mem32*	C4	$\overline{\text{RXB}}$.02	0.1111.1.01	18 /r

Related Instructions

VBROADCASTF128, VBROADCASTSD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	MODRM.mod = 11b when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX, AVX2 exception.				

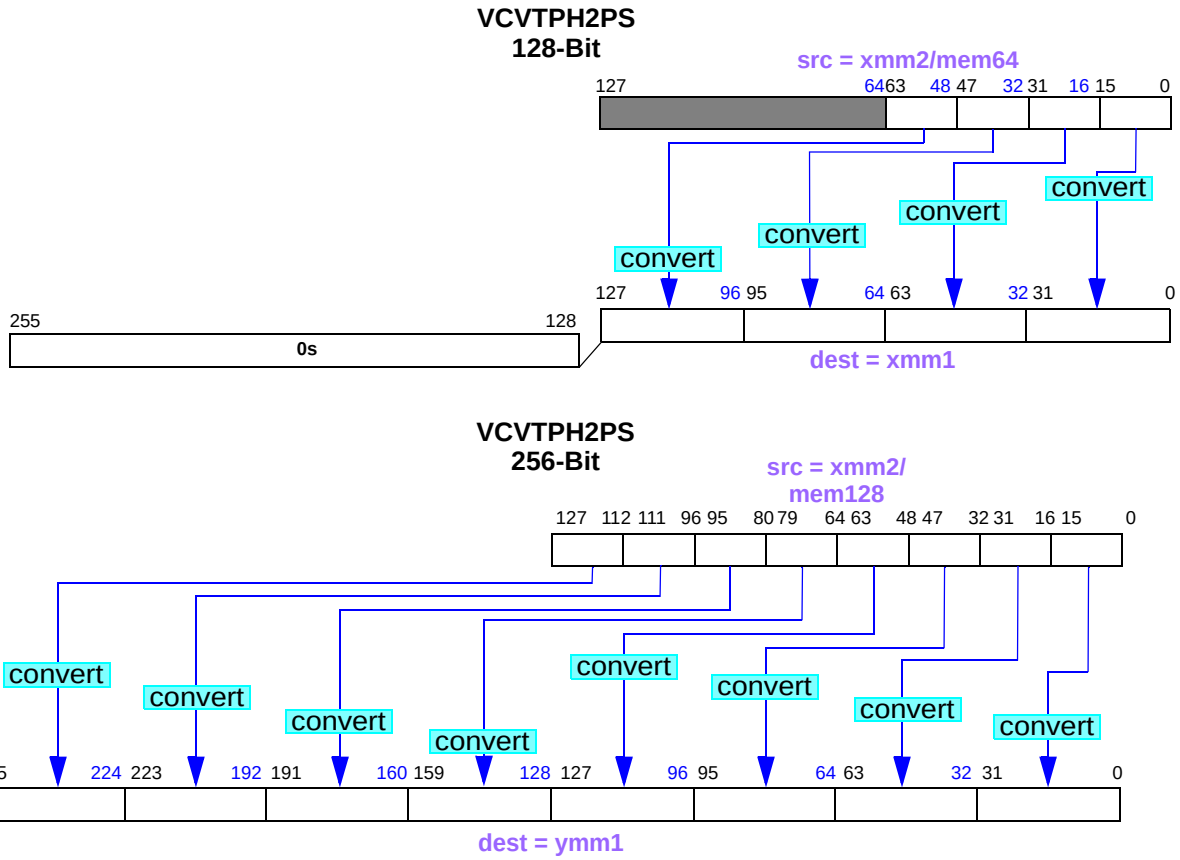
VCVTPH2PS Convert Packed 16-Bit Floating-Point to Single-Precision Floating-Point

Converts packed 16-bit floating point values to single-precision floating point values.

A denormal source operand is converted to a normal result in the destination register. MXCSR.DAZ is ignored and no MXCSR denormal exception is reported.

Because the full range of 16-bit floating-point encodings, including denormal encodings, can be represented exactly in single-precision format, rounding, inexact results, and denormalized results are not applicable.

The operation of this instruction is illustrated in the following diagram.



This extended-form instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Converts four packed 16-bit floating-point values in the low-order 64 bits of an XMM register or in a 64-bit memory location to four packed single-precision floating-point values and writes the converted values to an XMM destination register. When the result operand is written to the destination register, the upper 128 bits of the corresponding YMM register are zeroed.

YMM Encoding

Converts eight packed 16-bit floating-point values in the low-order 128 bits of a YMM register or in a 128-bit memory location to eight packed single-precision floating-point values and writes the converted values to a YMM destination register.

Instruction Support

Form	Subset	Feature Flag
VCVTPH2PS	F16C	CPUID Fn0000_0001_ECX[F16C] (bit 29)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTPH2PS *xmm1*, *xmm2*/*mem64*	C4	$\overline{\text{RXB}}$.02	0.1111.0.01	13 /r
VCVTPH2PS *ymm1*, *xmm2*/*mem128*	C4	$\overline{\text{RXB}}$.02	0.1111.1.01	13 /r

Related Instructions

VCVTPS2PH

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
																M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: A flag that may be set to one or cleared to zero is M (modified). Unaffected flags are blank.

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		AVX instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	VEX.W field = 1.
			A	VEX.vvvv ! = 1111b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Alignment check, #AC			F	Unaligned memory reference when alignment checking enabled.
Page fault, #PF			F	Instruction execution caused a page fault.
SIMD Floating-Point Exception, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid-operation exception (IE)			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized-operand exception (DE)			F	A source operand was a denormal value.
Overflow exception (OE)			F	Rounded result too large to fit into the format of the destination operand.
Underflow exception (UE)			F	Rounded result too small to fit into the format of the destination operand.
Precision exception (PE)			F	A result could not be represented exactly in the destination format.

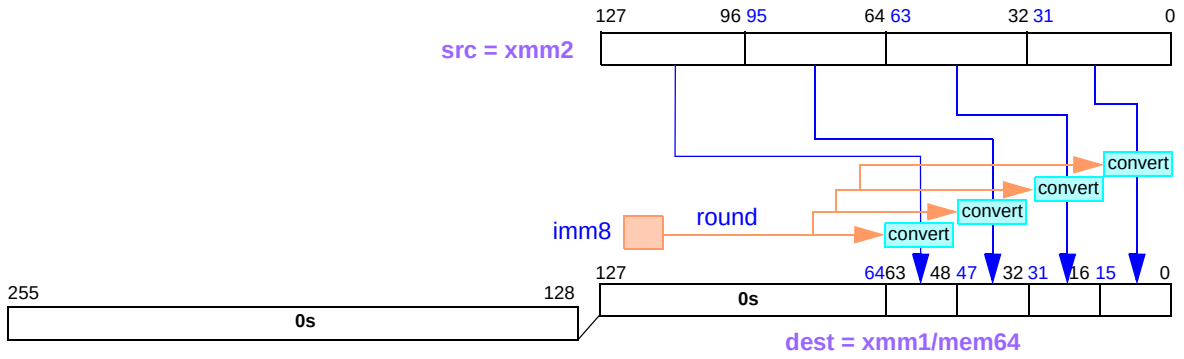
F — F16C exception.

VCVTPS2PH Convert Packed Single-Precision Floating-Point to 16-Bit Floating-Point

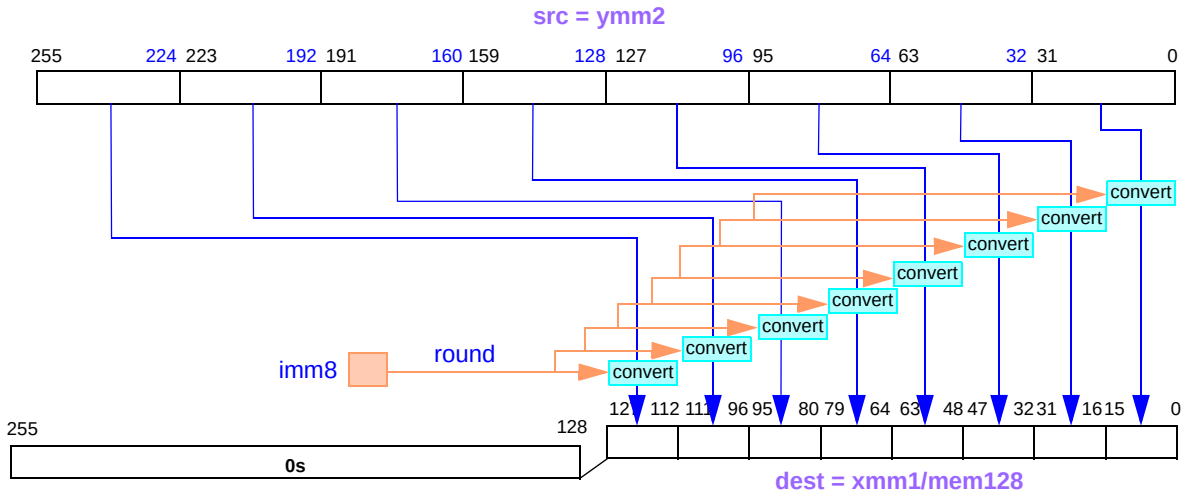
Converts packed single-precision floating-point values to packed 16-bit floating-point values and writes the converted values to the destination register or to memory. An 8-bit immediate operand provides dynamic control of rounding.

The operation of this instruction is illustrated in the following diagram.

VCVTPS2PH
128-Bit



VCVTPS2PH
256-Bit



The handling of rounding is controlled by fields in the immediate byte, as shown in the following table.

Rounding Control with Immediate Byte Operand

Mnemonic	Rounding Source (RS)	Rounding Control (RC)			
Bit	**2**	**1**	**0**	**Description**	**Notes**
Value	0	0	0	Nearest	Ignore MXCSR.RC.
		0	1	Down	
		1	0	Up	
		1	1	Truncate	
	1	X	X	Use MXCSR.RC for rounding.	

MXCSR[FTZ] has no effect on this instruction. Values within the half-precision denormal range are unconditionally converted to denormals.

This extended-form instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Converts four packed single-precision floating-point values in an XMM register to four packed 16-bit floating-point values and writes the converted values to the low-order 64 bits of the destination XMM register or to a 64-bit memory location. When the result is written to the destination XMM register, the high-order 64 bits in the destination XMM register and the upper 128 bits of the corresponding YMM register are cleared to 0s.

YMM Encoding

Converts eight packed single-precision floating-point values in a YMM register to eight packed 16-bit floating-point values and writes the converted values to the low-order 128 bits of a YMM register or to a 128-bit memory location. When the result is written to the destination YMM register, the high-order 128 bits in the register are cleared to 0s.

Instruction Support

Form	Subset	Feature Flag
VCVTPH2PH	F16C	CPUID Fn0000_0001_ECX[F16C] (bit 29)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VCVTPS2PH *xmm1/mem64*, *xmm2, imm8*	C4	$\overline{\text{RXB}}$.03	0.1111.0.01	1D /r /imm8
VCVTPS2PH *xmm1/mem128*, *ymm2, imm8*	C4	$\overline{\text{RXB}}$.03	0.1111.1.01	1D /r /imm8

Related Instructions

VCVTPH2PS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: A flag that may be set to one or cleared to zero is M (modified). Unaffected flags are blank.

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		AVX instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	VEX.W field = 1.
			A	VEX.vvvv ! = 1111b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Alignment check, #AC			F	Unaligned memory reference when alignment checking enabled.
Page fault, #PF			F	Instruction execution caused a page fault.
SIMD Floating-Point Exception, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid-operation exception (IE)			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized-operand exception (DE)			F	A source operand was a denormal value.
Overflow exception (OE)			F	Rounded result too large to fit into the format of the destination operand.
Underflow exception (UE)			F	Rounded result too small to fit into the format of the destination operand.
Precision exception (PE)			F	A result could not be represented exactly in the destination format.
F — F16C exception.				

VEXTRACTF128 Extract Packed Floating-Point Values

Extracts 128 bits of packed data from a YMM register as specified by an immediate byte operand, and writes it to either an XMM register or a 128-bit memory location.

Only bit [0] of the immediate operand is used. Operation is as follows.

- When imm8[0] = 0, copy bits [127:0] of the source to the destination.
- When imm8[0] = 1, copy bits [255:128] of the source to the destination.

This extended-form instruction has a single 256-bit encoding.

The source operand is a YMM register and the destination is either an XMM register or a 128-bit memory location. There is a third immediate byte operand.

Instruction Support

Form	Subset	Feature Flag
VEXTRACTF128	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VEXTRACTF128 *xmm/mem128, ymm, imm8*	C4	$\overline{\text{RXB}}$.03	0.1111.1.01	19 /r ib

Related Instructions

VBROADCASTF128, VINSERTF128

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.L = 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Write to a read-only data segment.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
A — AVX exception.				

VEXTRACTI128 Extract 128-bit Integer

Writes a selected 128-bit half of a YMM register to an XMM register or a 128-bit memory location based on the value of bit 0 of an immediate byte.

There is a single form of this instruction:

VEXTRACTI128 *dest, src, imm8*

If imm8[0] = 0, the lower half of the source YMM register is selected; if imm8[0] = 1, the upper half of the source register is selected.

There is a single VEX.L = 1 encoding of this instruction.

The source operand is a YMM register. The destination is either an XMM register or a 128-bit memory location. When the destination is a register, bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VEXTRACTI128	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

Instruction Encoding

	Encoding			
Mnemonic	**VEX**	**RXB.map_select**	**W.vvvv.L.pp**	**Opcode**
VEXTRACTI128 *xmm1/mem128, ymm2, imm8*	C4	$\overline{\text{RXB}}$.03	0.1111.1.01	39 /r ib

Related Instructions

VBROADCASTF128, VBROADCASTI128, VEXTRACTF128, VINSERTF128, VINSERTI128

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

VFMADDPD VFMADD132PD VFMADD213PD VFMADD231PD — Multiply and Add Packed Double-Precision Floating-Point

Multiplies together two double-precision floating-point vectors and adds the unrounded product to a third double-precision floating-point vector producing a precise result which is then rounded to double-precision based on the mode specified by the MXCSR[RC] field. The rounded sum is written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the vector equation in the comment on the right.

There are two four-operand forms:

```
VFMADDPD dest, src1, src2/mem, src3     // dest = (src1* src2/mem) + src3
VFMADDPD dest, src1, src2, src3/mem     // dest = (src1* src2) + src3/mem
```

and three three-operand forms:

```
VFMADD132PD src1, src2, src3/mem        // src1 = (src1* src3/mem) + src2
VFMADD213PD src1, src2, src3/mem        // src1 = (src2* src1) + src3/mem
VFMADD231PD src1, src2, src3/mem        // src1 = (src2* src3/mem) + src1
```

When VEX.L = 0, the vector size is 128 bits (two double-precision elements per vector) and register-based source operands are held in XMM registers.

When VEX.L = 1, the vector size is 256 bits (four double-precision elements per vector) and register-based source operands are held in YMM registers.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a memory location.

For the three-operand forms, VEX.W is 1. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is either an XMM register or a YMM register, as determined by VEX.L. When the destination is an XMM register (L = 0), bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFMADDPD	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFMADD*nnn*PD	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMADDPD *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.0.01	69 /r /is4
VFMADDPD *ymm1*, *ymm2*, *ymm3/mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.1.01	69 /r /is4
VFMADDPD *xmm1*, *xmm2, xmm3, xmm4/mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.0.01	69 /r /is4
VFMADDPD *ymm1*, *ymm2, ymm3, ymm4/mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.1.01	69 /r /is4
VFMADD132PD *xmm0, xmm1, xmm2/m128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.0.01	98 /r
VFMADD132PD *ymm0, ymm1, ymm2/m256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.1.01	98 /r
VFMADD213PD *xmm0, xmm1, xmm2/m128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.0.01	A8 /r
VFMADD213PD *ymm0, ymm1, ymm2/m256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.1.01	A8 /r
VFMADD231PD *xmm0, xmm1, xmm2/m128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.0.01	B8 /r
VFMADD231PD *ymm0, ymm1, ymm2/m256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.1.01	B8 /r

Related Instructions

VFMADDPS, VFMADD132PS, VFMADD213PS, VFMADD231PS, VFMADDSD, VFMADD132SD, VFMADD213SD, VFMADD231SD, VFMADDSS, VFMADD132SS, VFMADD213SS, VFMADD231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFMADDPS
VFMADD132PS
VFMADD213PS
VFMADD231PS

Multiply and Add Packed Single-Precision Floating-Point

Multiplies together two single-precision floating-point vectors and adds the unrounded product to a third single-precision floating-point vector producing a precise result which is then rounded to single-precision based on the mode specified by the MXCSR[RC] field. The rounded sum is written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the vector equation in the comment on the right.

There are two four-operand forms:

```
VFMADDPS dest, src1, src2/mem, src3     // dest = (src1* src2/mem) + src3
VFMADDPS dest, src1, src2, src3/mem     // dest = (src1* src2) + src3/mem
```

and three three-operand forms:

```
VFMADD132PS src1, src2, src3/mem        // src1 = (src1* src3/mem) + src2
VFMADD213PS src1, src2, src3/mem        // src1 = (src2* src1) + src3/mem
VFMADD231PS src1, src2, src3/mem        // src1 = (src2* src3/mem) + src1
```

When VEX.L = 0, the vector size is 128 bits (four single-precision elements per vector) and register-based source operands are held in XMM registers.

When VEX.L = 1, the vector size is 256 bits (eight single-precision elements per vector) and register-based source operands are held in YMM registers.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a memory location.

For the three-operand forms, VEX.W is 0. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is either an XMM register or a YMM register, as determined by VEX.L. When the destination is an XMM register (L = 0), bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFMADDPS	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFMADD*nnn*PS	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMADDPS *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.0.01	68 /r /is4
VFMADDPS *ymm1*, *ymm2*, *ymm3*/*mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.1.01	68 /r /is4
VFMADDPS *xmm1*, *xmm2, xmm3, xmm4*/*mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.0.01	68 /r /is4
VFMADDPS *ymm1*, *ymm2, ymm3, ymm4*/*mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.1.01	68 /r /is4
VFMADD132PS *xmm0, xmm1, xmm2/m128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	98 /r
VFMADD132PS *ymm0, ymm1, ymm2/m256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.1.01	98 /r
VFMADD213PS *xmm0, xmm1, xmm2/m128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	A8 /r
VFMADD213PS *ymm0, ymm1, ymm2/m256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.1.01	A8 /r
VFMADD231PS *xmm0, xmm1, xmm2/m128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	B8 /r
VFMADD231PS *ymm0, ymm1, ymm2/m256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.1.01	B8 /r

Related Instructions

VFMADDPD, VFMADD132PD, VFMADD213PD, VFMADD231PD, VFMADDSD, VFMADD132SD, VFMADD213SD, VFMADD231SD, VFMADDSS, VFMADD132SS, VFMADD213SS, VFMADD231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFMADDSD VFMADD132SD VFMADD213SD VFMADD231SD — Multiply and Add Scalar Double-Precision Floating-Point

Multiplies together two double-precision floating-point values and adds the unrounded product to a third double-precision floating-point value producing a precise result which is then rounded to double-precision based on the mode specified by the MXCSR[RC] field. The rounded sum is written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the equation in the comment on the right.

There are two four-operand forms:

```
VFMADDSD dest, src1, src2/mem64, src3     // dest = (src1* src2/mem64) + src3
VFMADDSD dest, src1, src2, src3/mem64     // dest = (src1* src2) + src3/mem64
```

and three three-operand forms:

```
VFMADD132SD src1, src2, src3/mem64        // src1 = (src1* src3/mem64) + src2
VFMADD213SD src1, src2, src3/mem64        // src1 = (src2* src1) + src3/mem64
VFMADD231SD src1, src2, src3/mem64        // src1 = (src2* src3/mem64) + src1
```

All 64-bit double-precision floating-point register-based operands are held in the lower quadword of XMM registers. The result is written to the lower quadword of the destination register. For those instructions that use a memory-based operand, one of the source operands is a 64-bit value read from memory.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a 64-bit memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a 64-bit memory location.

For the three-operand forms, VEX.W is 1. The first and second operands are registers and the third operand is either a register or a 64-bit memory location.

The destination is an XMM register. When the result is written to the destination XMM register, bits [127:64] of the destination and bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFMADDSD	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFMADD*nnn*SD	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMADDSD *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.X.01	6B /r /is4
VFMADDSD *xmm1*, *xmm2, xmm3, xmm4/mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.X.01	6B /r /is4
VFMADD132SD *xmm0, xmm1, xmm2/m128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.X.01	99 /r
VFMADD213SD *xmm0, xmm1, xmm2/m128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.X.01	A9 /r
VFMADD231SD *xmm0, xmm1, xmm2/m128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.X.01	B9 /r

Related Instructions

VFMADDPD, VFMADD132PD, VFMADD213PD, VFMADD231PD, VFMADDPS, VFMADD132PS, VFMADD213PS, VFMADD231PS, VFMADDSS, VFMADD132SS, VFMADD213SS, VFMADD231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Non-aligned memory reference when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFMADDSS Multiply and Add
VFMADD132SS Scalar Single-Precision Floating-Point
VFMADD213SS
VFMADD231SS

Multiplies together two single-precision floating-point values and adds the unrounded product to a third single-precision floating-point value producing a precise result which is then rounded to single-precision based on the mode specified by the MXCSR[RC] field. The rounded sum is written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the equation in the comment on the right.

There are two four-operand forms:

```
VFMADDSS dest, src1, src2/mem32, src3     // dest = (src1* src2/mem32) + src3
VFMADDSS dest, src1, src2, src3/mem32     // dest = (src1* src2) + src3/mem32
```

and three three-operand forms:

```
VFMADD132SS src1, src2, src3/mem32        // src1 = (src1* src3/mem32) + src2
VFMADD213SS src1, src2, src3/mem32        // src1 = (src2* src1) + src3/mem32
VFMADD231SS src1, src2, src3/mem32        // src1 = (src2* src3/mem32) + src1
```

All 32-bit single-precision floating-point register-based operands are held in the lower doubleword of XMM registers. The result is written to the low doubleword of the destination register. For those instructions that use a memory-based operand, one of the source operands is a 32-bit value read from memory.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a 32-bit memory location and the third source is a register.
- When VEX.W = 1, the second source is a a register and the third source is either a register or a 32-bit memory location.

For the three-operand forms, VEX.W is 0. The first and second operands are registers and the third operand is either a register or a 32-bit memory location.

The destination is an XMM register. When the result is written to the destination XMM register, bits [127:32] of the destination and bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFMADDSS	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFMADD*nnn*SS	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMADDSS *xmm1, xmm2, xmm3/mem32, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.X.01	6A /r /is4
VFMADDSS *xmm1*, *xmm2*, *xmm3, xmm4*/*mem32*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.X.01	6A /r /is4
VFMADD132SS *xmm1*, *xmm2*, *xmm3*/*mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.X.01	99 /r
VFMADD213SS *xmm1*, *xmm2*, *xmm3*/*mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.X.01	A9 /r
VFMADD231SS *xmm1*, *xmm2*, *xmm3*/*mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.X.01	B9 /r

Related Instructions

VFMADDPD, VFMADD132PD, VFMADD213PD, VFMADD231PD, VFMADDPS, VFMADD132PS, VFMADD213PS, VFMADD231PS, VFMADDSD, VFMADD132SD, VFMADD213SD, VFMADD231SD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Non-aligned memory reference when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFMADDSUBPD VFMADDSUB132PD VFMADDSUB213PD VFMADDSUB231PD — Multiply with Alternating Add/Subtract Packed Double-Precision Floating-Point

Multiplies together two double-precision floating-point vectors, adds odd elements of the unrounded product to odd elements of a third double-precision floating-point vector, and subtracts even elements of the third floating point vector from even elements of unrounded product. The precise result of each addition or subtraction is then rounded to double-precision based on the mode specified by the MXCSR[RC] field and written to the corresponding element of the destination.

The role of each of the source operands specified by the assembly language prototypes given below is reflected in the equation in the comment on the right.

There are two four-operand forms:

VFMADDSUBPD *dest, src1, src2/mem, src3*	// $dest_{odd} = (src1_{odd} * src2_{odd}/mem_{odd}) + src3_{odd}$ // $dest_{even} = (src1_{even} * src2_{even}/mem_{even}) - src3_{even}$
VFMADDSUBPD *dest, src1, src2, src3/mem*	// $dest_{odd} = (src1_{odd} * src2_{odd}) + src3_{odd}/mem_{odd}$ // $dest_{even} = (src1_{even} * src2_{even}) - src3_{even}/mem_{even}$

and three three-operand forms:

VFMADDSUB132PD *src1, src2, src3/mem*	// $src1_{odd} = (src1_{odd} * src3_{odd}/mem_{odd}) + src2_{odd}$ // $src1_{even} = (src1_{even} * src3_{even}/mem_{even}) - src2_{even}$
VFMADDSUB213PD *src1, src2, src3/mem*	// $src1_{odd} = (src2_{odd} * src1_{odd}) + src3_{odd}/mem_{odd}$ // $src1_{even} = (src2_{even} * src1_{even}) - src3_{even}/mem_{even}$
VFMADDSUB231PD *src1, src2, src3/mem*	// $src1_{odd} = (src2_{odd} * src3_{odd}/mem_{odd}) + src1_{odd}$ // $src1_{even} = (src2_{even} * src3_{even}/mem_{even}) - src1_{even}$

When VEX.L = 0, the vector size is 128 bits (two double-precision elements per vector) and register-based source operands are held in XMM registers.

When VEX.L = 1, the vector size is 256 bits (four double-precision elements per vector) and register-based source operands are held in YMM registers.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a memory location.

For the three-operand forms, VEX.W is 1. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is either an XMM register or a YMM register, as determined by VEX.L. When the destination is an XMM register (L = 0), bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFMADDSUBPD	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFMADDSUB*nnn*PD	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMADDSUBPD *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.0.01	5D /r /is4
VFMADDSUBPD *ymm1*, *ymm2*, *ymm3*/*mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.1.01	5D /r /is4
VFMADDSUBPD *xmm1*, *xmm2, xmm3, xmm4*/*mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.0.01	5D /r /is4
VFMADDSUBPD *ymm1*, *ymm2, ymm3, ymm4*/*mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.1.01	5D /r /is4
VFMADDSUB132PD *xmm1*, *xmm2, xmm3*/*mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	96 /r
VFMADDSUB132PD *ymm1*, *ymm2, ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	96 /r
VFMADDSUB213PD *xmm1*, *xmm2, xmm3*/*mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	A6 /r
VFMADDSUB213PD *ymm1*, *ymm2, ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	A6 /r
VFMADDSUB231PD *xmm1*, *xmm2, xmm3*/*mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	B6 /r
VFMADDSUB231PD *ymm1*, *ymm2, ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	B6 /r

Related Instructions

VFMSUBADDPD, VFMSUBADD132PD, VFMSUBADD213PD, VFMSUBADD231PD, VFMADDSUBPS, VFMADDSUB132PS, VFMADDSUB213PS, VFMADDSUB231PS, VFMSUBADDPS, VFMSUBADD132PS, VFMSUBADD213PS, VFMSUBADD231PS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFMADDSUBPS VFMADDSUB132PS VFMADDSUB213PS VFMADDSUB231PS — Multiply with Alternating Add/Subtract Packed Single-Precision Floating-Point

Multiplies together two single-precision floating-point vectors, adds odd elements of the unrounded product to odd elements of a third single-precision floating-point vector, and subtracts even elements of the third floating point vector from even elements of unrounded product. The precise result of each addition or subtraction is then rounded to single-precision based on the mode specified by the MXCSR[RC] field and written to the corresponding element of the destination.

The role of each of the source operands specified by the assembly language prototypes given below is reflected in the equation in the comment on the right.

There are two four-operand forms:

VFMADDSUBPS *dest, src1, src2/mem, src3*	// $dest_{odd} = (src1_{odd} * src2_{odd}/mem_{odd}) + src3_{odd}$ // $dest_{even} = (src1_{even} * src2_{even}/mem_{even}) - src3_{even}$
VFMADDSUBPS *dest, src1, src2, src3/mem*	// $dest_{odd} = (src1_{odd} * src2_{odd}) + src3_{odd}/mem_{odd}$ // $dest_{even} = (src1_{even} * src2_{even}) - src3_{even}/mem_{even}$

and three three-operand forms:

VFMADDSUB132PS *src1, src2, src3/mem*	// $src1_{odd} = (src1_{odd} * src3_{odd}/mem_{odd}) + src2_{odd}$ // $src1_{even} = (src1_{even} * src3_{even}/mem_{even}) - src2_{even}$
VFMADDSUB213PS *src1, src2, src3/mem*	// $src1_{odd} = (src2_{odd} * src1_{odd}) + src3_{odd}/mem_{odd}$ // $src1_{even} = (src2_{even} * src1_{even}) - src3_{even}/mem_{even}$
VFMADDSUB231PS *src1, src2, src3/mem*	// $src1_{odd} = (src2_{odd} * src3_{odd}/mem_{odd}) + src1_{odd}$ // $src1_{even} = (src2_{even} * src3_{even}/mem_{even}) - src1_{even}$

When VEX.L = 0, the vector size is 128 bits (four single-precision elements per vector) and register-based source operands are held in XMM registers.

When VEX.L = 1, the vector size is 256 bits (eight single-precision elements per vector) and register-based source operands are held in YMM registers.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a memory location.

For the three-operand forms, VEX.W is 0. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is either an XMM register or a YMM register, as determined by VEX.L. When the destination is an XMM register (L = 0), bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFMADDSUBPS	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFMADDSUB*nnn*PS	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMADDSUBPS *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.0.01	5C /r /is4
VFMADDSUBPS *ymm1*, *ymm2*, *ymm3/mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.1.01	5C /r /is4
VFMADDSUBPS *xmm1*, *xmm2, xmm3, xmm4/mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.0.01	5C /r /is4
VFMADDSUBPS *ymm1*, *ymm2, ymm3, ymm4/mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.1.01	5C /r /is4
VFMADDSUB132PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.0.01	96 /r
VFMADDSUB132PS *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.1.01	96 /r
VFMADDSUB213PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.0.01	A6 /r
VFMADDSUB213PS *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.1.01	A6 /r
VFMADDSUB231PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.0.01	B6 /r
VFMADDSUB231PS *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.1.01	B6 /r

Related Instructions

VFMADDSUBPD, VFMADDSUB132PD, VFMADDSUB213PD, VFMADDSUB231PD, VFMSUBADDPD, VFMSUBADD132PD, VFMSUBADD213PD, VFMSUBADD231PD, VFMSUBADDPS, VFMSUBADD132PS, VFMSUBADD213PS, VFMSUBADD231PS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFMSUBADDPD VFMSUBADD132PD VFMSUBADD213PD VFMSUBADD231PD — Multiply with Alternating Subtract/Add Packed Double-Precision Floating-Point

Multiplies together two double-precision floating-point vectors, adds even elements of the unrounded product to even elements of a third double-precision floating-point vector, and subtracts odd elements of the third floating point vector from odd elements of unrounded product. The precise result of each addition or subtraction is then rounded to double-precision based on the mode specified by the MXCSR[RC] field and written to the corresponding element of the destination.

The role of each of the source operands specified by the assembly language prototypes given below is reflected in the equation in the comment on the right.

There are two four-operand forms:

VFMSUBADDPD *dest, src1, src2/mem, src3*	// $dest_{odd} = (src1_{odd} * src2_{odd}/mem_{odd}) - src3_{odd}$ // $dest_{even} = (src1_{even} * src2_{even}/mem_{even}) + src3_{even}$
VFMSUBADDPD *dest, src1, src2, src3/mem*	// $dest_{odd} = (src1_{odd} * src2_{odd}) - src3_{odd}/mem_{odd}$ // $dest_{even} = (src1_{even} * src2_{even}) + src3_{even}/mem_{even}$

and three three-operand forms:

VFMSUBADD132PD *src1, src2, src3/mem*	// $src1_{odd} = (src1_{odd} * src3_{odd}/mem_{odd}) - src2_{odd}$ // $src1_{even} = (src1_{even} * src3_{even}/mem_{even}) + src2_{even}$
VFMSUBADD213PD *src1, src2, src3/mem*	// $src1_{odd} = (src2_{odd} * src1_{odd}) - src3_{odd}/mem_{odd}$ // $src1_{even} = (src2_{even} * src1_{even}) + src3_{even}/mem_{even}$
VFMSUBADD231PD *src1, src2, src3/mem*	// $src1_{odd} = (src2_{odd} * src3_{odd}/mem_{odd}) - src1_{odd}$ // $src1_{even} = (src2_{even} * src3_{even}/mem_{even}) + src1_{even}$

For VEX.L = 0, vector size is 128 bits and register-based operands are held in XMM registers. For VEX.L = 1, vector size is 256 bits and register-based operands are held in YMM registers.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source operand is either a register or a memory location.

For the three-operand forms, VEX.W is 1. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is either an XMM register or a YMM register, as determined by VEX.L. When the destination is an XMM register (L = 0), bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFMSUBADDPD	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFMSUBADD*nnn*PD	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMSUBADDPD *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.0.01	5F /r /is4
VFMSUBADDPD *ymm1*, *ymm2*, *ymm3*/*mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.1.01	5F /r /is4
VFMSUBADDPD *xmm1*, *xmm2, xmm3, xmm4*/*mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.0.01	5F /r /is4
VFMSUBADDPD *ymm1*, *ymm2, ymm3, ymm4*/*mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.1.01	5F /r /is4
VFMSUBADD132PD *xmm1, xmm2, xmm3*/*mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	97 /r
VFMSUBADD132PD *ymm1, ymm2, ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	97 /r
VFMSUBADD213PD *xmm1, xmm2, xmm3*/*mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	A7 /r
VFMSUBADD213PD *ymm1, ymm2, ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	A7 /r
VFMSUBADD231PD *xmm1, xmm2, xmm3*/*mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	B7 /r
VFMSUBADD231PD *ymm1, ymm2, ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	B7 /r

Related Instructions

VFMADDSUBPD, VFMADDSUB132PD, VFMADDSUB213PD, VFMADDSUB231PD, VFMADDSUBPS, VFMADDSUB132PS, VFMADDSUB213PS, VFMADDSUB231PS, VFMSUBADDPS, VFMSUBADD132PS, VFMSUBADD213PS, VFMSUBADD231PS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFMSUBADDPS VFMSUBADD132PS VFMSUBADD213PS VFMSUBADD231PS — Multiply with Alternating Subtract/Add Packed Single-Precision Floating-Point

Multiplies together two single-precision floating-point vectors, adds even elements of the unrounded product to even elements of a third single-precision floating-point vector, and subtracts odd elements of the third floating point vector from odd elements of unrounded product. The precise result of each addition or subtraction is then rounded to single-precision based on the mode specified by the MXCSR[RC] field and written to the corresponding element of the destination.

The role of each of the source operands specified by the assembly language prototypes given below is reflected in the equation in the comment on the right.

There are two four-operand forms:

VFMSUBADDPS *dest, src1, src2/mem, src3*	// $dest_{odd} = (src1_{odd} * src2_{odd}/mem_{odd}) - src3_{odd}$ // $dest_{even} = (src1_{even} * src2_{even}/mem_{even}) + src3_{even}$
VFMSUBADDPS *dest, src1, src2, src3/mem*	// $dest_{odd} = (src1_{odd} * src2_{odd}) - src3_{odd}/mem_{odd}$ // $dest_{even} = (src1_{even} * src2_{even}) + src3_{even}/mem_{even}$

and three three-operand forms:

VFMSUBADD132PS *src1, src2, src3/mem*	// $src1_{odd} = (src1_{odd} * src3_{odd}/mem_{odd}) - src2_{odd}$ // $src1_{even} = (src1_{even} * src3_{even}/mem_{even}) + src2_{even}$
VFMSUBADD213PS *src1, src2, src3/mem*	// $src1_{odd} = (src2_{odd} * src1_{odd}) - src3_{odd}/mem_{odd}$ // $src1_{even} = (src2_{even} * src1_{even}) + src3_{even}/mem_{even}$
VFMSUBADD231PS *src1, src2, src3/mem*	// $src1_{odd} = (src2_{odd} * src3_{odd}/mem_{odd}) - src1_{odd}$ // $src1_{even} = (src2_{even} * src3_{even}/mem_{even}) + src1_{even}$

When VEX.L = 0, the vector size is 128 bits (four single-precision elements per vector) and register-based source operands are held in XMM registers.

When VEX.L = 1, the vector size is 256 bits (eight single-precision elements per vector) and register-based source operands are held in YMM registers.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a memory location.

For the three-operand forms, VEX.W is 0. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is either an XMM register or a YMM register, as determined by VEX.L. When the destination is an XMM register (L = 0), bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFMSUBADDPS	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFMSUBADD*nnn*PS	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMSUBADDPS *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.0.01	5E /r /is4
VFMSUBADDPS *ymm1*, *ymm2*, *ymm3/mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.1.01	5E /r /is4
VFMSUBADDPS *xmm1*, *xmm2, xmm3, xmm4/mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.0.01	5E /r /is4
VFMSUBADDPS *ymm1*, *ymm2, ymm3, ymm4/mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.1.01	5E /r /is4
VFMSUBADD132PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.00010	0.$\overline{src2}$.0.01	97 /r
VFMSUBADD132PS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00010	0.$\overline{src2}$.1.01	97 /r
VFMSUBADD213PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.00010	0.$\overline{src2}$.0.01	A7 /r
VFMSUBADD213PS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00010	0.$\overline{src2}$.1.01	A7 /r
VFMSUBADD231PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.00010	0.$\overline{src2}$.0.01	B7 /r
VFMSUBADD231PS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00010	0.$\overline{src2}$.1.01	B7 /r

Related Instructions

VFMADDSUBPD, VFMADDSUB132PD, VFMADDSUB213PD, VFMADDSUB231PD, VFMADDSUBPS, VFMADDSUB132PS, VFMADDSUB213PS, VFMADDSUB231PS, VFMSUBADDPD, VFMSUBADD132PD, VFMSUBADD213PD, VFMSUBADD231PD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFMSUBPD VFMSUB132PD VFMSUB213PD VFMSUB231PD — Multiply and Subtract Packed Double-Precision Floating-Point

Multiplies together two double-precision floating-point vectors and subtracts a third double-precision floating-point vector from the unrounded product to produce a precise intermediate result. The intermediate result is then rounded to double-precision based on the mode specified by the MXCSR[RC] field and written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the vector equation in the comment on the right.

There are two four-operand forms:

```
VFMSUBPD dest, src1, src2/mem, src3     // dest = (src1* src2/mem) − src3
VFMSUBPD dest, src1, src2, src3/mem     // dest = (src1* src2) − src3/mem
```

and three three-operand forms:

```
VFMSUB132PD src1, src2, src3/mem        // src1 = (src1* src3/mem) − src2
VFMSUB213PD src1, src2, src3/mem        // src1 = (src2* src1) − src3/mem
VFMSUB231PD src1, src2, src3/mem        // src1 = (src2* src3/mem) − src1
```

For VEX.L = 0, vector size is 128 bits and register-based operands are held in XMM registers. For VEX.L = 1, vector size is 256 bits and register-based operands are held in YMM registers.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a memory location.

For the three-operand forms, VEX.W is 1. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is either an XMM register or a YMM register, as determined by VEX.L. When the destination is an XMM register (L = 0), bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFMSUBPD	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFMSUB*nnn*PD	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMSUBPD *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.0.01	6D /r /is4
VFMSUBPD *ymm1*, *ymm2*, *ymm3*/*mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.1.01	6D /r /is4
VFMSUBPD *xmm1*, *xmm2, xmm3, xmm4*/*mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.0.01	6D /r /is4
VFMSUBPD *ymm1*, *ymm2, ymm3, ymm4*/*mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.1.01	6D /r /is4
VFMSUB132PD *xmm1*, *xmm2*, *xmm3*/*mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	9A /r
VFMSUB132PD *ymm1*, *ymm2*, *ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	9A /r
VFMSUB213PD *xmm1*, *xmm2*, *xmm3*/*mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	AA /r
VFMSUB213PD *ymm1*, *ymm2*, *ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	AA /r
VFMSUB231PD *xmm1*, *xmm2*, *xmm3*/*mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	BA /r
VFMSUB231PD *ymm1*, *ymm2*, *ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	BA /r

Related Instructions

VFMSUBPS, VFMSUB132PS, VFMSUB213PS, VFMSUB231PPS, VFMSUBSD, VFMSUB-132SD, VFMSUB213SD, VFMSUB2P31SD, VFMSUBSS, VFMSUB132SS, VFMSUB213SS, VFMSUBP231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFMSUBPS VFMSUB132PS VFMSUB213PS VFMSUB231PS — Multiply and Subtract Packed Single-Precision Floating-Point

Multiplies together two single-precision floating-point vectors and subtracts a third single-precision floating-point vector from the unrounded product to produce a precise intermediate result. The intermediate result is then rounded to single-precision based on the mode specified by the MXCSR[RC] field and written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the vector equation in the comment on the right.

There are two four-operand forms:

```
VFMSUBPS dest, src1, src2/mem, src3     // dest = (src1* src2/mem) − src3
VFMSUBPS dest, src1, src2, src3/mem     // dest = (src1* src2) − src3/mem
```

and three three-operand forms:

```
VFMSUB132PS src1, src2, src3/mem        // src1 = (src1* src3/mem) − src2
VFMSUB213PS src1, src2, src3/mem        // src1 = (src2* src1) − src3/mem
VFMSUB231PS src1, src2, src3/mem        // src1 = (src2* src3/mem) − src1
```

When VEX.L = 0, the vector size is 128 bits (four single-precision elements per vector) and register-based source operands are held in XMM registers.

When VEX.L = 1, the vector size is 256 bits (eight single-precision elements per vector) and register-based source operands are held in YMM registers.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a memory location and the third source is a register.
- When VEX.W = 1, the second source is a a register and the third source is either a register or a memory location.

For the three-operand forms, VEX.W is 0. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is either an XMM register or a YMM register, as determined by VEX.L. When the destination is an XMM register (L = 0), bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFMSUBPS	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFMSUB*nnn*PS	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMSUBPS *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.0.01	6C /r /is4
VFMSUBPS *ymm1*, *ymm2*, *ymm3*/*mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.1.01	6C /r /is4
VFMSUBPS *xmm1*, *xmm2, xmm3, xmm4*/*mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.0.01	6C /r /is4
VFMSUBPS *ymm1*, *ymm2, ymm3, ymm4*/*mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.1.01	6C /r /is4
VFMSUB132PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	9A /r
VFMSUB132PS *ymm1*, *ymm2*, *ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.1.01	9A /r
VFMSUB213PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	AA /r
VFMSUB213PS *ymm1*, *ymm2*, *ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.1.01	AA /r
VFMSUB231PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	BA /r
VFMSUB231PS *ymm1*, *ymm2*, *ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.1.01	BA /r

Related Instructions

VFMSUBPD, VFMSUB132PD, VFMSUB213PD, VFMSUB231PD, VFMSUBSD, VFMSUB-132SD, VFMSUB213SD, VFMSUB231SD, VFMSUBSS, VFMSUB132SS, VFMSUB213SS, VFMSUB231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFMSUBSD VFMSUB132SD VFMSUB213SD VFMSUB231SD — Multiply and Subtract Scalar Double-Precision Floating-Point

Multiplies together two double-precision floating-point values and subtracts a third double-precision floating-point value from the unrounded product to produce a precise intermediate result. The intermediate result is then rounded to double-precision based on the mode specified by the MXCSR[RC] field and written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the vector equation in the comment on the right.

There are two four-operand forms:

```
VFMSUBSD dest, src1, src2/mem, src3     // dest = (src1* src2/mem) − src3
VFMSUBSD dest, src1, src2, src3/mem     // dest = (src1* src2) − src3/mem
```

and three three-operand forms:

```
VFMSUB132SD src1, src2, src3/mem        // src1 = (src1* src3/mem) − src2
VFMSUB213SD src1, src2, src3/mem        // src1 = (src2* src1) − src3/mem
VFMSUB231SD src1, src2, src3/mem        // src1 = (src2* src3/mem) − src1
```

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or 64-bit memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is a register or 64-bit memory location.

For the three-operand forms, VEX.W is 1. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is an XMM register. When the result is written to the destination XMM register, bits [127:64] of the destination and bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFMSUBSD	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFMSUB*nnn*SD	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMSUBSD *xmm1, xmm2, xmm3/mem64, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.X.01	6F /r /is4
VFMSUBSD *xmm1*, *xmm2, xmm3, xmm4*/*mem64*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.X.01	6F /r /is4
VFMSUB132SD *xmm1, xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.X.01	9B /r
VFMSUB213SD *xmm1, xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.X.01	AB /r
VFMSUB231SD *xmm1, xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.X.01	BB /r

Related Instructions

VFMSUBPD, VFMSUB132PD, VFMSUB213PD, VFMSUB231PD, VFMSUBPS, VFMSUB-132PS, VFMSUB213PS, VFMSUB231PS, VFMSUBSS, VFMSUB132SS, VFMSUB213SS, VFM-SUB231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Non-aligned memory reference when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFMSUBSS VFMSUB132SS VFMSUB213SS VFMSUB231SS — Multiply and Subtract Scalar Single-Precision Floating-Point

Multiplies together two single-precision floating-point values and subtracts a third single-precision floating-point value from the unrounded product to produce a precise intermediate result. The intermediate result is then rounded to single-precision based on the mode specified by the MXCSR[RC] field and written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the vector equation in the comment on the right.

There are two four-operand forms:

```
VFMSUBSS dest, src1, src2/mem, src3     // dest = (src1* src2/mem) − src3
VFMSUBSS dest, src1, src2, src3/mem     // dest = (src1* src2) − src3/mem
```

and three three-operand forms:

```
VFMSUB132SS src1, src2, src3/mem        // src1 = (src1* src3/mem) − src2
VFMSUB213SS src1, src2, src3/mem        // src1 = (src2* src1) − src3/mem
VFMSUB231SS src1, src2, src3/mem        // src1 = (src2* src3/mem) − src1
```

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or 32-bit memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is a register or 32-bit memory location.

For the three-operand forms, VEX.W is 0. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is an XMM register. When the result is written to the destination XMM register, bits [127:32] of the XMM register and bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFMSUBSS	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFMSUB*nnn*SS	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFMSUBSS *xmm1, xmm2, xmm3/mem32, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.X.01	6E /r /is4
VFMSUBSS *xmm1*, *xmm2*, *xmm3, xmm4*/*mem32*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.X.01	6E /r /is4
VFMSUB132SS *xmm1, xmm2, xmm3/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.X.01	9B /r
VFMSUB213SS *xmm1, xmm2, xmm3/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.X.01	AB /r
VFMSUB231SS *xmm1, xmm2, xmm3/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.X.01	BB /r

Related Instructions

VFMSUBPD, VFMSUB132PD, VFMSUB213PD, VFMSUB231PD, VFMSUBPS, VFMSUB-132PS, VFMSUB213PS, VFMSUB231PS, VFMSUBSD, VFMSUB132SD, VFMSUB213SD, VFMSUB231SD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Non-aligned memory reference when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFNMADDPD VFNMADD132PD VFNMADD213PD VFNMADD231PD — Negative Multiply and Add Packed Double-Precision Floating-Point

Multiplies together two double-precision floating-point vectors, negates the unrounded product, and adds it to a third double-precision floating-point vector. The precise result is then rounded to double-precision based on the mode specified by the MXCSR[RC] field and written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the vector equation in the comment on the right.

There are two four-operand forms:

```
VFNMADDPD dest, src1, src2/mem, src3    // dest = −(src1* src2/mem) + src3
VFNMADDPD dest, src1, src2, src3/mem    // dest = −(src1* src2) + src3/mem
```

and three three-operand forms:

```
VFNMADD132PD src1, src2, src3/mem    // src1 = −(src1* src3/mem) + src2
VFNMADD213PD src1, src2, src3/mem    // src1 = −(src2* src1) + src3/mem
VFNMADD231PD src1, src2, src3/mem    // src1 = −(src2* src3/mem) + src1
```

When VEX.L = 0, the vector size is 128 bits (two double-precision elements per vector) and register-based source operands are held in XMM registers.

When VEX.L = 1, the vector size is 256 bits (four double-precision elements per vector) and register-based source operands are held in YMM registers.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a memory location.

For the three-operand forms, VEX.W is 1. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is either an XMM register or a YMM register, as determined by VEX.L. When the destination is an XMM register (L = 0), bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFNMADDPD	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFNMADD*nnn*PD	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFNMADDPD *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.0.01	79 /r /is4
VFNMADDPD *ymm1*, *ymm2*, *ymm3/mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.1.01	79 /r /is4
VFNMADDPD *xmm1*, *xmm2, xmm3, xmm4/mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.0.01	79 /r /is4
VFNMADDPD *ymm1*, *ymm2, ymm3, ymm4/mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.1.01	79 /r /is4
VFNMADD132PD *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.0.01	9C /r
VFNMADD132PD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.1.01	9C /r
VFNMADD213PD *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.0.01	AC /r
VFNMADD213PD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.1.01	AC /r
VFNMADD231PD *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.0.01	BC /r
VFNMADD231PD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.1.01	BC /r

Related Instructions

VFNMADDPS, VFNMADD132PS, VFNMADD213PS, VFNMADD231PS, VFNMADDSD, VFNMADD132SD, VFNMADD213SD, VFNMADD231SD, VFNMADDSS, VFNMADD132SS, VFNMADD213SS, VFNMADD231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFNMADDPS
VFNMADD132PS
VFNMADD213PS
VFNMADD231PS

Negative Multiply and Add Packed Single-Precision Floating-Point

Multiplies together two single-precision floating-point vectors, negates the unrounded product, and adds it to a third single-precision floating-point vector. The precise result is then rounded to single-precision based on the mode specified by the MXCSR[RC] field and written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the vector equation in the comment on the right.

There are two four-operand forms:

```
VFNMADDPS dest, src1, src2/mem, src3      // dest = −(src1* src2/mem) + src3
VFNMADDPS dest, src1, src2, src3/mem      // dest = −(src1* src2) + src3/mem
```

and three three-operand forms:

```
VFNMADD132PS src1, src2, src3/mem         // src1 = −(src1* src3/mem) + src2
VFNMADD213PS src1, src2, src3/mem         // src1 = −(src2* src1) + src3/mem
VFNMADD231PS src1, src2, src3/mem         // src1 = −(src2* src3/mem) + src1
```

When VEX.L = 0, the vector size is 128 bits (four single-precision elements per vector) and register-based source operands are held in XMM registers.

When VEX.L = 1, the vector size is 256 bits (eight single-precision elements per vector) and register-based source operands are held in YMM registers.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a memory location.

For the three-operand forms, VEX.W is 0. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is either an XMM register or a YMM register, as determined by VEX.L. When the destination is an XMM register (L = 0), bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFNMADDPS	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFNMADD*nnn*PS	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFNMADDPS *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.0.01	78 /r /is4
VFNMADDPS *ymm1*, *ymm2*, *ymm3/mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.1.01	78 /r /is4
VFNMADDPS *xmm1*, *xmm2, xmm3, xmm4*/*mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.0.01	78 /r /is4
VFNMADDPS *ymm1*, *ymm2, ymm3, ymm4*/*mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.1.01	78 /r /is4
VFNMADD132PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	9C / r
VFNMADD132PS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.1.01	9C / r
VFNMADD213PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	AC / r
VFNMADD213PS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.1.01	AC / r
VFNMADD231PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	BC / r
VFNMADD231PS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.1.01	BC / r

Related Instructions

VFNMADDPD, VFNMADD132PD, VFNMADD213PD, VFNMADD231PD, VFNMADDSD, VFNMADD132SD, VFNMADD213SD, VFNMADD231SD, VFNMADDSS, VFNMADD132SS, VFNMADD213SS, VFNMADD231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFNMADDSD VFNMADD132SD VFNMADD213SD VFNMADD231SD — Negative Multiply and Add Scalar Double-Precision Floating-Point

Multiplies together two double-precision floating-point values, negates the unrounded product, and adds it to a third double-precision floating-point value. The precise result is then rounded to double-precision based on the mode specified by the MXCSR[RC] field and written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the equation in the comment on the right.

There are two four-operand forms:

```
VFNMADDSD dest, src1, src2/mem, src3     // dest = −(src1* src2/mem) + src3
VFNMADDSD dest, src1, src2, src3/mem     // dest = −(src1* src2) + src3/mem
```

and three three-operand forms:

```
VFNMADD132SD src1, src2, src3/mem        // src1 = −(src1* src3/mem) + src2
VFNMADD213SD src1, src2, src3/mem        // src1 = −(src2* src1) + src3/mem
VFNMADD231SD src1, src2, src3/mem        // src1 = −(src2* src3/mem) + src1
```

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or 64-bit memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is a register or 64-bit memory location.

For the three-operand forms, VEX.W is 1. The first and second operands are registers and the third operand is either a register or a 64-bit memory location.

The destination is an XMM register. When the result is written to the destination, bits [127:64] of the XMM register and bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFNMADDSD	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFNMADD*nnn*SD	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFNMADDSD *xmm1, xmm2, xmm3/mem64, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.X.01	7B /r /is4
VFNMADDSD *xmm1*, *xmm2, xmm3, xmm4/mem64*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.X.01	7B /r /is4
VFNMADD132SD *xmm1, xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.X.01	9D /r
VFNMADD213SD *xmm1, xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.X.01	AD /r
VFNMADD231SD *xmm1, xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.X.01	BD /r

Related Instructions

VFNMADDPD, VFNMADD132PD, VFNMADD213PD, VFNMADD231PD, VFNMADDPS, VFNMADD132PS, VFNMADD213PS, VFNMADD231PS, VFNMADDSS, VFNMADD132SS, VFNMADD213SS, VFNMADD231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Non-aligned memory reference when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFNMADDSS VFNMADD132SS VFNMADD213SS VFNMADD231SS — Negative Multiply and Add Scalar Single-Precision Floating-Point

Multiplies together two single-precision floating-point values, negates the unrounded product, and adds it to a third single-precision floating-point value. The precise result is then rounded to single-precision based on the mode specified by the MXCSR[RC] field and written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the equation in the comment on the right.

There are two four-operand forms:

```
VFNMADDSS dest, src1, src2/mem, src3     // dest = −(src1* src2/mem) + src3
VFNMADDSS dest, src1, src2, src3/mem     // dest = −(src1* src2) + src3/mem
```

and three three-operand forms:

```
VFNMADD132SS src1, src2, src3/mem        // src1 = −(src1* src3/mem) + src2
VFNMADD213SS src1, src2, src3/mem        // src1 = −(src2* src1) + src3/mem
VFNMADD231SS src1, src2, src3/mem        // src1 = −(src2* src3/mem) + src1
```

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or 32-bit memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is a register or 32-bit memory location.

For the three-operand forms, VEX.W is 0. The first and second operands are registers and the third operand is either a register or a 32-bit memory location.

The destination is an XMM register. When the result is written to the destination, bits [127:32] of the XMM register and bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFNMADDSS	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFNMADD*nnn*SS	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFNMADDSS *xmm1, xmm2, xmm3/mem32, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.X.01	7A /r /is4
VFNMADDSS *xmm1*, *xmm2*, *xmm3, xmm4/mem32*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.X.01	7A /r /is4
VFNMADD132SS *xmm1, xmm2, xmm3/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.X.01	9D /r
VFNMADD213SS *xmm1, xmm2, xmm3/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.X.01	AD /r
VFNMADD231SS *xmm1, xmm2, xmm3/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.X.01	BD /r

Related Instructions

VFNMADDPD, VFNMADD132PD, VFNMADD213PD, VFNMADD231PD, VFNMADDPS, VFNMADD132PS, VFNMADD213PS, VFNMADD231PS, VFNMADDSS, VFNMADD132SS, VFNMADD213SS, VFNMADD231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Non-aligned memory reference when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFNMSUBPD VFNMSUB132PD VFNMSUB213PD VFNMSUB231PD — Negative Multiply and Subtract Packed Double-Precision Floating-Point

Multiplies together two double-precision floating-point vectors, negates the unrounded product, and subtracts a third double-precision floating-point vector from it. The precise result is then rounded to double-precision based on the mode specified by the MXCSR[RC] field and written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the vector equation in the comment on the right.

There are two four-operand forms:

```
VFNMSUBPD dest, src1, src2/mem, src3    // dest = −(src1* src2/mem) − src3
VFNMSUBPD dest, src1, src2, src3/mem    // dest = −(src1* src2) − src3/mem
```

and three three-operand forms:

```
VFNMSUB132PD src1, src2, src3/mem    // src1 = −(src1* src3/mem) − src2
VFNMSUB213PD src1, src2, src3/mem    // src1 = −(src2* src1) − src3/mem
VFNMSUB231PD src1, src2, src3/mem    // src1 = −(src2* src3/mem) − src1
```

When VEX.L = 0, the vector size is 128 bits (two double-precision elements per vector) and register-based source operands are held in XMM registers.

When VEX.L = 1, the vector size is 256 bits (four double-precision elements per vector) and register-based source operands are held in YMM registers.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a memory location.

For the three-operand forms, VEX.W is 1. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is either an XMM register or a YMM register, as determined by VEX.L. When the destination is an XMM register (L = 0), bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFNMSUBPD	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFNMSUB*nnn*PD	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFNMSUBPD *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.0.01	7D /r /is4
VFNMSUBPD *ymm1*, *ymm2*, *ymm3/mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.1.01	7D /r /is4
VFNMSUBPD *xmm1*, *xmm2, xmm3, xmm4/mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.0.01	7D /r /is4
VFNMSUBPD *ymm1*, *ymm2, ymm3, ymm4/mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.1.01	7D /r /is4
VFNMSUB132PD *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.0.01	9E /r
VFNMSUB132PD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.1.01	9E /r
VFNMSUB213PD *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.0.01	AE /r
VFNMSUB213PD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.1.01	AE /r
VFNMSUB231PD *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.0.01	BE /r
VFNMSUB231PD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src2}}$.1.01	BE /r

Related Instructions

VFNMSUBPS, VFNMSUB132PS, VFNMSUB213PS, VFNMSUB231PS, VFNMSUBSD, VFNMSUB132SD, VFNMSUB213SD, VFNMSUB231SD, VFNMSUBSS, VFNMSUB132SS, VFNMSUB213SS, VFNMSUB231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFNMSUBPS
VFNMSUB132PS
VFNMSUB213PS
VFNMSUB231PS

Negative Multiply and Subtract Packed Single-Precision Floating-Point

Multiplies together two single-precision floating-point vectors, negates the unrounded product, and subtracts a third single-precision floating-point vector from it. The precise result is then rounded to single-precision based on the mode specified by the MXCSR[RC] field and written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the vector equation in the comment on the right.

There are two four-operand forms:

```
VFNMADDPS dest, src1, src2/mem, src3      // dest = −(src1* src2/mem) − src3
VFNMADDPS dest, src1, src2, src3/mem      // dest = −(src1* src2) − src3/mem
```

and three three-operand forms:

```
VFNMADD132PS src1, src2, src3/mem         // src1 = −(src1* src3/mem) − src2
VFNMADD213PS src1, src2, src3/mem         // src1 = −(src2* src1) − src3/mem
VFNMADD231PS src1, src2, src3/mem         // src1 = −(src2* src3/mem) − src1
```

When VEX.L = 0, the vector size is 128 bits (four single-precision elements per vector) and register-based source operands are held in XMM registers.

When VEX.L = 1, the vector size is 256 bits (eight single-precision elements per vector) and register-based source operands are held in YMM registers.

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a memory location.

For the three-operand forms, VEX.W is 0. The first and second operands are registers and the third operand is either a register or a memory location.

The destination is either an XMM register or a YMM register, as determined by VEX.L. When the destination is an XMM register (L = 0), bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFNMSUBPS	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFNMSUB*nnn*PS	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFNMSUBPS *xmm1, xmm2, xmm3/mem128, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.0.01	7C /r /is4
VFNMSUBPS *ymm1*, *ymm2*, *ymm3/mem256*, *ymm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.1.01	7C /r /is4
VFNMSUBPS *xmm1*, *xmm2, xmm3, xmm4/mem128*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.0.01	7C /r /is4
VFNMSUBPS *ymm1*, *ymm2, ymm3, ymm4/mem256*	C4	$\overline{\text{RXB}}$.03	1.$\overline{\textit{src1}}$.1.01	7C /r /is4
VFNMSUB132PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	9E /r
VFNMSUB132PS *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.1.01	9E /r
VFNMSUB213PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	AE /r
VFNMSUB213PS *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.1.01	AE /r
VFNMSUB231PS *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.0.01	BE /r
VFNMSUB231PS *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src2}}$.1.01	BE /r

Related Instructions

VFNMSUBPD, VFNMSUB132PD, VFNMSUB213PD, VFNMSUB231PD, VFNMSUBSD, VFNMSUB132SD, VFNMSUB213SD, VFNMSUB231SD, VFNMSUBSS, VFNMSUB132SS, VFNMSUB213SS, VFNMSUB231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFNMSUBSD VFNMSUB132SD VFNMSUB213SD VFNMSUB231SD — Negative Multiply and Subtract Scalar Double-Precision Floating-Point

Multiplies together two double-precision floating-point values, negates the unrounded product, and subtracts a third double-precision floating-point value from it. The precise result is then rounded to double-precision based on the mode specified by the MXCSR[RC] field and written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the equation in the comment on the right.

There are two four-operand forms:

```
VFNMSUBSD dest, src1, src2/mem, src3    // dest = −(src1* src2/mem) − src3
VFNMSUBSD dest, src1, src2, src3/mem    // dest = −(src1* src2) − src3/mem
```

and three three-operand forms:

```
VFNMSUB132SD src1, src2, src3/mem    // src1 = −(src1* src3/mem) − src2
VFNMSUB213SD src1, src2, src3/mem    // src1 = −(src2* src1) − src3/mem
VFNMSUB231SD src1, src2, src3/mem    // src1 = −(src2* src3/mem) − src1
```

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a 64-bit memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a 64-bit memory location.

For the three-operand forms, VEX.W is 1. The first and second operands are registers and the third operand is either a register or a 64-bit memory location.

The destination is an XMM register. Bits [127:64] of the destination XMM register and bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFNMSUBSD	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFNMSUB*nnn*SD	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFNMSUBSD *xmm1, xmm2, xmm3/mem64, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.X.01	7F /r /is4
VFNMSUBSD *xmm1*, *xmm2, xmm3, xmm4/mem64*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.X.01	7F /r /is4
VFNMSUB132SD *xmm1, xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.X.01	9F /r
VFNMSUB213SD *xmm1, xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.X.01	AF /r
VFNMSUB231SD *xmm1, xmm2, xmm3/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.X.01	BF /r

Related Instructions

VFNMSUBPD, VFNMSUB132PD, VFNMSUB213PD, VFNMSUB231PD, VFNMSUBPS, VFNMSUB132PS, VFNMSUB213PS, VFNMSUB231PS, VFNMSUBSS, VFNMSUB132SS, VFNMSUB213SS, VFNMSUB231SS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Non-aligned memory reference when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFNMSUBSS VFNMSUB132SS VFNMSUB213SS VFNMSUB231SS — Negative Multiply and Subtract Scalar Single-Precision Floating-Point

Multiplies together two single-precision floating-point values, negates the unrounded product, and subtracts a third single-precision floating-point value from it. The precise result is then rounded to single-precision based on the mode specified by the MXCSR[RC] field and written to the destination register. The role of each of the source operands specified by the assembly language prototypes given below is reflected in the equation in the comment on the right.

There are two four-operand forms:

```
VFNMSUBSS dest, src1, src2/mem, src3    // dest = −(src1* src2/mem) − src3
VFNMSUBSS dest, src1, src2, src3/mem    // dest = −(src1* src2) − src3/mem
```

and three three-operand forms:

```
VFNMSUB132SS src1, src2, src3/mem    // src1 = −(src1* src3/mem) − src2
VFNMSUB213SS src1, src2, src3/mem    // src1 = −(src2* src1) − src3/mem
VFNMSUB231SS src1, src2, src3/mem    // src1 = −(src2* src3/mem) − src1
```

For the four-operand forms, VEX.W determines operand configuration.

- When VEX.W = 0, the second source is either a register or a 32-bit memory location and the third source is a register.
- When VEX.W = 1, the second source is a register and the third source is either a register or a 32-bit memory location.

For the three-operand forms, VEX.W is 0. The first and second operands are registers and the third operand is either a register or a 32-bit memory location.

The destination is an XMM register. Bits[127:32] of the destination XMM register and bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VFNMSUBSS	FMA4	CPUID Fn8000_0001_ECX[FMA4] (bit 16)
VFNMSUB*nnn*SS	FMA	CPUID Fn0000_0001_ECX[FMA] (bit 12)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VFNMSUBSS *xmm1, xmm2, xmm3/mem32, xmm4*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.X.01	7E /r /is4
VFNMSUBSS *xmm1*, *xmm2*, *xmm3, xmm4/mem32*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.X.01	7E /r /is4
VFNMSUB132SS *xmm1, xmm2, xmm3/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.X.01	9F /r
VFNMSUB213SS *xmm1, xmm2, xmm3/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.X.01	AF /r
VFNMSUB231SS *xmm1, xmm2, xmm3/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.X.01	BF /r

Related Instructions

VFNMSUBPD, VFNMSUB132PD, VFNMSUB213PD, VFNMSUB231PD, VFNMSUBPS, VFNMSUB132PS, VFNMSUB213PS, VFNMSUB231PS, VFNMSUBSD, VFNMSUB132SD, VFNMSUB213SD, VFNMSUB231SD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Non-aligned memory reference when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

VFRCZPD Extract Fraction Packed Double-Precision Floating-Point

Extracts the fractional portion of each double-precision floating-point value of either a source register or a memory location and writes the resulting values to the corresponding elements of the destination. The fractional results are precise.

- When XOP.L = 0, the source is either an XMM register or a 128-bit memory location.
- When XOP.L = 1, the source is a YMM register or 256-bit memory location.

When the destination is an XMM register, bits [255:128] of the corresponding YMM register are cleared.

Exception conditions are the same as for other arithmetic instructions, except with respect to the sign of a zero result. A zero is returned in the following cases:

- When the operand is a zero.
- When the operand is a normal integer.
- When the operand is a denormal value and is coerced to zero by MXCSR.DAZ.
- When the operand is a denormal value that is not coerced to zero by MXCSR.DAZ.

In the first three cases, when MXCSR.RC = 01b (round toward – ∞) the sign of the zero result is negative, and is otherwise positive.

In the fourth case, the operand is its own fractional part, which results in underflow, and the result is forced to zero by MXCSR.FZ; the result has the same sign as the operand.

Instruction Support

Form	Subset	Feature Flag
VFRCZPD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VFRCZPD *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	81 /r
VFRCZPD *ymm1*, *ymm2/mem256*	8F	$\overline{\text{RXB}}$.09	0.1111.1.00	81 /r

Related Instructions

(V)ROUNDPD, (V)ROUNDPS, (V)ROUNDSD, (V)ROUNDSS, VFRCZPS, VFRCZSS, VFRCZSD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M			M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.vvvv ! = 1111b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0. See *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			X	A source operand was an SNaN value.
			X	Undefined operation.
Denormalized operand, DE			X	A source operand was a denormal value.
Underflow, UE			X	Rounded result too small to fit into the format of the destination operand.
Precision, PE			X	A result could not be represented exactly in the destination format.
X — XOP exception				

VFRCZPS Extract Fraction Packed Single-Precision Floating-Point

Extracts the fractional portion of each single-precision floating-point value of either a source register or a memory location and writes the resulting values to the corresponding elements of the destination. The fractional results are exact.

- When XOP.L = 0, the source is either an XMM register or a 128-bit memory location.
- When XOP.L = 1, the source is a YMM register or 256-bit memory location.

When the destination is an XMM register, bits [255:128] of the corresponding YMM register are cleared.

Exception conditions are the same as for other arithmetic instructions, except with respect to the sign of a zero result. A zero is returned in the following cases:

- When the operand is a zero.
- When the operand is a normal integer.
- When the operand is a denormal value and is coerced to zero by MXCSR.DAZ.
- When the operand is a denormal value that is not coerced to zero by MXCSR.DAZ.

In the first three cases, when MXCSR.RC = 01b (round toward – ∞) the sign of the zero result is negative, and is otherwise positive.

In the fourth case, the operand is its own fractional part, which results in underflow, and the result is forced to zero by MXCSR.FZ; the result has the same sign as the operand.

Instruction Support

Form	Subset	Feature Flag
VFRCZPS	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VFRCZPS *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	80 /r
VFRCZPS *ymm1*, *ymm2/mem256*	8F	$\overline{\text{RXB}}$.09	0.1111.1.00	80 /r

Related Instructions

(V)ROUNDPD, (V)ROUNDPS, (V)ROUNDSD, (V)ROUNDSS, VFRCZPD, VFRCZSS, VFRCZSD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M			M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.vvvv ! = 1111b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0. See *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			X	A source operand was an SNaN value.
			X	Undefined operation.
Denormalized operand, DE			X	A source operand was a denormal value.
Underflow, UE			X	Rounded result too small to fit into the format of the destination operand.
Precision, PE			X	A result could not be represented exactly in the destination format.
X — XOP exception				

VFRCZSD Extract Fraction Scalar Double-Precision Floating-Point

Extracts the fractional portion of the double-precision floating-point value of either the low-order quadword of an XMM register or a 64-bit memory location and writes the result to the low-order quadword of the destination XMM register. The fractional results are precise.

When the result is written to the destination XMM register, bits [127:64] of the destination and bits [255:128] of the corresponding YMM register are cleared.

Exception conditions are the same as for other arithmetic instructions, except with respect to the sign of a zero result. A zero is returned in the following cases:

- When the operand is a zero.
- When the operand is a normal integer.
- When the operand is a denormal value and is coerced to zero by MXCSR.DAZ.
- When the operand is a denormal value that is not coerced to zero by MXCSR.DAZ.

In the first three cases, when MXCSR.RC = 01b (round toward – ∞) the sign of the zero result is negative, and is otherwise positive.

In the fourth case, the operand is its own fractional part, which results in underflow, and the result is forced to zero by MXCSR.FZ; the result has the same sign as the operand.

Instruction Support

Form	Subset	Feature Flag
VFRCZSD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VFRCZSD *xmm1*, *xmm2/mem64*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	83 /r

Related Instructions

(V)ROUNDPD, (V)ROUNDPS, (V)ROUNDSD, (V)ROUNDSS, VFRCZPS, VFRCZPD, VFRCZSS

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M			M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.vvvv ! = 1111b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0. See *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			X	A source operand was an SNaN value.
			X	Undefined operation.
Denormalized operand, DE			X	A source operand was a denormal value.
Underflow, UE			X	Rounded result too small to fit into the format of the destination operand.
Precision, PE			X	A result could not be represented exactly in the destination format.
X — XOP exception				

VFRCZSS Extract Fraction Scalar Single-Precision Floating Point

Extracts the fractional portion of the single-precision floating-point value of the low-order double-word of an XMM register or 32-bit memory location and writes the result in the low-order double-word of the destination XMM register. The fractional results are precise.

When the result is written to the destination XMM register, bits [127:32] of the destination and bits [255:128] of the corresponding YMM register are cleared.

Exception conditions are the same as for other arithmetic instructions, except with respect to the sign of a zero result. A zero is returned in the following cases:

- When the operand is a zero.
- When the operand is a normal integer.
- When the operand is a denormal value and is coerced to zero by MXCSR.DAZ.
- When the operand is a denormal value that is not coerced to zero by MXCSR.DAZ.

In the first three cases, when MXCSR.RC = 01b (round toward – ∞) the sign of the zero result is negative, and is otherwise positive.

In the fourth case, the operand is its own fractional part, which results in underflow, and the result is forced to zero by MXCSR.FZ; the result has the same sign as the operand.

Instruction Support

Form	Subset	Feature Flag
VFRCZSS	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VFRCZSS *xmm1*, *xmm2/mem32*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	82 /r

Related Instructions

ROUNDPD, ROUNDPS, ROUNDSD, ROUNDSS, VFRCZPS, VFRCZPD, VFRCZSD

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M			M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that may be set or cleared is M (modified). Unaffected flags are blank.*																

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.vvvv ! = 1111b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0. See *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			X	A source operand was an SNaN value.
			X	Undefined operation.
Denormalized operand, DE			X	A source operand was a denormal value.
Underflow, UE			X	Rounded result too small to fit into the format of the destination operand.
Precision, PE			X	A result could not be represented exactly in the destination format.
X — XOP exception				

VGATHERDPD Conditionally Gather Double-Precision Floating-Point Values, Doubleword Indices

Conditionally loads double-precision (64-bit) values from memory using VSIB addressing with doubleword indices.

The instruction is of the form:

VGATHERDPD *dest, mem64[vm32x], mask*

Loading of each element of the destination register is conditional based on the value of the corresponding element of the mask operand. If the most-significant bit of the *i*th element of the mask is set, the *i*th element of the destination is loaded from memory using the *i*th address of the array of effective addresses calculated using VSIB addressing.

The index register is treated as an array of signed 32-bit values. Quadword elements of the destination for which the corresponding mask element is zero are not affected by the operation. If no exceptions occur, the mask register is set to zero.

Execution of the instruction can be suspended by an exception if the exception is triggered by an element other than the rightmost element loaded. When this happens, the destination register and the mask operand may be observed as partially updated. Elements that have been loaded will have their mask elements set to zero. If any traps or faults are pending from elements that have been loaded, they will be delivered in lieu of the exception; in this case, the RF flag is set so that an instruction breakpoint is not re-triggered when the instruction execution is resumed.

See Section 1.3, “VSIB Addressing,” on page 6 for a discussion of the VSIB addressing mode.

There are 128-bit and 256-bit forms of this instruction.

XMM Encoding

The destination is an XMM register. The first source operand is up to two 64-bit values located in memory. The second source operand (the mask) is an XMM register. The index vector is the two low-order doublewords of an XMM register; the two high-order doublewords of the index register are not used. Bits [255:128] of the YMM register that corresponds to the destination and bits [255:128] of the YMM register that corresponds to the second source (mask) operand are cleared.

YMM Encoding

The destination is a YMM register. The first source operand is up to four 64-bit values located in memory. The second source operand (the mask) is a YMM register. The index vector is the four doublewords of an XMM register.

Instruction Support

Form	Subset	Feature Flag
VGATHERDPD	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VGATHERDPD *xmm1, vm32x, xmm2*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	92 /r
VGATHERDPD *ymm1, vm32x, ymm2*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	92 /r

Related Instructions

VGATHERDPS, VGATHERQPD, VGATHERQPS, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ

rFLAGS Affected

RF

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
			A	MODRM.mod = 11b
			A	MODRM.rm ! = 100b
			A	YMM/XMM registers specified for destination, mask, and index not unique.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

VGATHERDPS Conditionally Gather Single-Precision Floating-Point Values, Doubleword Indices

Conditionally loads single-precision (32-bit) values from memory using VSIB addressing with doubleword indices.

The instruction is of the form:

VGATHERDPS *dest, mem32[vm32x/y], mask*

Loading of each element of the destination register is conditional based on the value of the corresponding element of the mask operand. If the most-significant bit of the *i*th element of the mask is set, the *i*th element of the destination is loaded from memory using the *i*th address of the array of effective addresses calculated using VSIB addressing.

The index register is treated as an array of signed 32-bit values. Doubleword elements of the destination for which the corresponding mask element is zero are not affected by the operation. If no exceptions occur, the mask register is set to zero.

Execution of the instruction can be suspended by an exception if the exception is triggered by an element other than the rightmost element loaded. When this happens, the destination register and the mask operand may be observed as partially updated. Elements that have been loaded will have their mask elements set to zero. If any traps or faults are pending from elements that have been loaded, they will be delivered in lieu of the exception; in this case, the RF flag is set so that an instruction breakpoint is not re-triggered when the instruction execution is resumed.

See Section 1.3, “VSIB Addressing,” on page 6 for a discussion of the VSIB addressing mode.

There are 128-bit and 256-bit forms of this instruction.

XMM Encoding

The destination is an XMM register. The first source operand is up to four 32-bit values located in memory. The second source operand (the mask) is an XMM register. The index vector is the four doublewords of an XMM register. Bits [255:128] of the YMM register that corresponds to the destination and bits [255:128] of the YMM register that corresponds to the second source (mask) operand are cleared.

YMM Encoding

The destination is a YMM register. The first source operand is up to eight 32-bit values located in memory. The second source operand (the mask) is a YMM register. The index vector is the eight doublewords of a YMM register.

Instruction Support

Form	Subset	Feature Flag
VGATHERDPS	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VGATHERDPS *xmm1, vm32x, xmm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.0.01	92 /r
VGATHERDPS *ymm1, vm32y, ymm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.1.01	92 /r

Related Instructions

VGATHERDPD, VGATHERQPD, VGATHERQPS, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ

rFLAGS Affected

RF

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
			A	MODRM.mod = 11b
			A	MODRM.rm ! = 100b
			A	YMM/XMM registers specified for destination, mask, and index not unique.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

VGATHERQPD Conditionally Gather Double-Precision Floating-Point Values, Quadword Indices

Conditionally loads double-precision (64-bit) values from memory using VSIB addressing with quadword indices.

The instruction is of the form:

VGATHERQPD *dest, mem64[vm64x/y], mask*

Loading of each element of the destination register is conditional based on the value of the corresponding element of the mask operand. If the most-significant bit of the *i*th element of the mask is set, the *i*th element of the destination is loaded from memory using the *i*th address of the array of effective addresses calculated using VSIB addressing.

The index register is treated as an array of signed 64-bit values. Quadword elements of the destination for which the corresponding mask element is zero are not affected by the operation. If no exceptions occur, the mask register is set to zero.

Execution of the instruction can be suspended by an exception if the exception is triggered by an element other than the rightmost element loaded. When this happens, the destination register and the mask operand may be observed as partially updated. Elements that have been loaded will have their mask elements set to zero. If any traps or faults are pending from elements that have been loaded, they will be delivered in lieu of the exception; in this case, the RF flag is set so that an instruction breakpoint is not re-triggered when the instruction execution is resumed.

See Section 1.3, “VSIB Addressing,” on page 6 for a discussion of the VSIB addressing mode.

There are 128-bit and 256-bit forms of this instruction.

XMM Encoding

The destination is an XMM register. The first source operand is up to two 64-bit values located in memory. The second source operand (the mask) is an XMM register. The index vector is the two quadwords of an XMM register. Bits [255:128] of the YMM register that corresponds to the destination and bits [255:128] of the YMM register that corresponds to the second source (mask) operand are cleared.

YMM Encoding

The destination is a YMM register. The first source operand is up to four 64-bit values located in memory. The second source operand (the mask) is a YMM register. The index vector is the four quadwords of a YMM register.

Instruction Support

Form	Subset	Feature Flag
VGATHERQPD	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VGATHERQPD *xmm1, vm64x, xmm2*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	93 /r
VGATHERQPD *ymm1, vm64y, ymm2*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	93 /r

Related Instructions

VGATHERDPD, VGATHERDPS, VGATHERQPS, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ

rFLAGS Affected

RF

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
			A	MODRM.mod = 11b
			A	MODRM.rm ! = 100b
			A	YMM/XMM registers specified for destination, mask, and index not unique.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

VGATHERQPS Conditionally Gather Single-Precision Floating-Point Values, Quadword Indices

Conditionally loads single-precision (32-bit) values from memory using VSIB addressing with quadword indices.

The instruction is of the form:

VGATHERQPS *dest, mem32[vm64x/y], mask*

Loading of each element of the destination register is conditional based on the value of the corresponding element of the mask operand. If the most-significant bit of the *i*th element of the mask is set, the *i*th element of the destination is loaded from memory using the *i*th address of the array of effective addresses calculated using VSIB addressing.

The index register is treated as an array of signed 64-bit values. Doubleword elements of the destination for which the corresponding mask element is zero are not affected by the operation. The upper half of the destination is zeroed. If no exceptions occur, the mask register is set to zero.

Execution of the instruction can be suspended by an exception if the exception is triggered by an element other than the rightmost element loaded. When this happens, the destination register and the mask operand may be observed as partially updated. Elements that have been loaded will have their mask elements set to zero. If any traps or faults are pending from elements that have been loaded, they will be delivered in lieu of the exception; in this case, the RF flag is set so that an instruction breakpoint is not re-triggered when the instruction execution is resumed.

See Section 1.3, “VSIB Addressing,” on page 6 for a discussion of the VSIB addressing mode.

There are 128-bit and 256-bit forms of this instruction.

XMM Encoding

The destination is an XMM register. The first source operand is up to two 32-bit values located in memory. The second source operand (the mask) is an XMM register. Only the lower half of the mask is used. The index vector is the two quadwords of an XMM register. Bits [255:64] of the YMM register that corresponds to the destination and bits [255:64] of the YMM register that corresponds to the second source (mask) operand are cleared.

YMM Encoding

The destination is an XMM register. The first source operand is up to four 32-bit values located in memory. The second source operand (the mask) is an XMM register. The index vector is the four quadwords of a YMM register. Bits [255:128] of the YMM register that corresponds to the destination and bits [255:128] of the YMM register that corresponds to the second source (mask) operand are cleared.

Instruction Support

Form	Subset	Feature Flag
VGATHERQPS	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VGATHERQPS *xmm1, vm64x, xmm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.0.01	93 /r
VGATHERQPS *xmm1, vm64y, xmm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.1.01	93 /r

Related Instructions

VGATHERDPD, VGATHERDPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ

rFLAGS Affected

RF

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
			A	MODRM.mod = 11b
			A	MODRM.rm ! = 100b
			A	YMM/XMM registers specified for destination, mask, and index not unique.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

VINSERTF128 Insert Packed Floating-Point Values 128-bit

Combines 128 bits of data from a YMM register with 128-bit packed-value data from an XMM register or a 128-bit memory location, as specified by an immediate byte operand, and writes the combined data to the destination.

Only bit [0] of the immediate operand is used. Operation is as follows.

- When imm8[0] = 0, copy bits [255:128] of the first source to bits [255:128] of the destination and copy bits [127:0] of the second source to bits [127:0] of the destination.
- When imm8[0] = 1, copy bits [127:0] of the first source to bits [127:0] of the destination and copy bits [127:0] of the second source to bits [255:128] of the destination.

This extended-form instruction has a single 256-bit encoding.

The first source operand is a YMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a YMM register. There is a third immediate byte operand.

Instruction Support

Form	Subset	Feature Flag
VINSERTF128	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VINSERTF128 *ymm1, ymm2, xmm3/mem128, imm8*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src}$.1.01	18 /r ib

Related Instructions

VBROADCASTF128, VBROADCASTI128, VEXTRACTF128, VEXTRACTI128, VINSERTI128

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.L = 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
A — AVX exception.				

VINSERTI128 Insert Packed Integer Values 128-bit

Combines 128 bits of data from a YMM register with 128-bit packed-value data from an XMM register or a 128-bit memory location, as specified by an immediate byte operand, and writes the combined data to the destination.

Bit [0] of the immediate operand controls how the 128-bit values from the source operands are merged into the destination. The operation is as follows.

- When imm8[0] = 0, copy bits [255:128] of the first source to bits [255:128] of the destination and copy bits [127:0] of the second source to bits [127:0] of the destination.
- When imm8[0] = 1, copy bits [127:0] of the first source to bits [127:0] of the destination and copy bits [127:0] of the second source to bits [255:128] of the destination.

This instruction has a single 256-bit encoding.

The first source operand is a YMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a YMM register. The immediate byte is encoded in the instruction.

Instruction Support

Form	Subset	Feature Flag
VINSERTI128	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VINSERTI128 *ymm1, ymm2, xmm3/mem128, imm8*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.1.01	38 /r ib

Related Instructions

VBROADCASTF128, VBROADCASTI128, VEXTRACTF128, VEXTRACTI128, VINSERTF128

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.L = 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
A — AVX exception.				

VMASKMOVPD Masked Move Packed Double-Precision

Moves packed double-precision data elements from a source element to a destination element, as specified by mask bits in a source operand. There are load and store versions of the instruction.

For loads, the data elements are in a source memory location; for stores the data elements are in a source register. The mask bits are the most-significant bit of the corresponding data element of a source register.

- For loads, when a mask bit = 1, the corresponding data element is copied from the source to the same element of the destination; when a mask bit = 0, the corresponding element of the destination is cleared.
- For stores, when a mask bit = 1, the corresponding data element is copied from the source to the same element of the destination; when a mask bit = 0, the corresponding element of the destination is not affected.

Exception and trap behavior for elements not selected for loading or storing from/to memory is implementation dependent. For instance, a given implementation may signal a data breakpoint or a page fault for quadwords that are zero-masked and not actually written.

XMM Encoding

There are load and store encodings.

- For loads, there are two 64-bit source data elements in a 128-bit memory location, the mask operand is an XMM register, and the destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.
- For stores, there are two 64-bit source data elements in an XMM register, the mask operand is an XMM register, and the destination is a 128-bit memory location.

YMM Encoding

There are load and store encodings.

- For loads, there are four 64-bit source data elements in a 256-bit memory location, the mask operand is a YMM register, and the destination is a YMM register.
- For stores, there are four 64-bit source data elements in a YMM register, the mask operand is a YMM register, and the destination is a 128-bit memory location.

Instruction Support

Form	Subset	Feature Flag
VMASKMOVPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
Loads:				
VMASKMOVPD *xmm1*, *xmm2*, *mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.0.01	2D /r
VMASKMOVPD *ymm1*, *ymm2*, *mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.1.01	2D /r
Stores:				
VMASKMOVPD *mem128, xmm1*, *xmm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.0.01	2F /r
VMASKMOVPD *mem256, ymm1*, *ymm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.1.01	2F /r

Related Instructions

VMASKMOVPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
	S	S	X	Write to a read-only data segment.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX exception.				

VMASKMOVPS Masked Move Packed Single-Precision

Moves packed single-precision data elements from a source element to a destination element, as specified by mask bits in a source operand. There are load and store versions of the instruction.

For loads, the data elements are in a source memory location; for stores the data elements are in a source register. The mask bits are the most-significant bits of the corresponding data element of a source register.

- For loads, when a mask bit = 1, the corresponding data element is copied from the source to the same element of the destination; when a mask bit = 0, the corresponding element of the destination is cleared.
- For stores, when a mask bit = 1, the corresponding data element is copied from the source to the same element of the destination; when a mask bit = 0, the corresponding element of the destination is not affected.

Exception and trap behavior for elements not selected for loading or storing from/to memory is implementation dependent. For instance, a given implementation may signal a data breakpoint or a page fault for doublewords that are zero-masked and not actually written.

XMM Encoding

There are load and store encodings.

- For loads, there are four 32-bit source data elements in a 128-bit memory location, the mask operand is an XMM register, and the destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.
- For stores, there are four 32-bit source data elements in an XMM register, the mask operand is an XMM register, and the destination is a 128-bit memory location.

YMM Encoding

There are load and store encodings.

- For loads, there are eight 32-bit source data elements in a 256-bit memory location, the mask operand is a YMM register, and the destination is a YMM register.
- For stores, there are eight 32-bit source data elements in a YMM register, the mask operand is a YMM register, and the destination is a 128-bit memory location.

Instruction Support

Form	Subset	Feature Flag
VMASKMOVPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
Loads:				
VMASKMOVPS *xmm1*, *xmm2*, *mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.0.01	2C /r
VMASKMOVPS *ymm1*, *ymm2*, *mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.1.01	2C /r
Stores:				
VMASKMOVPS *mem128, xmm1*, *xmm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.0.01	2E /r
VMASKMOVPS *mem256, ymm1*, *ymm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.1.01	2E /r

Related Instructions

VMASKMOVPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
	S	S	X	Write to a read-only data segment.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX exception.				

VPBLENDD — Blend Packed Doublewords

Copies packed doublewords from either of two sources to a destination, as specified by an immediate 8-bit mask operand.

Each bit of the mask selects a doubleword from one of the source operands to be copied to the destination. The least-significant bit controls the selection of the doubleword to be copied to the lowest doubleword of the destination. For each doubleword *i* of the destination:

- When mask bit [*i*] = 0, doubleword *i* of the first source operand is copied to the corresponding doubleword of the destination.
- When mask bit [*i*] = 1, doubleword *i* of the second source operand is copied to the corresponding doubleword of the destination.

VPBLENDD

The instruction has 128-bit and 256-bit encodings.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
VPBLENDD	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPBLENDD *xmm1, xmm2, xmm3/mem128, imm8*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.0.01	02 /r /ib
VPBLENDD *ymm1, ymm2, ymm3/mem256, imm8*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.1.01	02 /r /ib

Related Instructions

VBLENDW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX2 exception				

VPBROADCASTB Broadcast Packed Byte

Loads a byte from a register or memory and writes it to all 16 or 32 bytes of an XMM or YMM register.

This instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Copies the source operand to all 16 bytes of the destination.

The source operand is the least-significant 8 bits of an XMM register or an 8-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Copies the source operand to all 32 bytes of the destination.

The source operand is the least-significant 8 bits of an XMM register or an 8-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VPBROADCASTB	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPBROADCASTB *xmm1*, *xmm2/mem8*	C4	$\overline{\text{RXB}}$.02	0.1111.0.01	78 /r
VPBROADCASTB *ymm1*, *xmm2/mem8*	C4	$\overline{\text{RXB}}$.02	0.1111.1.01	78 /r

Related Instructions

VPBROADCASTD, VPBROADCASTQ, VPBROADCASTW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

VPBROADCASTD Broadcast Packed Doubleword

Loads a doubleword from a register or memory and writes it to all 4 or 8 doublewords of an XMM or YMM register.

This instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Copies the source operand to all 4 doublewords of the destination.

The source operand is the least-significant 32 bits of an XMM register or a 32-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Copies the source operand to all 8 doublewords of the destination.

The source operand is the least-significant 32 bits of an XMM register or a 32-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VPBROADCASTD	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPBROADCASTD *xmm1*, *xmm2/mem32*	C4	$\overline{\text{RXB}}$.02	0.1111.0.01	58 /r
VPBROADCASTD *ymm1*, *xmm2/mem32*	C4	$\overline{\text{RXB}}$.02	0.1111.1.01	58 /r

Related Instructions

VPBROADCASTB, VPBROADCASTQ, VPBROADCASTW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

VPBROADCASTQ Broadcast Packed Quadword

Loads a quadword from a register or memory and writes it to all 2 or 4 quadwords of an XMM or YMM register.

This instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Copies the source operand to both quadwords of the destination.

The source operand is the least-significant 64 bits of an XMM register or a 64-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Copies the source operand to all 4 quadwords of the destination.

The source operand is the least-significant 64 bits of an XMM register or a 64-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VPBROADCASTQ	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPBROADCASTQ *xmm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	0.1111.0.01	59 /r
VPBROADCASTQ *ymm1*, *xmm2/mem64*	C4	$\overline{\text{RXB}}$.02	0.1111.1.01	59 /r

Related Instructions

VPBROADCASTB, VPBROADCASTD, VPBROADCASTW

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

VPBROADCASTW Broadcast Packed Word

Loads a word from a register or memory and writes it to all 8 or 16 words of an XMM or YMM register.

This instruction has both 128-bit and 256-bit encodings:

XMM Encoding

Copies the source operand to all 8 words of the destination.

The source operand is the least-significant 16 bits of an XMM register or a 16-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

Copies the source operand to all 16 words of the destination.

The source operand is the least-significant 16 bits of an XMM register or a 16-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VPBROADCASTW	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPBROADCASTW *xmm1*, *xmm2/mem16*	C4	$\overline{\text{RXB}}$.02	0.1111.0.01	79 /r
VPBROADCASTW *ymm1*, *xmm2/mem16*	C4	$\overline{\text{RXB}}$.02	0.1111.1.01	79 /r

Related Instructions

VPBROADCASTB, VPBROADCASTD, VPBROADCASTQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

VPCMOV Vector Conditional Move

Moves bits of either the first source or the second source to the corresponding positions in the destination, depending on the value of the corresponding bit of a third source.

When a bit of the third source = 1, the corresponding bit of the first source is moved to the destination; when a bit of the third source = 0, the corresponding bit of the second source is moved to the destination.

This instruction directly implements the C-language ternary "?" operation on each source bit.

Arbitrary bit-granular predicates can be constructed by any number of methods, or loaded as constants from memory. This instruction may use the results of any SSE instructions as the predicate in the selector. VPCMPEQB (VPCMPGTB), VPCMPEQW (VPCMPGTW), VPCMPEQD (VPCMPGTD) and VPCMPEQQ (VPCMPGTQ) compare bytes, words, doublewords, quadwords and integers, respectively, and set the predicate in the destination to masks of 1s and 0s accordingly. VCMPPS (VCMPSS) and VCMPPD (VCMPSD) compare word and doubleword floating-point source values, respectively, and provide the predicate for the floating-point instructions.

There are four operands: VPCMOV *dest, src1, src2, src3.*

The first source (*src1*) is an XMM or YMM register specified by XOP.vvvv.

XOP.W and bits [7:4] of an immediate byte (*imm8*) configure *src2* and *src3*:

- When XOP.W = 0, *src2* is either a register or a memory location specified by ModRM.r/m and *src3* is a register specified by *imm8[7:4]*.
- When XOP.W = 1, s*rc2* is a register specified by *imm8[7:4]* and *src3* is either a register or a memory location specified by ModRM.r/m.

The destination (*dest*) is either an XMM or a YMM register, as determined by XOP.L. When the destination is an XMM register, bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPCMOV	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPCMOV *xmm1*, *xmm2, xmm3*/*mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	A2 /r ib
VPCMOV *ymm1*, *ymm2, ymm3*/*mem256, ymm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.1.00	A2 /r ib
VPCMOV *xmm1*, *xmm2, xmm3, xmm4*/*mem128*	8F	$\overline{\text{RXB}}$.08	1.$\overline{src1}$.0.00	A2 /r ib
VPCMOV *ymm1*, *ymm2, ymm3*, *ymm4*/*mem256*	8F	$\overline{\text{RXB}}$.08	1.$\overline{src1}$.1.00	A2 /r ib

Related Instructions

VPCOMUB, VPCOMUD, VPCOMUQ, VPCOMUW, VCMPPD, VCMPPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPCOMB — Compare Vector Signed Bytes

Compares corresponding packed signed bytes in the first and second sources and writes the result of each comparison in the corresponding byte of the destination. The result of each comparison is an 8-bit value of all 1s (TRUE) or all 0s (FALSE).

There are four operands: VPCOMB *dest*, *src1*, *src2*, *imm8*

The destination (*dest*) is an XMM registers specified by ModRM.reg. When the comparison results are written to the destination XMM register, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by the XOP.vvvv field and the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field.

The comparison type is specified by bits [2:0] of the immediate-byte operand (*imm8*). Each type has an alias mnemonic to facilitate coding.

imm8[2:0]	Comparison	Mnemonic
000	Less Than	VPCOMLTB
001	Less Than or Equal	VPCOMLEB
010	Greater Than	VPCOMGTB
011	Greater Than or Equal	VPCOMGEB
100	Equal	VPCOMEQB
101	Not Equal	VPCOMNEQB
110	False	VPCOMFALSEB
111	True	VPCOMTRUEB

Instruction Support

Form	Subset	Feature Flag
VPCOMB	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPCOMB *xmm1, xmm2, xmm3/mem128, imm8*	8F	$\overline{\text{RXB}}$.08	0.$\overline{\textit{src1}}$.0.00	CC /r ib

Related Instructions

VPCOMUB, VPCOMUW, VPCOMUD, VPCOMUQ, VPCOMW, VPCOMD, VPCOMQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPCOMD — Compare Vector Signed Doublewords

Compares corresponding packed signed doublewords in the first and second sources and writes the result of each comparison to the corresponding doubleword of the destination. The result of each comparison is a 32-bit value of all 1s (TRUE) or all 0s (FALSE).

There are four operands: VPCOMD *dest, src1, src2, imm8*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the results of the comparisons are written to the destination XMM register, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by the XOP.vvvv field and the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field.

The comparison type is specified by bits [2:0] of an immediate-byte operand (*imm8*). Each type has an alias mnemonic to facilitate coding.

imm8[2:0]	Comparison	Mnemonic
000	Less Than	VPCOMLTD
001	Less Than or Equal	VPCOMLED
010	Greater Than	VPCOMGTD
011	Greater Than or Equal	VPCOMGED
100	Equal	VPCOMEQD
101	Not Equal	VPCOMNEQD
110	False	VPCOMFALSED
111	True	VPCOMTRUED

Instruction Support

Form	Subset	Feature Flag
VPCOMD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPCOMD xmm1, xmm2, xmm3/mem128, imm8	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	CE /r ib

Related Instructions

VPCOMUB, VPCOMUW, VPCOMUD, VPCOMUQ, VPCOMB, VPCOMW, VPCOMQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPCOMQ Compare Vector Signed Quadwords

Compares corresponding packed signed quadwords in the first and second sources and writes the result of each comparison to the corresponding quadword of the destination. The result of each comparison is a 64-bit value of all 1s (TRUE) or all 0s (FALSE).

There are four operands: VPCOMQ *dest, src1, src2, imm8*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the result is written to the destination XMM register, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by the XOP.vvvv field and the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field.

The comparison type is specified by bits [2:0] of an immediate-byte operand (*imm8)*. Each type has an alias mnemonic to facilitate coding.

imm8[2:0]	Comparison	Mnemonic
000	Less Than	VPCOMLTQ
001	Less Than or Equal	VPCOMLEQ
010	Greater Than	VPCOMGTQ
011	Greater Than or Equal	VPCOMGEQ
100	Equal	VPCOMEQQ
101	Not Equal	VPCOMNEQQ
110	False	VPCOMFALSEQ
111	True	VPCOMTRUEQ

Instruction Support

Form	Subset	Feature Flag
VPCOMQ	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPCOMQ *xmm1*, *xmm2, xmm3/mem128, imm8*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	CF /r ib

Related Instructions

VPCOMUB, VPCOMUW, VPCOMUD, VPCOMUQ, VPCOMB, VPCOMW, VPCOMD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPCOMUB Compare Vector Unsigned Bytes

Compares corresponding packed unsigned bytes in the first and second sources and writes the result of each comparison to the corresponding byte of the destination. The result of each comparison is an 8-bit value of all 1s (TRUE) or all 0s (FALSE).

There are four operands: VPCOMUB *dest, src1, src2, imm8*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the result is written to the destination XMM register, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by the XOP.vvvv field and the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field.

The comparison type is specified by bits [2:0] of an immediate-byte operand (*imm8)*. Each type has an alias mnemonic to facilitate coding.

imm8[2:0]	Comparison	Mnemonic
000	Less Than	VPCOMLTUB
001	Less Than or Equal	VPCOMLEUB
010	Greater Than	VPCOMGTUB
011	Greater Than or Equal	VPCOMGEUB
100	Equal	VPCOMEQUB
101	Not Equal	VPCOMNEQUB
110	False	VPCOMFALSEUB
111	True	VPCOMTRUEUB

Instruction Support

Form	Subset	Feature Flag
VPCOMUB	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPCOMUB *xmm1*, *xmm2, xmm3/mem128, imm8*	8F	$\overline{\text{RXB}}$.08	0.$\overline{\textit{src1}}$.0.00	EC /r ib

Related Instructions

VPCOMUW, VPCOMUD, VPCOMUQ, VPCOMB, VPCOMW, VPCOMD, VPCOMQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPCOMUD Compare Vector Unsigned Doublewords

Compares corresponding packed unsigned doublewords in the first and second sources and writes the result of each comparison to the corresponding doubleword of the destination. The result of each comparison is a 32-bit value of all 1s (TRUE) or all 0s (FALSE).

There are four operands: VPCOMUD *dest, src1, src2, imm8*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the results are written to the destination XMM register, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by the XOP.vvvv field and the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field.

The comparison type is specified by bits [2:0] of an immediate-byte operand (*imm8)*. Each type has an alias mnemonic to facilitate coding.

imm8[2:0]	Comparison	Mnemonic
000	Less Than	VPCOMLTUD
001	Less Than or Equal	VPCOMLEUD
010	Greater Than	VPCOMGTUD
011	Greater Than or Equal	VPCOMGEUD
100	Equal	VPCOMEQUD
101	Not Equal	VPCOMNEQUD
110	False	VPCOMFALSEUD
111	True	VPCOMTRUEUD

Instruction Support

Form	Subset	Feature Flag
VPCOMUD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPCOMUD *xmm1*, *xmm2, xmm3/mem128, imm8*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	EE /r ib

Related Instructions

VPCOMUB, VPCOMUW, VPCOMUQ, VPCOMB, VPCOMW, VPCOMD, VPCOMQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPCOMUQ Compare Vector Unsigned Quadwords

Compares corresponding packed unsigned quadwords in the first and second sources and writes the result of each comparison to the corresponding quadword of the destination. The result of each comparison is a 64-bit value of all 1s (TRUE) or all 0s (FALSE).

There are four operands: VPCOMUQ *dest, src1, src2, imm8*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the results are written to the destination XMM register, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by the XOP.vvvv field and the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field.

The comparison type is specified by bits [2:0] of an immediate-byte operand (*imm8)*. Each type has an alias mnemonic to facilitate coding.

imm8[2:0]	Comparison	Mnemonic
000	Less Than	VPCOMLTUQ
001	Less Than or Equal	VPCOMLEUQ
010	Greater Than	VPCOMGTUQ
011	Greater Than or Equal	VPCOMGEUQ
100	Equal	VPCOMEQUQ
101	Not Equal	VPCOMNEQUQ
110	False	VPCOMFALSEUQ
111	True	VPCOMTRUEUQ

Instruction Support

Form	Subset	Feature Flag
VPCOMUQ	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPCOMUQ *xmm1*, *xmm2, xmm3/mem128, imm8*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	EF /r ib

Related Instructions

VPCOMUB, VPCOMUW, VPCOMUD, VPCOMB, VPCOMW, VPCOMD, VPCOMQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPCOMUW — Compare Vector Unsigned Words

Compares corresponding packed unsigned words in the first and second sources and writes the result of each comparison to the corresponding word of the destination. The result of each comparison is a 16-bit value of all 1s (TRUE) or all 0s (FALSE).

There are four operands: VPCOMUW *dest, src1, src2, imm8*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the results are written to the destination XMM register, bits [255:128] of the corresponding YMM register are cleared.

The first source (src1) is an XMM register specified by the XOP.vvvv field and the second source (src2) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field.

The comparison type is specified by bits [2:0] of an immediate-byte operand (*imm8)*. Each type has an alias mnemonic to facilitate coding.

imm8[2:0]	Comparison	Mnemonic
000	Less Than	VPCOMLTUW
001	Less Than or Equal	VPCOMLEUW
010	Greater Than	VPCOMGTUW
011	Greater Than or Equal	VPCOMGEUW
100	Equal	VPCOMEQUW
101	Not Equal	VPCOMNEQUW
110	False	VPCOMFALSEUW
111	True	VPCOMTRUEUW

Instruction Support

Form	Subset	Feature Flag
VPCOMUW	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPCOMUW *xmm1*, *xmm2, xmm3/mem128, imm8*	8F	$\overline{\text{RXB}}$.08	0.$\overline{\textit{src1}}$.0.00	ED /r ib

Related Instructions

VPCOMUB, VPCOMUD, VPCOMUQ, VPCOMB, VPCOMW, VPCOMD, VPCOMQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPCOMW Compare Vector Signed Words

Compares corresponding packed signed words in the first and second sources and writes the result of each comparison in the corresponding word of the destination. The result of each comparison is a 16-bit value of all 1s (TRUE) or all 0s (FALSE).

There are four operands: VPCOMW *dest, src1, src2, imm8*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the results are written to the destination XMM register, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by the XOP.vvvv field and the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field.

The comparison type is specified by bits [2:0] of an immediate-byte operand (*imm8)*. Each type has an alias mnemonic to facilitate coding.

imm8[2:0]	Comparison	Mnemonic
000	Less Than	VPCOMLTW
001	Less Than or Equal	VPCOMLEW
010	Greater Than	VPCOMGTW
011	Greater Than or Equal	VPCOMGEW
100	Equal	VPCOMEQW
101	Not Equal	VPCOMNEQW
110	False	VPCOMFALSEW
111	True	VPCOMTRUEW

Instruction Support

Form	Subset	Feature Flag
VPCOMW	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPCOMW *xmm1*, *xmm2, xmm3/mem128, imm8*	8F	$\overline{\text{RXB}}$.08	0.$\overline{\textit{src1}}$.0.00	CD /r ib

Related Instructions

VPCOMUB, VPCOMUW, VPCOMUD, VPCOMUQ, VPCOMB, VPCOMD, VPCOMQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPERM2F128 Permute Floating-Point 128-bit

Copies 128 bits of floating-point data from a selected octword of two 256-bit source operands or zero to each octword of a 256-bit destination, as specified by an immediate byte operand.

The immediate operand is encoded as follows.

Destination	Immediate-Byte Bit Field	Value of Bit Field	Source 1 Bits Copied	Source 2 Bits Copied
[127:0]	[1:0]	00	[127:0]	—
		01	[255:128]	—
		10	—	[127:0]
		11	—	[255:128]
Setting *imm8* [3] clears bits [127:0] of the destination; i*mm8* [2] is ignored.				
[255:128]	[5:4]	00	[127:0]	—
		01	[255:128]	—
		10	—	[127:0]
		11	—	[255:128]
Setting *imm8* [7] clears bits [255:128] of the destination; i*mm8* [6] is ignored.				

This is a 256-bit extended-form instruction:

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
VPERM2F128	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPERM2F128 *ymm1*, *ymm2*, *ymm3/mem256, imm8*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.1.01	06 /r ib

Related Instructions

VEXTRACTF128, VINSERTF128, VPERMILPD, VPERMILPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.L = 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
A — AVX exception.				

VPERM2I128 Permute Integer 128-bit

Copies 128 bits of integer data from a selected octword of two 256-bit source operands or zero to each octword of a 256-bit destination, as specified by an immediate byte operand.

The immediate operand is encoded as follows.

Destination	Immediate-Byte Bit Field	Value of Bit Field	Source 1 Bits Copied	Source 2 Bits Copied
[127:0]	[1:0]	00	[127:0]	—
		01	[255:128]	—
		10	—	[127:0]
		11	—	[255:128]
Setting *imm8* [3] clears bits [127:0] of the destination; *imm8* [2] is ignored.				
[255:128]	[5:4]	00	[127:0]	—
		01	[255:128]	—
		10	—	[127:0]
		11	—	[255:128]
Setting *imm8* [7] clears bits [255:128] of the destination; *imm8* [6] is ignored.				

This is a 256-bit extended-form instruction:

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register. Bits 2 and 6 of the immediate byte are ignored.

Instruction Support

Form	Subset	Feature Flag
VPERM2I128	AVX2	CPUID Fn0000_0007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPERM2I128 *ymm1*, *ymm2*, *ymm3/mem256, imm8*	C4	$\overline{\text{RXB}}$.03	0.$\overline{\textit{src1}}$.1.01	46 /r ib

Related Instructions

VEXTRACTI128, VEXTRACTF128, VINSERTI128, VINSERTF128, VPERMILPD, VPERMILPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.L = 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
A — AVX exception.				

VPERMD Packed Permute Doubleword

Copies selected doublewords from a 256-bit value located either in memory or a YMM register to specific doublewords of the destination YMM register. For each doubleword of the destination, selection of which doubleword to copy from the source is specified by a selector field in the corresponding doubleword of a YMM register.

There is a single form of this instruction:

VPERMD *dest, src1, src2*

The first source operand provides eight 3-bit selectors, each selector occupying the least-significant bits of a doubleword. Each selector specifies the index of the doubleword of the second source operand to be copied to the destination. The doubleword in the destination that each selector controls is based on its position within the first source operand.

The index value may be the same in multiple selectors. This results in multiple copies of the same source doubleword being copied to the destination.

There is no 128-bit form of this instruction.

YMM Encoding

The destination is a YMM register. The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
VPERMD	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

Instruction Encoding

	Encoding			
Mnemonic	**VEX**	**RXB.map_select**	**W.vvvv.L.pp**	**Opcode**
VPERMD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src1}}$.1.01	36 /r

Related Instructions

VPERMQ, VPERMPD, VPERMPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	A	A	A	CR0.EM = 1.
	A	A	A	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L= 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	A	A	A	Lock prefix (F0h) preceding opcode.
Device not available, #NM	A	A	A	CR0.TS = 1.
Stack, #SS	A	A	A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	A	A	A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

VPERMIL2PD Permute Two-Source Double-Precision Floating-Point

Copies a selected quadword from one of two source operands to a selected quadword of the destination or clears the selected quadword of the destination. Values in a third source operand and an immediate two-bit operand control the operation.

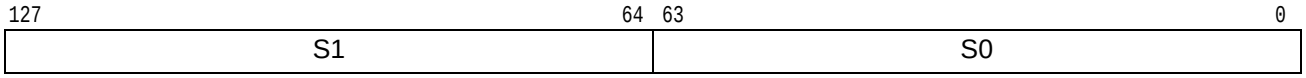
There are 128-bit and 256-bit versions of this instruction. Both versions have five operands:

VPERMIL2PD *dest, src1, src2, src3, m2z.*

The first four operands are either 128 bits or 256 bits wide, as determined by VEX.L. When the destination is an XMM register, bits [255:128] of the corresponding YMM register are cleared.

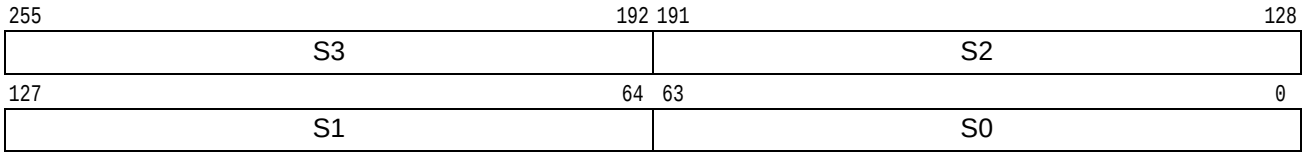
The third source operand is a selector that specifies how quadwords are copied or cleared in the destination. The selector contains one selector element for each quadword of the destination register.

Selector for 128-bit Instruction Form



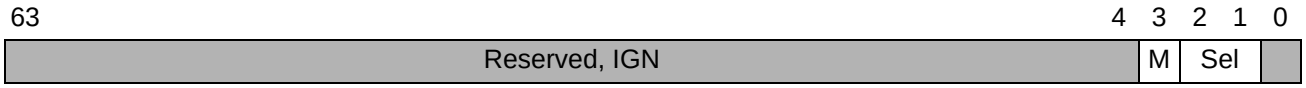
The selector for the 128-bit instruction form is an octword composed of two quadword selector elements S0 and S1. S0 (the lower quadword) controls the value written to destination quadword 0 (bits [63:0]) and S1 (the upper quadword) controls the destination quadword 1 (bits [127:64]).

Selector for 256-bit Instruction Form



The selector for the 256-bit instruction form is a double octword and adds two more selector elements S2 and S3. S0 controls the value written to the destination quadword 0 (bits [63:0]), S1 controls the destination quadword 1 (bits [127:64]), S2 controls the destination quadword 2 (bits [191:128]), and S3 controls the destination quadword 3 (bits [255:192]).

The layout of each selector element is as follows:



Bits	Mnemonic	Description
[63:4]	—	Reserved, IGN
[3]	M	Match
[2:1]	Sel	Select
[0]	—	Reserved, IGN

The fields are defined as follows:

- Sel — Select. Selects the source quadword to copy into the corresponding quadword of the destination:

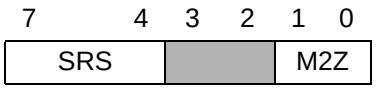
Sel Value	Source Selected for Destination Quadwords 0 and 1 (both forms)	Source Selected for Destination Quadwords 2 and 3 (256-bit form)
00b	*src1*[63:0]	*src1*[191:128]
01b	*src1*[127:64]	*src1*[255:192]
10b	*src2*[63:0]	*src2*[191:128]
11b	*src2*[127:64]	*src2*[255:192]

- M — Match bit. The combination of the Match bit in each selector element and the value of the M2Z field determines if the Select field is overridden. This is described below.

m2z immediate operand

The fifth operand is *m2z*. The assembler uses this 2-bit value to encode the M2Z field in the instruction. M2Z occupies bits [1:0] of an immediate byte. Bits [7:4] of the same byte are used to select one of 16 YMM/XMM registers. This dual use of the immediate byte is indicated in the instruction synopsis by the symbol “is5”.

The immediate byte is defined as follows.



Bits	Mnemonic	Description
[7:4]	SRS	Source Register Select
[3:2]	—	Reserved, IGN
[1:0]	M2Z	Match to Zero

Fields are defined as follows:

- SRS — Source Register Select. As with many other extended instructions, bits in the immediate byte are used to select a source operand register. This field is set by the assembler based on the operands listed in the instruction. See discussion in “*src2* and *src3* Operand Addressing” below.
- M2Z — Match to Zero. This field, combined with the M bit of the selector element, controls the function of the Sel field as follows:

M2Z Field	Selector M Bit	Value Loaded into Destination Quadword
0Xb	X	Source quadword selected by selector element Sel field.
10b	0	Source quadword selected by selector element Sel field.
10b	1	Zero
11b	0	Zero
11b	1	Source quadword selected by selector element Sel field.

src2 and *src3* Operand Addressing

In 64-bit mode, VEX.W and bits [7:4] of the immediate byte specify *src2* and *src3*:

- When VEX.W = 0, *src2* is either a register or a memory location specified by ModRM.r/m and *src3* is a register specified by bits [7:4] of the immediate byte.
- When VEX.W = 1, s*rc2* is a register specified by bits [7:4] of the immediate byte and *src3* is either a register or a memory location specified by ModRM.r/m.

In non-64-bit mode, bit 7 is ignored.

Instruction Support

Form	Subset	Feature Flag
VPERMIL2PD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

		Encoding		
Mnemonic	**VEX**	**RXB.map_select**	**W.vvvv.L.pp**	**Opcode**
VPERMIL2PD *xmm1*, *xmm2*, *xmm3/mem128*, *xmm4, m2z*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.0.01	49 /r is5
VPERMIL2PD *xmm1*, *xmm2*, *xmm3*, *xmm4/mem128, m2z*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.0.01	49 /r is5
VPERMIL2PD *ymm1*, *ymm2*, *ymm3/mem256*, *ymm4, m2z*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.1.01	49 /r is5
VPERMIL2PD *ymm1*, *ymm2*, *ymm3*, *ymm4/mem256, m2z*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.1.01	49 /r is5

NOTE: VPERMIL2PD is encoded using the VEX prefix even though it is an XOP instruction.

Related Instructions

VPERM2F128, VPERMIL2PS, VPERMILPD, VPERMILPS, VPPERM

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPERMIL2PS Permute Two-Source Single-Precision Floating-Point

Copies a selected doubleword from one of two source operands to a selected doubleword of the destination or clears the selected doubleword of the destination. Values in a third source operand and an immediate two-bit operand control operation.

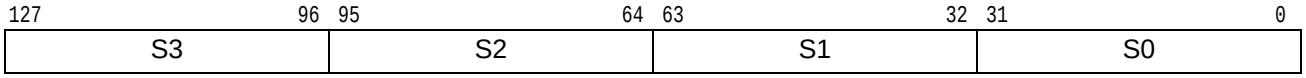
There are 128-bit and 256-bit versions of this instruction. Both versions have five operands:

VPERMIL2PS *dest, src1, src2, src3, m2z*

The first four operands are either 128 bits or 256 bits wide, as determined by VEX.L. When the destination is an XMM register, bits [255:128] of the corresponding YMM register are cleared.

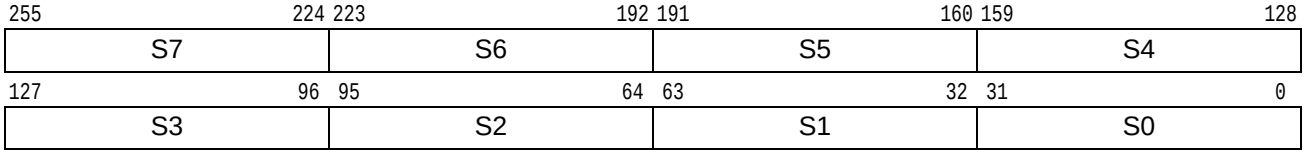
The third source operand is a selector that specifies how doublewords are copied or cleared in the destination. The selector contains one selector element for each doubleword of the destination register.

Selector for 128-bit Instruction Form



The selector for the 128-bit instruction form is an octword containing four selector elements S0–S3. S0 controls the value written to the destination doubleword 0 (bits [31:0]), S1 controls the destination doubleword 1 (bits [63:32]), S2 controls the destination doubleword 2 (bits [95:64]), and S3 controls the destination doubleword 3 (bits [127:96]).

Selector for 256-bit Instruction Form



The selector for the 256-bit instruction form is a double octword and adds four more selector elements S4–S7. S4 controls the value written to the destination doubleword 4 (bits [159:128]), S5 controls the destination doubleword 5 (bits [191:160]), S6 controls the destination doubleword 6 (bits [223:192]), and S7 controls the destination doubleword 7 (bits [255:224]).

The layout of each selector element is as follows.

Bits	Mnemonic	Description
[31:4]	—	Reserved, IGN
[3]	M	Match
[2:0]	Sel	Select

The fields are defined as follows:

- Sel — Select. Selects the source doubleword to copy into the corresponding doubleword of the destination:

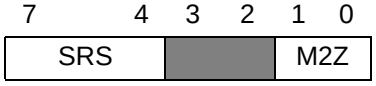
Sel Value	Source Selected for Destination Doublewords 0, 1, 2 and 3 (both forms)	Source Selected for Destination Doublewords 4, 5, 6 and 7 (256-bit form)
000b	*src1*[31:0]	*src1*[159:128]
001b	*src1*[63:32]	*src1*[191:160]
010b	*src1*[95:64]	*src1*[223:192]
011b	*src1*[127:96]	*src1*[255:224]
100b	*src2*[31:0]	*src2*[159:128]
101b	*src2*[63:32]	*src2*[191:160]
110b	*src2*[95:64]	*src2*[223:192]
111b	*src2*[127:96]	*src2*[255:224]

- M — Match. The combination of the M bit in each selector element and the value of the M2Z field determines if the Sel field is overridden. This is described below.

m2z immediate operand

The fifth operand is *m2z*. The assembler uses this 2-bit value to encode the M2Z field in the instruction. M2Z occupies bits [1:0] of an immediate byte. Bits [7:4] of the same byte are used to select one of 16 YMM/XMM registers. This dual use of the immediate byte is indicated in the instruction synopsis by the symbol “is5”.

The immediate byte is defined as follows.



Bits	Mnemonic	Description
[7:4]	SRS	Source Register Select
[3:2]	—	Reserved, IGN
[1:0]	M2Z	Match to Zero

Fields are defined as follows:

- SRS — Source Register Select. As with many other extended instructions, bits in the immediate byte are used to select a source operand register. This field is set by the assembler based on the operands listed in the instruction. See discussion in “*src2* and *src3* Operand Addressing” below.
- M2Z — Match to Zero. This field, combined with the M bit of the selector element, controls the function of the Sel field as follows:

M2Z Field	Selector M Bit	Value Loaded into Destination Doubleword
0Xb	X	Source doubleword selected by Sel field.
10b	0	Source doubleword selected by Sel field.

M2Z Field	Selector M Bit	Value Loaded into Destination Doubleword
10b	1	Zero
11b	0	Zero
11b	1	Source doubleword selected by Sel field.

src2 and *src3* Operand Addressing

In 64-bit mode, VEX.W and bits [7:4] of the immediate byte specify *src2* and *src3*:

- When VEX.W = 0, *src2* is either a register or a memory location specified by ModRM.r/m and *src3* is a register specified by bits [7:4] of the immediate byte.
- When VEX.W = 1, s*rc2* is a register specified by bits [7:4] of the immediate byte and *src3* is either a register or a memory location specified by ModRM.r/m.

In non-64-bit mode, bit 7 is ignored.

Instruction Support

Form	Subset	Feature Flag
VPERMIL2PS	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

		Encoding		
Mnemonic	**VEX**	**RXB.map_select**	**W.vvvv.L.pp**	**Opcode**
VPERMIL2PS *xmm1*, *xmm2, xmm3/mem128, xmm4, m2z*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.0.01	48 /r is5
VPERMIL2PS *xmm1*, *xmm2, xmm3, xmm4/mem128, m2z*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.0.01	48 /r is5
VPERMIL2PS *ymm1*, *ymm2, ymm3/mem256, ymm4, m2z*	C4	$\overline{\text{RXB}}$.03	0.$\overline{src1}$.1.01	48 /r is5
VPERMIL2PS *ymm1*, *ymm2, ymm3, ymm4/mem256, m2z*	C4	$\overline{\text{RXB}}$.03	1.$\overline{src1}$.1.01	48 /r is5

NOTE: VPERMIL2PS is encoded using the VEX prefix even though it is an XOP instruction.

Related Instructions

VPERM2F128, VPERMIL2PD, VPERMILPD, VPERMILPS, VPPERM

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

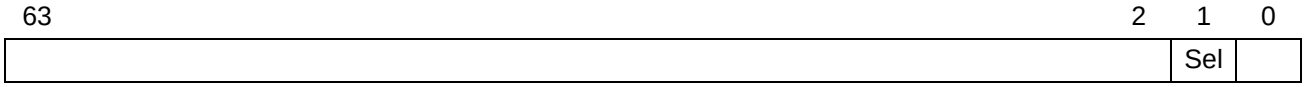
Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPERMILPD Permute Double-Precision

Copies double-precision floating-point values from a source to a destination. Source and destination can be selected in two ways. There are different encodings for each selection method.

Selection by bits in a source register or memory location:

Each quadword of the operand is defined as follows.



A bit selects source and destination. Only bit [1] is used; bits [63:2} and bit [0] are ignored. Setting the bit selects the corresponding quadword element of the source and the destination.

Selection by bits in an immediate byte:

Each bit corresponds to a destination quadword. Only bits [3:2] and bits [1:0] are used; bits [7:4] are ignored. Selections are defined as follows.

Destination Quadword	Immediate-Byte Bit Field	Value of Bit Field	Source 1 Bits Copied
Used by 128-bit encoding and 256-bit encoding			
[63:0]	[0]	0	[63:0]
		1	[127:64]
[127:64]	[1]	0	[63:0]
		1	[127:64]
Used only by 256-bit encoding			
[191:128]	[2]	0	[191:128]
		1	[255:192]
[255:192]	[3]	0	[191:128]
		1	[255:192]

This extended-form instruction has both 128-bit and 256-bit encoding.

XMM Encoding

There are two encodings, one for each selection method:

- The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.
- The first source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. There is a third, immediate byte operand. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

There are two encodings, one for each selection method:

- The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.
- The first source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register. There is a third, immediate byte operand.

Instruction Support

Form	Subset	Feature Flag
VPERMILPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
Selection by source register or memory:				
VPERMILPD *xmm1, xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src1}}$.0.01	0D /r
VPERMILPD *ymm1, ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src1}}$.1.01	0D /r
Selection by immediate byte operand:				
VPERMILPD *xmm1*, *xmm2/mem128*, *imm8*	C4	$\overline{\text{RXB}}$.03	0.1111.0.01	05 /r ib
VPERMILPD *ymm1*, *ymm2*/mem256, *imm8*	C4	$\overline{\text{RXB}}$.03	0.1111.1.01	05 /r ib

Related Instructions

VPERM2F128, VPERMIL2PD, VPERMIL2PS, VPERMILPS, VPPERM

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

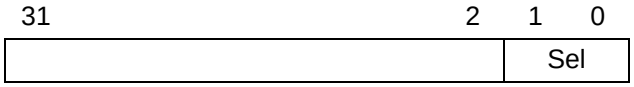
Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b (for versions with immediate byte operand only).
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

VPERMILPS Permute Single-Precision

Copies single-precision floating-point values from a source to a destination. Source and destination can be selected in two ways. There are different encodings for each selection method.

Selection by bit fields in a source register or memory location:

Each doubleword of the operand is defined as follows.



Each bit field corresponds to a destination doubleword. Bit values select a source doubleword. Only bits [1:0] of each word are used; bits [31:2} are ignored. The 128-bit encoding uses four two-bit fields; the 256-bit version uses eight two-bit fields. Field encoding is as follows.

Destination Doubleword	Immediate Operand Bit Field	Value of Bit Field	Source Bits Copied
[31:0]	[1:0]	00	[31:0]
		01	[63:32]
		10	[95:64]
		11	[127:96]
[63:32]	[33:32]	00	[31:0]
		01	[63:32]
		10	[95:64]
		11	[127:96]
[95:64]	[65:64]	00	[31:0]
		01	[63:32]
		10	[95:64]
		11	[127:96]
[127:96]	[97:96]	00	[31:0]
		01	[63:32]
		10	[95:64]
		11	[127:96]

Destination Doubleword	Immediate Operand Bit Field	Value of Bit Field	Source Bits Copied
Upper 128 bits of 256-bit source and destination used by 256-bit encoding			
[159:128]	[129:128]	00	[159:128]
		01	[191:160]
		10	[223:192]
		11	[255:224]
[191:160]	[161:160]	00	[159:128]
		01	[191:160]
		10	[223:192]
		11	[255:224]
[223:192]	[193:192]	00	[159:128]
		01	[191:160]
		10	[223:192]
		11	[255:224]
[255:224]	[225:224]	00	[159:128]
		01	[191:160]
		10	[223:192]
		11	[255:224]

Selection by bit fields in an immediate byte:

Each bit field corresponds to a destination doubleword. For the 256-bit encoding, the fields specify sources and destinations in both the upper and lower 128 bits of the register. Selections are defined as follows.

Destination Doubleword	Bit Field	Value of Bit Field	Source Bits Copied
[31:0]	[1:0]	00	[31:0]
		01	[63:32]
		10	[95:64]
		11	[127:96]
[63:32]	[3:2]	00	[31:0]
		01	[63:32]
		10	[95:64]
		11	[127:96]
[95:64]	[5:4]	00	[31:0]
		01	[63:32]
		10	[95:64]
		11	[127:96]
[127:96]	[7:6]	00	[31:0]
		01	[63:32]
		10	[95:64]
		11	[127:96]

Destination Doubleword	Bit Field	Value of Bit Field	Source Bits Copied
Upper 128 bits of 256-bit source and destination used by 256-bit encoding			
[159:128]	[1:0]	00	[159:128]
		01	[191:160]
		10	[223:192]
		11	[255:224]
[191:160]	[3:2]	00	[159:128]
		01	[191:160]
		10	[223:192]
		11	[255:224]
[223:192]	[5:4]	00	[159:128]
		01	[191:160]
		10	[223:192]
		11	[255:224]
[255:224]	[7:6]	00	[159:128]
		01	[191:160]
		10	[223:192]
		11	[255:224]

This extended-form instruction has both 128-bit and 256-bit encodings:

XMM Encoding

There are two encodings, one for each selection method:

- The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.
- The first source operand is either an XMM register or a 128-bit memory location. The destination is an XMM register. There is a third, immediate byte operand. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

There are two encodings, one for each selection method:

- The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.
- The first source operand is either a YMM register or a 256-bit memory location. The destination is a YMM register. There is a third, immediate byte operand.

Instruction Support

Form	Subset	Feature Flag
VPERMILPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
Selection by source register or memory:				
VPERMILPS *xmm1, xmm2*, *xmm3*/*mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.0.01	0C /r
VPERMILPS *ymm1, ymm2*, *ymm3*/*mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.1.01	0C /r
Selection by immediate byte operand:				
VPERMILPS *xmm1*, *xmm2/mem128*, *imm8*	C4	$\overline{\text{RXB}}$.03	0.1111.0.01	04 /r ib
VPERMILPS *ymm1*, *ymm2*/mem256, *imm8*	C4	$\overline{\text{RXB}}$.03	0.1111.1.01	04 /r ib

Related Instructions

VPERM2F128, VPERMIL2PD, VPERMIL2PS, VPERMILPD, VPPERM

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b (for versions with immediate byte operand only).
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

VPERMPD Packed Permute Double-Precision Floating-Point

Copies selected quadwords from a 256-bit value located either in memory or a YMM register to specific quadwords of the destination. For each quadword of the destination, selection of which quadword to copy from the source is specified by a 2 bit selector field in an immediate byte.

There is a single form of this instruction:

VPERMPD *dest, src, imm8*

The selection of which quadword of the source operand to copy to each quadword of the destination is specified by four 2-bit selector fields in the immediate byte. Bits [1:0] specify the index of the quadword to be copied to the destination quadword 0. Bits [3:2] select the quadword to be copied to quadword 1, bits [5:4] select the quadword to be copied to quadword 2, and bits [7:6] select the quadword to be copied to quadword 3.

The index value may be the same in multiple selectors. This results in multiple copies of the same source quadword being copied to the destination.

There is no 128-bit form of this instruction.

YMM Encoding

The destination is a YMM register. The source operand is a YMM register or a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
VPERMPD	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

	Encoding			
Mnemonic	**VEX**	**RXB.map_select**	**W.vvvv.L.pp**	**Opcode**
VPERMPD *ymm1*, *ymm2/mem256, imm8*	C4	$\overline{\text{RXB}}$.03	1.1111.1.01	01 /r ib

Related Instructions

VPERMD, VPERMQ, VPERMPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	A	A	A	CR0.EM = 1.
	A	A	A	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L= 0.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	A	A	A	Lock prefix (F0h) preceding opcode.
Device not available, #NM	A	A	A	CR0.TS = 1.
Stack, #SS	A	A	A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	A	A	A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

VPERMPS Packed Permute Single-Precision Floating-Point

Copies selected doublewords from a 256-bit value located either in memory or a YMM register to specific doublewords of the destination YMM register. For each doubleword of the destination, selection of which doubleword to copy from the source is specified by a selector field in the corresponding doubleword of a YMM register.

There is a single form of this instruction:

VPERMPS *dest, src1, src2*

The first source operand provides eight 3-bit selectors, each selector occupying the least-significant bits of a doubleword. Each selector specifies the index of the doubleword of the second source operand to be copied to the destination. The doubleword in the destination that each selector controls is based on its position within the first source operand.

The index value may be the same in multiple selectors. This results in multiple copies of the same source doubleword being copied to the destination.

There is no 128-bit form of this instruction.

YMM Encoding

The destination is a YMM register. The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
VPERMPS	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

	Encoding			
Mnemonic	**VEX**	**RXB.map_select**	**W.vvvv.L.pp**	**Opcode**
VPERMPS *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.1.01	16 /r

Related Instructions

VPERMD, VPERMQ, VPERMPD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	A	A	A	CR0.EM = 1.
	A	A	A	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L= 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	A	A	A	Lock prefix (F0h) preceding opcode.
Device not available, #NM	A	A	A	CR0.TS = 1.
Stack, #SS	A	A	A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	A	A	A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

VPERMQ Packed Permute Quadword

Copies selected quadwords from a 256-bit value located either in memory or a YMM register to specific quadwords of the destination. For each quadword of the destination, selection of which quadword to copy from the source is specified by a 2 bit selector field in an immediate byte.

There is a single form of this instruction:

VPERMQ *dest, src, imm8*

The selection of which quadword of the source operand to copy to each quadword of the destination is specified by four 2-bit selector fields in the immediate byte. Bits [1:0] specify the index of the quadword to be copied to the destination quadword 0. Bits [3:2] select the quadword to be copied to quadword 1, bits [5:4] select the quadword to be copied to quadword 2, and bits [7:6] select the quadword to be copied to quadword 3.

The index value may be the same in multiple selectors. This results in multiple copies of the same source quadword being copied to the destination.

There is no 128-bit form of this instruction.

YMM Encoding

The destination is a YMM register. The source operand is a YMM register or a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
VPERMQ	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

	Encoding			
Mnemonic	**VEX**	**RXB.map_select**	**W.vvvv.L.pp**	**Opcode**
VPERMQ *ymm1*, *ymm2/mem256, imm8*	C4	$\overline{\text{RXB}}$.03	1.1111.1.01	00 /r ib

Related Instructions

VPERMD, VPERMPD, VPERMPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	A	A	A	CR0.EM = 1.
	A	A	A	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L= 0.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	A	A	A	Lock prefix (F0h) preceding opcode.
Device not available, #NM	A	A	A	CR0.TS = 1.
Stack, #SS	A	A	A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	A	A	A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

VPGATHERDD Conditionally Gather Doublewords, Doubleword Indices

Conditionally loads doubleword values from memory using VSIB addressing with doubleword indices.

The instruction is of the form:

VPGATHERDD *dest, mem32[vm32x/y], mask*

The loading of each element of the destination register is conditional based on the value of the corresponding element of the mask (second source operand). If the most-significant bit of the *i*th element of the mask is set, the *i*th element of the destination is loaded from memory using the *i*th address of the array of effective addresses calculated using VSIB addressing.

The index register is treated as an array of signed 32-bit values. Doubleword elements of the destination for which the corresponding mask element is zero are not affected by the operation. If no exceptions occur, the mask register is set to zero.

Execution of the instruction can be suspended by an exception if the exception is triggered by an element other than the rightmost element loaded. When this happens, the destination register and the mask operand may be observed as partially updated. Elements that have been loaded will have their mask elements set to zero. If any traps or faults are pending from elements that have been loaded, they will be delivered in lieu of the exception; in this case, the RF flag is set so that an instruction breakpoint is not re-triggered when the instruction execution is resumed.

See Section 1.3, “VSIB Addressing,” on page 6 for a discussion of the VSIB addressing mode.

There are 128-bit and 256-bit forms of this instruction.

XMM Encoding

The destination is an XMM register. The first source operand is up to four 32-bit values located in memory. The second source operand (the mask) is an XMM register. The index vector is the four doublewords of an XMM register. Bits [255:128] of the YMM register that corresponds to the destination and bits [255:128] of the YMM register that corresponds to the second source (mask) operand are cleared.

YMM Encoding

The destination is a YMM register. The first source operand is up to eight 32-bit values located in memory. The second source operand (the mask) is a YMM register. The index vector is the eight doublewords of a YMM register.

Instruction Support

Form	Subset	Feature Flag
VPGATHERDD	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPGATHERDD *xmm1, vm32x, xmm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.0.01	90 /r
VPGATHERDD *ymm1, vm32y, ymm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.1.01	90 /r

Related Instructions

VGATHERDPD, VGATHERDPS, VGATHERQPD, VGATHERQPS, VPGATHERDQ, VPGATHERQD, VPGATHERQQ

rFLAGS Affected

RF

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
			A	MODRM.mod = 11b
			A	MODRM.rm ! = 100b
			A	YMM/XMM registers specified for destination, mask, and index not unique.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

VPGATHERDQ Conditionally Gather Quadwords, Doubleword Indices

Conditionally loads quadword values from memory using VSIB addressing with doubleword indices.

The instruction is of the form:

VPGATHERDQ *dest, mem64[vm32x], mask*

The loading of each element of the destination register is conditional based on the value of the corresponding element of the mask (second source operand). If the most-significant bit of the *i*th element of the mask is set, the *i*th element of the destination is loaded from memory using the *i*th address of the array of effective addresses calculated using VSIB addressing.

The index register is treated as an array of signed 32-bit values. Quadword elements of the destination for which the corresponding mask element is zero are not affected by the operation. If no exceptions occur, the mask register is set to zero.

Execution of the instruction can be suspended by an exception if the exception is triggered by an element other than the rightmost element loaded. When this happens, the destination register and the mask operand may be observed as partially updated. Elements that have been loaded will have their mask elements set to zero. If any traps or faults are pending from elements that have been loaded, they will be delivered in lieu of the exception; in this case, the RF flag is set so that an instruction breakpoint is not re-triggered when the instruction execution is resumed.

See Section 1.3, “VSIB Addressing,” on page 6 for a discussion of the VSIB addressing mode.

There are 128-bit and 256-bit forms of this instruction.

XMM Encoding

The destination is an XMM register. The first source operand is up to two 64-bit values located in memory. The second source operand (the mask) is an XMM register. The index vector is the two low-order doublewords of an XMM register; the two high-order doublewords of the index register are not used. Bits [255:128] of the YMM register that corresponds to the destination and bits [255:128] of the YMM register that corresponds to the second source (mask) operand are cleared.

YMM Encoding

The destination is a YMM register. The first source operand is up to four 64-bit values located in memory. The second source operand (the mask) is a YMM register. The index vector is the four doublewords of an XMM register.

Instruction Support

Form	Subset	Feature Flag
VPGATHERDQ	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPGATHERDQ *xmm1, vm32x, xmm2*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	90 /r
VPGATHERDQ *ymm1, vm32x, ymm2*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	90 /r

Related Instructions

VGATHERDPD, VGATHERDPS, VGATHERQPD, VGATHERQPS, VPGATHERDD, VPGATHERQD, VPGATHERQQ

rFLAGS Affected

RF

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
			A	MODRM.mod = 11b
			A	MODRM.rm ! = 100b
			A	YMM/XMM registers specified for destination, mask, and index not unique.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

VPGATHERQD Conditionally Gather Doublewords, Quadword Indices

Conditionally loads doubleword values from memory using VSIB addressing with quadword indices.

The instruction is of the form:

VPGATHERQD *dest, mem32[vm64x/y], mask*

The loading of each element of the destination register is conditional based on the value of the corresponding element of the mask (second source operand). If the most-significant bit of the *i*th element of the mask is set, the *i*th element of the destination is loaded from memory using the *i*th address of the array of effective addresses calculated using VSIB addressing.

The index register is treated as an array of signed 64-bit values. Doubleword elements of the destination for which the corresponding mask element is zero are not affected by the operation. If no exceptions occur, the mask register is set to zero.

Execution of the instruction can be suspended by an exception if the exception is triggered by an element other than the rightmost element loaded. When this happens, the destination register and the mask operand may be observed as partially updated. Elements that have been loaded will have their mask elements set to zero. If any traps or faults are pending from elements that have been loaded, they will be delivered in lieu of the exception; in this case, the RF flag is set so that an instruction breakpoint is not re-triggered when the instruction execution is resumed.

See Section 1.3, “VSIB Addressing,” on page 6 for a discussion of the VSIB addressing mode.

There are 128-bit and 256-bit forms of this instruction.

XMM Encoding

The destination is an XMM register. The first source operand is up to two 32-bit values located in memory. The second source operand (the mask) is an XMM register. The index vector is the two quadwords of an XMM register. The upper half of the destination register and the mask register are cleared. Bits [255:128] of the YMM register that corresponds to the destination and bits [255:128] of the YMM register that corresponds to the mask register are cleared.

YMM Encoding

The destination is an XMM register. The first source operand is up to four 32-bit values located in memory. The second source operand (the mask) is an XMM register. The index vector is the four quadwords of a YMM register. Bits [255:128] of the YMM register that corresponds to the destination and bits [255:128] of the YMM register that corresponds to the mask register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPGATHERQD	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPGATHERQD *xmm1, vm64x, xmm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.0.01	91 /r
VPGATHERQD *xmm1, vm64y, xmm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.1.01	91 /r

Related Instructions

VGATHERDPD, VGATHERDPS, VGATHERQPD, VGATHERQPS, VPGATHERDD, VPGATHERDQ, VPGATHERQQ

rFLAGS Affected

RF

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
			A	MODRM.mod = 11b
			A	MODRM.rm ! = 100b
			A	YMM/XMM registers specified for destination, mask, and index not unique.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

VPGATHERQQ Conditionally Gather Quadwords, Quadword Indices

Conditionally loads quadword values from memory using VSIB addressing with quadword indices.

The instruction is of the form:

VPGATHERQQ *dest, mem64[vm64x/y], mask*

The loading of each element of the destination register is conditional based on the value of the corresponding element of the mask (second source operand). If the most-significant bit of the *i*th element of the mask is set, the *i*th element of the destination is loaded from memory using the *i*th address of the array of effective addresses calculated using VSIB addressing.

The index register is treated as an array of signed 64-bit values. Quadword elements of the destination for which the corresponding mask element is zero are not affected by the operation. If no exceptions occur, the mask register is set to zero.

Execution of the instruction can be suspended by an exception if the exception is triggered by an element other than the rightmost element loaded. When this happens, the destination register and the mask operand may be observed as partially updated. Elements that have been loaded will have their mask elements set to zero. If any traps or faults are pending from elements that have been loaded, they will be delivered in lieu of the exception; in this case, the RF flag is set so that an instruction breakpoint is not re-triggered when the instruction execution is resumed.

See Section 1.3, “VSIB Addressing,” on page 6 for a discussion of the VSIB addressing mode.

There are 128-bit and 256-bit forms of this instruction.

XMM Encoding

The destination is an XMM register. The first source operand is up to two 64-bit values located in memory. The second source operand (the mask) is an XMM register. The index vector is the two quadwords of an XMM register. Bits [255:128] of the YMM register that corresponds to the destination and bits [255:128] of the YMM register that corresponds to the second source (mask) operand are cleared.

YMM Encoding

The destination is a YMM register. The first source operand is up to four 64-bit values located in memory. The second source operand (the mask) is a YMM register. The index vector is the four quadwords of a YMM register.

Instruction Support

Form	Subset	Feature Flag
VPGATHERQQ	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPGATHERQQ *xmm1, vm64x, xmm2*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.0.01	91 /r
VPGATHERQQ *ymm1, vm64y, ymm2*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.1.01	91 /r

Related Instructions

VGATHERDPD, VGATHERDPS, VGATHERQPD, VGATHERQPS, VPGATHERDD, VPGATHERDQ, VPGATHERQD

rFLAGS Affected

RF

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
			A	MODRM.mod = 11b
			A	MODRM.rm ! = 100b
			A	YMM/XMM registers specified for destination, mask, and index not unique.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

VPHADDBD Packed Horizontal Add Signed Byte to Signed Doubleword

Adds four sets of four 8-bit signed integer values of the source and packs the sign-extended sums into the corresponding doubleword of the destination.

There are two operands: VPHADDBD *dest*, *src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. Bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHADDBD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDBD *xmm1*, *xmm2*/*mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	C2 /r

Related Instructions

VPHADDBW, VPHADDBQ, VPHADDWD, VPHADDWQ, VPHADDDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHADDBQ Packed Horizontal Add Signed Byte to Signed Quadword

Adds two sets of eight 8-bit signed integer values of the source and packs the sign-extended sums into the corresponding quadword of the destination.

There are two operands: VPHADDBQ *dest*, *src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. Bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHADDBQ	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDBQ *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	C3 /r

Related Instructions

VPHADDBW, VPHADDBD, VPHADDWD, VPHADDWQ, VPHADDDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHADDBW Packed Horizontal Add Signed Byte to Signed Word

Adds each adjacent pair of 8-bit signed integer values of the source and packs the sign-extended 16-bit integer result of each addition into the corresponding word element of the destination.

There are two operands: VPHADDBW *dest*, *src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. Bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHADDBW	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDBW *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	C1 /r

Related Instructions

VPHADDBD, VPHADDBQ, VPHADDWD, VPHADDWQ, VPHADDDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHADDDQ Packed Horizontal Add Signed Doubleword to Signed Quadword

Adds each adjacent pair of signed doubleword integer values of the source and packs the sign-extended sums into the corresponding quadword of the destination.

There are two operands: VPHADDDQ *dest*, *src*

The source is either an XMM register or a 128-bit memory location and the destination is an XMM register. Bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHADDDQ	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDDQ *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	CB /r

Related Instructions

VPHADDBW, VPHADDBD, VPHADDBQ, VPHADDWD, VPHADDWQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHADDUBD Packed Horizontal Add Unsigned Byte to Doubleword

Adds four sets of four 8-bit unsigned integer values of the source and packs the sums into the corresponding doublewords of the destination.

There are two operands: VPHADDUBD *dest, src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. Bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHADDUBD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDUBD *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	D2 /r

Related Instructions

VPHADDUBW, VPHADDUBQ, VPHADDUWD, VPHADDUWQ, VPHADDUDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHADDUBQ Packed Horizontal Add Unsigned Byte to Quadword

Adds two sets of eight 8-bit unsigned integer values from the second source and packs the sums into the corresponding quadword of the destination.

There are two operands: VPHADDUBQ *dest, src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. When the destination XMM register is written, bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHADDUBQ	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDUBQ *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	D3 /r

Related Instructions

VPHADDUBW, VPHADDUBD, VPHADDUWD, VPHADDUWQ, VPHADDUDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHADDUBW Packed Horizontal Add Unsigned Byte to Word

Adds each adjacent pair of 8-bit unsigned integer values of the source and packs the 16-bit integer sums to the corresponding word of the destination.

There are two operands: VPHADDUBW *dest*, *src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. Bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHADDUBW	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDUBW *xmm1*, *xmm2*/*mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	D1 /r

Related Instructions

VPHADDUBD, VPHADDUBQ, VPHADDUWD, VPHADDUWQ, VPHADDUDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHADDUDQ Packed Horizontal Add Unsigned Doubleword to Quadword

Adds two adjacent pairs of 32-bit unsigned integer values of the source and packs the sums into the corresponding quadword of the destination.

There are two operands: VPHADDUDQ *dest, src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. Bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHADDUDQ	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDUDQ *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	DB /r

Related Instructions

VPHADDUBW, VPHADDUBD, VPHADDUBQ, VPHADDUWD, VPHADDUWQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHADDUWD Packed Horizontal Add Unsigned Word to Doubleword

Adds four adjacent pairs of 16-bit unsigned integer values of the source and packs the sums into the corresponding doubleword of the destination.

There are two operands: VPHADDUWD *dest*, *src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. Bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHADDUWD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDUWD *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	D6 /r

Related Instructions

VPHADDUBW, VPHADDUBD, VPHADDUBQ, VPHADDUWQ, VPHADDUDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHADDUWQ Packed Horizontal Add Unsigned Word to Quadword

Adds two pairs of 16-bit unsigned integer values of the source and packs the sums into the corresponding quadword element of the destination.

There are two operands: VPHADDUWQ *dest, src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. Bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHADDUWQ	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDUWQ *xmm1*, *xmm2*/*mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	D7 /r

Related Instructions

VPHADDUBW, VPHADDUBD, VPHADDUBQ, VPHADDUWD, VPHADDUDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHADDWD Packed Horizontal Add Signed Word to Signed Doubleword

Adds four adjacent pairs of 16-bit signed integer values of the source and packs the sign-extended sums to the corresponding doubleword of the destination.

There are two operands: VPHADDWD *dest*, *src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. Bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHADDWD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDWD *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	C6 /r

Related Instructions

VPHADDBW, VPHADDBD, VPHADDBQ, VPHADDWQ, VPHADDDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHADDWQ Packed Horizontal Add Signed Word to Signed Quadword

Adds four successive pairs of 16-bit signed integer values of the source and packs the sign-extended sums to the corresponding quadword of the destination.

There are two operands: VPHADDWQ *dest*, *src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. Bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHADDWQ	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHADDWQ *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	C7 /r

Related Instructions

VPHADDBW, VPHADDBD, VPHADDBQ, VPHADDWD, VPHADDDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHSUBBW Packed Horizontal Subtract Signed Byte to Signed Word

Subtracts the most significant signed integer byte from the least significant signed integer byte of each word element in the source and packs the sign-extended 16-bit integer differences into the destination.

There are two operands: VPHSUBBW *dest*, *src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. When the destination is written, bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHSUBBW	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHSUBBW *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	E1 /r

Related Instructions

VPHSUBWD, VPHSUBDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHSUBDQ Packed Horizontal Subtract Signed Doubleword to Signed Quadword

Subtracts the most significant signed integer doubleword from the least significant signed integer doubleword of each quadword in the source and packs the sign-extended 64-bit integer differences into the corresponding quadword element of the destination.

There are two operands: VPHSUBDQ *dest, src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. When the destination is written, bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHSUBDQ	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHSUBDQ *xmm1*, *xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	E3 /r

Related Instructions

VPHSUBBW, VPHSUBWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPHSUBWD Packed Horizontal Subtract Signed Word to Signed Doubleword

Subtracts the most significant signed integer word from the least significant signed integer word of each doubleword of the source and packs the sign-extended 32-bit integer differences into the destination.

There are two operands: VPHSUBWD *dest*, *src*

The destination is an XMM register and the source is either an XMM register or a 128-bit memory location. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPHSUBWD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPHSUBWD *xmm1*, *xmm2*/*mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	E2 /r

Related Instructions

VPHSUBBW, VPHSUBDQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMACSDD Packed Multiply Accumulate Signed Doubleword to Signed Doubleword

Multiplies each packed 32-bit signed integer value of the first source by the corresponding value of the second source, adds the corresponding value of the third source to the 64-bit signed integer product, and writes four 32-bit sums to the destination.

No saturation is performed on the sum. When the result of the multiplication causes non-zero values to be set in the upper 32 bits of the 64-bit product, they are ignored. When the result of the add overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set). In both cases, only the signed low-order 32 bits of the result are written to the destination.

There are four operands: VPMACSDD *dest, src1, src2, src3* *dest = src1* src2 + src3*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the destination is written, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by XOP.vvvv; the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field; and the third source (*src3*) is an XMM register specified by bits [7:4] of an immediate byte operand.

When the third source designates the same XMM register as the destination, the XMM register behaves as an accumulator.

Instruction Support

Form	Subset	Feature Flag
VPMACSDD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPMACSDD *xmm1, xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	9E /r ib

Related Instructions

VPMACSSWW, VPMACSWW, VPMACSSWD, VPMACSWD, VPMACSSDD, VPMACSSDQL, VPMACSSDQH, VPMACSDQL, VPMACSDQH, VPMADCSSWD, VPMADCSWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMACSDQH Packed Multiply Accumulate Signed High Doubleword to Signed Quadword

Multiplies the second 32-bit signed integer value of the first source by the corresponding value of the second source, then adds the low-order 64-bit signed integer value of the third source to the 64-bit signed integer product. Simultaneously, multiplies the fourth 32-bit signed integer value of the first source by the fourth 32-bit signed integer value of the second source, then adds the high-order 64-bit signed integer value of the third source to the 64-bit signed integer product. Writes two 64-bit sums to the destination.

No saturation is performed on the sum. When the result of the add overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set).

There are four operands: VPMACSDQH *dest, src1, src2, src3* *dest = src1* src2 + src3*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the destination is written, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by the XOP.vvvv field; the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field; and the third source (*src3*) is an XMM register specified by bits [7:4] of an immediate byte operand.

When the third source designates the same XMM register as the destination, the XMM register behaves as an accumulator.

Instruction Support

Form	Subset	Feature Flag
VPMACSDQH	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPMACSDQH *xmm1, xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.01000	0.$\overline{src1}$.0.00	9F /r ib

Related Instructions

VPMACSSWW, VPMACSWW, VPMACSSWD, VPMACSWD, VPMACSSDD, VPMACSDD, VPMACSSDQL, VPMACSSDQH, VPMACSDQL, VPMADCSSWD, VPMADCSWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMACSDQL Packed Multiply Accumulate Signed Low Doubleword to Signed Quadword

Multiplies the low-order 32-bit signed integer value of the first source by the corresponding value of the second source, then adds the low-order 64-bit signed integer value of the third source to the 64-bit signed integer product. Simultaneously, multiplies the third 32-bit signed integer value of the first source by the corresponding value of the second source, then adds the high-order 64-bit signed integer value of the third source to the 64-bit signed integer product. Writes two 64-bit sums to the destination register.

No saturation is performed on the sum. When the result of the add overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set). Only the low-order 64 bits of each result are written to the destination.

There are four operands: VPMACSDQL *dest, src1, src2, src3* *dest = src1* src2 + src3*

The destination is a YMM register specified by ModRM.reg. When the destination is written, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by XOP.vvvv; the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field; and the third source (*src3*) is an XMM register specified by bits [7:4] of an immediate byte operand.

When *src3* designates the same XMM register as the *dest* register, the XMM register behaves as an accumulator.

Instruction Support

Form	Subset	Feature Flag
VPMACSDQL	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPMACSDQL *xmm1, xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	97 /r ib

Related Instructions

VPMACSSWW, VPMACSWW, VPMACSSWD, VPMACSWD, VPMACSSDD, VPMACSDD, VPMACSSDQL, VPMACSSDQH, VPMACSDQH, VPMADCSSWD, VPMADCSWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMACSSDD Packed Multiply Accumulate with Saturation Signed Doubleword to Signed Doubleword

Multiplies each packed 32-bit signed integer value of the first source by the corresponding value of the second source, then adds the corresponding packed 32-bit signed integer value of the third source to each 64-bit signed integer product. Writes four saturated 32-bit sums to the destination.

Out of range results of the addition are saturated to fit into a signed 32-bit integer. For each packed value of the destination, when the value is larger than the largest signed 32-bit integer, it is saturated to 7FFF_FFFFh, and when the value is smaller than the smallest signed 32-bit integer, it is saturated to 8000_0000h.

There are four operands: VPMACSSDD *dest, src1, src2, src3* *dest = src1* src2 + src3*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the destination is written, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by XOP.vvvv; the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field; and the third source (*src3*) is an XMM register specified by bits [7:4] of an immediate byte operand.

When *src3* designates the same XMM register as the *dest* register, the XMM register behaves as an accumulator.

Instruction Support

Form	Subset	Feature Flag
VPMACSSDD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPMACSSDD *xmm1, xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	8E /r ib

Related Instructions

VPMACSSWW, VPMACSWW, VPMACSSWD, VPMACSWD, VPMACSDD, VPMACSSDQL, VPMACSSDQH, VPMACSDQL, VPMACSDQH, VPMADCSSWD, VPMADCSWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMACSSDQH Packed Multiply Accumulate with Saturation Signed High Doubleword to Signed Quadword

Multiplies the second 32-bit signed integer value of the first source by the corresponding value of the second source, then adds the low-order 64-bit signed integer value of the third source to the 64-bit signed integer product. Simultaneously, multiplies the fourth 32-bit signed integer value of the first source by the corresponding value of the second source, then adds the high-order 64-bit signed integer value of the third source to the 64-bit signed integer product. Writes two saturated sums to the destination.

Out of range results of the addition are saturated to fit into a signed 64-bit integer. For each packed value of the destination, when the value is larger than the largest signed 64-bit integer, it is saturated to 7FFF_FFFF_FFFF_FFFFh, and when the value is smaller than the smallest signed 64-bit integer, it is saturated to 8000_0000_0000_0000h.

There are four operands: VPMACSSDQH *dest, src1, src2, src3* *dest = src1* src2 + src3*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the destination XMM register is written, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by XOP.vvvv; the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field; and the third source (*src3*) is an XMM register specified by bits [7:4] of an immediate byte operand.

When *src3* designates the same XMM register as the *dest* register, the XMM register behaves as an accumulator.

Instruction Support

Form	Subset	Feature Flag
VPMACSSDQH	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPMACSSDQH *xmm1, xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	8F /r ib

Related Instructions

VPMACSSWW, VPMACSWW, VPMACSSWD, VPMACSWD, VPMACSSDD, VPMACSDD, VPMACSSDQL, VPMACSDQL, VPMACSDQH, VPMADCSSWD, VPMADCSWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMACSSDQL Packed Multiply Accumulate with Saturation Signed Low Doubleword to Signed Quadword

Multiplies the low-order 32-bit signed integer value of the first source by the corresponding value of the second source, then adds the low-order 64-bit signed integer value of the third source to the 64-bit signed integer product. Simultaneously, multiplies the third 32-bit signed integer value of the first source by the third 32-bit signed integer value of the second source, then adds the high-order 64-bit signed integer value of the third source to the 64-bit signed integer product. Writes two saturated sums to the destination.

Out of range results of the addition are saturated to fit into a signed 64-bit integer. For each packed value of the destination, when the value is larger than the largest signed 64-bit integer, it is saturated to 7FFF_FFFF_FFFF_FFFFh, and when the value is smaller than the smallest signed 64-bit integer, it is saturated to 8000_0000_0000_0000h.

There are four operands: VPMACSSDQL *dest, src1, src2, src3* *dest = src1* src2 + src3*

The destination (*dest*) register is an XMM register specified by ModRM.reg. When the destination is written, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by XOP.vvvv; the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field; and the third source (*src3*) is an XMM register specified by bits [7:4] of an immediate byte operand.

When *src3* designates the same XMM register as the *dest* register, the XMM register behaves as an accumulator.

Instruction Support

Form	Subset	Feature Flag
VPMACSSDQL	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPMACSSDQL *xmm1, xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	87 /r ib

Related Instructions

VPMACSSWW, VPMACSWW, VPMACSSWD, VPMACSWD, VPMACSSDD, VPMACSDD, VPMACSSDQH, VPMACSDQL, VPMACSDQH, VPMADCSSWD, VPMADCSWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMACSSWD Packed Multiply Accumulate with Saturation Signed Word to Signed Doubleword

Multiplies the odd-numbered packed 16-bit signed integer values of the first source by the corresponding values of the second source, then adds the corresponding packed 32-bit signed integer values of the third source to the 32-bit signed integer products. Writes four saturated sums to the destination.

Out of range results of the addition are saturated to fit into a signed 32-bit integer. For each packed value of the destination, when the value is larger than the largest signed 32-bit integer, it is saturated to 7FFF_FFFFh, and when the value is smaller than the smallest signed 32-bit integer, it is saturated to 8000_0000h.

There are four operands:

VPMACSSWD d*est, src1, src2, src3* *dest = src1* src2 + src3*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the destination XMM register is written, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by the XOP.vvvv field; the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field; and the third source (*src3*) is an XMM register specified by bits [7:4] of an immediate byte operand.

When *src3* designates the same XMM register as the *dest* register, the XMM register behaves as an accumulator.

Instruction Support

Form	Subset	Feature Flag
VPMACSSWD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPMACSSWD *xmm1, xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	86 /r ib

Related Instructions

VPMACSSWW, VPMACSWW, VPMACSWD, VPMACSSDD, VPMACSDD, VPMACSSDQL, VPMACSSDQH, VPMACSDQL, VPMACSDQH, VPMADCSSWD, VPMADCSWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMACSSWW Packed Multiply Accumulate with Saturation Signed Word to Signed Word

Multiplies each packed 16-bit signed integer value of the first source by the corresponding packed 16-bit signed integer value of the second source, then adds the corresponding packed 16-bit signed integer value of the third source to the 32-bit signed integer products. Writes eight saturated sums to the destination.

Out of range results of the addition are saturated to fit into a signed 16-bit integer. For each packed value of the destination, when the value is larger than the largest signed 16-bit integer, it is saturated to 7FFFh, and when the value is smaller than the smallest signed 16-bit integer, it is saturated to 8000h.

There are four operands:

VPMACSSWW *dest, src1, src2, src3* *dest = src1* src2 + src3*

The destination is an XMM register specified by ModRM.reg. When the destination is written, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by XOP.vvvv; the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field; and the third source (*src3*) is an XMM register specified by bits [7:4] of an immediate byte.

When *src3* and *dest* designate the same XMM register, this register behaves as an accumulator.

Instruction Support

Form	Subset	Feature Flag
VPMACSSWW	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPMACSSWW *xmm1, xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	85 /r ib

Related Instructions

VPMACSWW, VPMACSSWD, VPMACSWD, VPMACSSDD, VPMACSDD, VPMACSSDQL, VPMACSSDQH, VPMACSDQL,VPMACSDQH, VPMADCSSWD, VPMADCSWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMACSWD Packed Multiply Accumulate Signed Word to Signed Doubleword

Multiplies each odd-numbered packed 16-bit signed integer value of the first source by the corresponding value of the second source, then adds the corresponding packed 32-bit signed integer value of the third source to the 32-bit signed integer products. Writes four 32-bit results to the destination.

When the result of the add overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set). Only the low-order 32 bits of the result are written to the destination.

There are four operands: VPMACSWD *dest, src1, src2, src3* *dest = src1* src2 + src3*

The destination (*dest*) register is an XMM register specified by ModRM.reg. When the destination XMM register is written, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by XOP.vvvv; the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field; and the third source (*src3*) is an XMM register specified by bits [7:4] of an immediate byte operand.

When *src3* designates the same XMM register as the *dest* register, the XMM register behaves as an accumulator.

Instruction Support

Form	Subset	Feature Flag
VPMACSWD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPMACSWD *xmm1, xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	96 /r ib

Related Instructions

VPMACSSWW, VPMACSWW, VPMACSSWD, VPMACSSDD, VPMACSDO, VPMACSSDQL, VPMACSSDQH, VPMACSDQL, VPMACSDQH, VPMADCSSWD, VPMADCSWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMACSWW Packed Multiply Accumulate Signed Word to Signed Word

Multiplies each packed 16-bit signed integer value of the first source by the corresponding value of the second source, then adds the corresponding packed 16-bit signed integer value of the third source to each 32-bit signed integer product. Writes eight 16-bit results to the destination.

No saturation is performed on the sum. When the result of the multiplication causes non-zero values to be set in the upper 16 bits of the 32 bit result, they are ignored. When the result of the add overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set). In both cases, only the signed low-order 16 bits of the result are written to the destination.

There are four operands: VPMACSWW *dest, src1, src2, src3* *dest = src1* src2 + src3*

The destination (*dest*) is an XMM register specified by ModRM.reg. When the destination XMM register is written, bits [255:128] of the corresponding YMM register are cleared.

The first source (*src1*) is an XMM register specified by XOP.vvvv; the second source (*src2*) is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field; and the third source (*src3*) is an XMM register specified by bits [7:4] of an immediate byte operand.

When *src3* designates the same XMM register as the *dest* register, the XMM register behaves as an accumulator.

Instruction Support

Form	Subset	Feature Flag
VPMACSWW	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPMACSWW *xmm1, xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{\textit{src1}}$.0.00	95 /r ib

Related Instructions

VPMACSSWW, VPMACSSWD, VPMACSWD, VPMACSSDD, VPMACSDD, VPMACSSDQL, VPMACSSDQH, VPMACSDQL, VPMACSDQH, VPMADCSSWD, VPMADCSWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMADCSSWD Packed Multiply Add Accumulate with Saturation Signed Word to Signed Doubleword

Multiplies each packed 16-bit signed integer value of the first source by the corresponding value of the second source, then adds the 32-bit signed integer products of the even-odd adjacent words. Each resulting sum is then added to the corresponding packed 32-bit signed integer value of the third source. Writes four 32-bit signed-integer results to the destination.

Out of range results of the addition are saturated to fit into a signed 32-bit integer. For each packed value of the destination, when the value is larger than the largest signed 32-bit integer, it is saturated to 7FFF_FFFFh, and when the value is smaller than the smallest signed 32-bit integer, it is saturated to 8000_0000h.

There are four operands: VPMADCSSWD *dest, src1, src2, src3* *dest = src1* src2 + src3*

The destination is an XMM register specified by ModRM.reg. When the destination is written, bits [255:128] of the corresponding YMM register are cleared.

The first source is an XMM register specified by XOP.vvvv; the second source is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field; and the third source is an XMM register specified by bits [7:4] of an immediate byte operand.

When *src3* designates the same XMM register as the *dest* register, the XMM register behaves as an accumulator.

Instruction Support

Form	Subset	Feature Flag
VPMADCSSWD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPMADCSSWD *xmm1, xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{\textit{src1}}$.0.00	A6 /r ib

Related Instructions

VPMACSSWW, VPMACSWW, VPMACSSWD, VPMACSWD, VPMACSSDD, VPMACSDD, VPMACSSDQL, VPMACSSDQH, VPMACSDQL, VPMACSDQH, VPMADCSWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMADCSWD Packed Multiply Add Accumulate Signed Word to Signed Doubleword

Multiplies each packed 16-bit signed integer value of the first source by the corresponding value of the second source, then adds the 32-bit signed integer products of the even-odd adjacent words together and adds the sums to the corresponding packed 32-bit signed integer values of the third source. Writes four 32-bit sums to the destination.

No saturation is performed on the sum. When the result of the addition overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set). Only the signed 32-bits of the result are written to the destination.

There are four operands: VPMADCSWD *dest, src1, src2, src3* *dest = src1* src2 + src3*

The destination is an XMM register specified by ModRM.reg. When the destination is written, bits [255:128] of the corresponding YMM register are cleared.

The first source is an XMM register specified by XOP.vvvv, the second source is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field; and the third source is an XMM register specified by bits [7:4] of an immediate byte operand.

When *src3* designates the same XMM register as the *dest* register, the XMM register behaves as an accumulator.

Instruction Support

Form	Subset	Feature Flag
PMADCSWD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
PMADCSWD *xmm1, xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	B6 /r ib

Related Instructions

VPMACSSWW, VPMACSWW, VPMACSSWD, VPMACSWD, VPMACSSDD, VPMACSDD, VPMACSSDQL, VPMACSSDQH, VPMACSDQL, VPMACSDQH, VPMADCSSWD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPMASKMOVD — Masked Move Packed Doubleword

Moves packed doublewords from a second source operand to a destination, as specified by mask bits in a first source operand. There are load and store versions of the instruction.

The mask bits are the most-significant bit of each doubleword in the first source operand (mask).

- For loads, when a mask bit = 1, the corresponding doubleword is copied from the source to the same element of the destination; when a mask bit = 0, the corresponding element of the destination is cleared.
- For stores, when a mask bit = 1, the corresponding doubleword is copied from the source to the same element of the destination; when a mask bit = 0, the corresponding element of the destination is not affected.

Exception and trap behavior for elements not selected for loading or storing from/to memory is implementation dependent. For instance, a given implementation may signal a data breakpoint or a page fault for doublewords that are zero-masked and not actually written.

This instruction provides no non-temporal access hint.

This instruction has both 128-bit and 256-bit forms:

XMM Encoding

There are load and store encodings.

- For loads, the four doublewords that make up the source operand are located in a 128-bit memory location, the mask operand is an XMM register, and the destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.
- For stores, the four doublewords that make up the source operand are located in an XMM register, the mask operand is an XMM register, and the destination is a 128-bit memory location.

YMM Encoding

There are load and store encodings.

- For loads, the eight doublewords that make up the source operand are located in a 256-bit memory location, the mask operand is a YMM register, and the destination is a YMM register.
- For stores, the eight doublewords that make up the source operand are located in a YMM register, the mask operand is a YMM register, and the destination is a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
VPMASKMOVD	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
Loads:				
VPMASKMOVD *xmm1*, *xmm2*, *mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src1}}$.0.01	8C /r
VPMASKMOVD *ymm1*, *ymm2*, *mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src1}}$.1.01	8C /r
Stores:				
VPMASKMOVD *mem128, xmm1*, *xmm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src1}}$.0.01	8E /r
VPMASKMOVD *mem256, ymm1*, *ymm2*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src1}}$.1.01	8E /r

Related Instructions

VPMASKMOVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX2 exception				

VPMASKMOVQ Masked Move Packed Quadword

Moves packed quadwords from a second source operand to a destination, as specified by mask bits in a first source operand. There are load and store versions of the instruction.

The mask bits are the most-significant bit of each quadword in the mask first source operand (mask).

- For loads, when a mask bit = 1, the corresponding quadword is copied from the source to the same element of the destination; when a mask bit = 0, the corresponding element of the destination is cleared.
- For stores, when a mask bit = 1, the corresponding quadword is copied from the source to the same element of the destination; when a mask bit = 0, the corresponding element of the destination is not affected.

Exception and trap behavior for elements not selected for loading or storing from/to memory is implementation dependent. For instance, a given implementation may signal a data breakpoint or a page fault for quadwords that are zero-masked and not actually written.

This instruction provides no non-temporal access hint.

This instruction has both 128-bit and 256-bit forms:

XMM Encoding

There are load and store encodings.

- For loads, the two quadwords that make up the source operand are located in a 128-bit memory location, the mask operand is an XMM register, and the destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.
- For stores, the two quadwords that make up the source operand are located in an XMM register, the mask operand is an XMM register, and the destination is a 128-bit memory location.

YMM Encoding

There are load and store encodings.

- For loads, the four quadwords that make up the source operand are located in a 256-bit memory location, the mask operand is a YMM register, and the destination is a YMM register.
- For stores, the four quadwords that make up the source operand are located in a YMM register, the mask operand is a YMM register, and the destination is a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
VPMASKMOVQ	AVX2	Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
Loads:				
VPMASKMOVQ *xmm1*, *xmm2*, *mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src1}}$.0.01	8C /r
VPMASKMOVQ *ymm1*, *ymm2*, *mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src1}}$.1.01	8C /r
Stores:				
VPMASKMOVQ *mem128, xmm1*, *xmm2*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src1}}$.0.01	8E /r
VPMASKMOVQ *mem256, ymm1*, *ymm2*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src1}}$.1.01	8E /r

Related Instructions

VPMASKMOVD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX2 exception				

VPPERM Packed Permute Bytes

Selects 16 of 32 packed bytes from two concatenated sources, applies a logical transformation to each selected byte, then writes the byte to a specified position in the destination.

There are four operands: VPPERM *dest, src1, src2, src3*

The second (*src2*) and first (*src1*) sources are concatenated to form the 32-byte source.

The *src1* operand is an XMM register specified by XOP.vvvv.

The third source (*src3*) contains 16 control bytes. Each control byte specifies the source byte and the logical operation to perform on that byte. The order of the bytes in the destination is the same as that of the control bytes in the *src3*.

For each byte of the 16-byte result, the corresponding *src3* byte is used as follows:

- Bits [7:5] select a logical operation to perform on the selected byte.

Bit Value	Selected Operation
000	Source byte (no logical operation)
001	Invert source byte
010	Bit reverse of source byte
011	Bit reverse of inverted source byte
100	00h (zero-fill)
101	FFh (ones-fill)
110	Most significant bit of source byte replicated in all bit positions.
111	Invert most significant bit of source byte and replicate in all bit positions.

- Bits [4:0] select a source byte to move from *src2*:*src1*.

Bit Value	Source Byte	Bit Value	Source Byte	Bit Value	Source Byte	Bit Value	Source Byte
00000	*src1*[7:0]	01000	*src1*[71:64]	10000	*src2*[7:0]	11000	*src2*[71:64]
00001	*src1*[15:8]	01001	*src1*[79:72]	10001	*src2*[15:8]	11001	*src2*[79:72]
00010	*src1*[23:16]	01010	*src1*[87:80]	10010	*src2*[23:16]	11010	*src2*[87:80]
00011	*src1*[31:24]	01011	*src1*[95:88]	10011	*src2*[31:24]	11011	*src2*[95:88]
00100	*src1*[39:32]	01100	*src1*[103:96]	10100	*src2*[39:32]	11100	*src2*[103:96]
00101	*src1*[47:40]	01101	*src1*[111:104]	10101	*src2*[47:40]	11101	*src2*[111:104]
00110	*src1*[55:48]	01110	*src1*[119:112]	10110	*src2*[55:48]	11110	*src2*[119:112]
00111	*src1*[63:56]	01111	*src1*[127:120]	10111	*src2*[63:56]	11111	*src2*[127:120]

XOP.W and an immediate byte (*imm8*) determine register configuration.

- When XOP.W = 0, *src2* is either an XMM register or a 128-bit memory location specified by ModRM.r/m and *src3* is an XMM register specified by *imm8[7:4]*.

- When XOP.W = 1, *src2* is an XMM register specified by *imm8[7:4]* and *src3* is either an XMM register or a 128-bit memory location specified by ModRM.r/m.

The destination (*dest*) is an XMM register specified by ModRM.reg. When the result is written to the *dest* XMM register, bits [255:128] of the corresponding YMM register are cleared.

Instruction Support

Form	Subset	Feature Flag
VPPERM	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPPERM *xmm1*, *xmm2, xmm3/mem128, xmm4*	8F	$\overline{\text{RXB}}$.08	0.$\overline{src1}$.0.00	A3 /r ib
VPPERM *xmm1*, *xmm2, xmm3, xmm4/mem128*	8F	$\overline{\text{RXB}}$.08	1.$\overline{src1}$.0.00	A3 /r ib

Related Instructions

VPSHUFHW, VPSHUFD, VPSHUFLW, VPSHUFW, VPERMIL2PS, VPERMIL2PD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPROTB Packed Rotate Bytes

Rotates each byte of the source as specified by a count operand and writes the result to the corresponding byte of the destination.

There are two versions of the instruction, one for each source of the count byte:

- VPROTB *dest, src, fixed-count*
- VPROTB *dest, src, variable-count*

For both versions of the instruction, the destination (*dest*) operand is an XMM register specified by ModRM.reg.

The *fixed-count* version of the instruction rotates each byte of the source (*src*) the number of bits specified by the immediate *fixed-count* byte. All bytes are rotated the same amount. The source XMM register or memory location is selected by the ModRM.r/m field.

The *variable-count* version of the instruction rotates each byte of the source the amount specified in the corresponding byte element of the *variable-count*. Both *src* and *variable-count* are configured by XOP.W.

- When XOP.W = 0, *variable-count* is an XMM register specified by XOP.vvvv and *src* is either an XMM register or a 128-bit memory location specified by ModRM.r/m.
- When XOP.W = 1, *variable-count* is either an XMM register or a 128-bit memory location specified by ModRM.r/m and *src* is an XMM register specified by XOP.vvvv.

When the count value is positive, bits are rotated to the left (toward the more significant bit positions). The bits rotated out left of the most significant bit are rotated back in at the right end (least-significant bit) of the byte.

When the count value is negative, bits are rotated to the right (toward the least significant bit positions). The bits rotated to the right out of the least significant bit are rotated back in at the left end (most-significant bit) of the byte.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPROTB	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPROTB *xmm1*, *xmm2/mem128, xmm3*	8F	$\overline{\text{RXB}}$.09	0.$\overline{count}$.0.00	90 /r
VPROTB *xmm1*, *xmm2, xmm3/mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	90 /r
VPROTB *xmm1*, *xmm2/mem128*, *imm8*	8F	$\overline{\text{RXB}}$.08	0.1111.0.00	C0 /r ib

Related Instructions

VPROTW, VPROTD, VPROTQ,VPSHLB, VPSHLW, VPSHLD, VPSHLQ, VPSHAB, VPSHAW, VPSHAD, VPSHAQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.vvvv ! = 1111b (for immediate operand variant only)
			X	XOP.L field = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPROTD Packed Rotate Doublewords

Rotates each doubleword of the source as specified by a count operand and writes the result to the corresponding doubleword of the destination.

There are two versions of the instruction, one for each source of the count byte:

- VPROTD *dest, src, fixed-count*
- VPROTD *dest, src, variable-count*

For both versions of the instruction, the *dest* operand is an XMM register specified by ModRM.reg.

The fixed count version of the instruction rotates each doubleword of the source operand the number of bits specified by the immediate *fixed-count* byte operand. All doublewords are rotated the same amount. The *src* XMM register or memory location is selected by the ModRM.r/m field.

The variable count version of the instruction rotates each doubleword of the source by the amount specified in the low order byte of the corresponding doubleword of the *variable-count* operand vector.

Both *src* and *variable-count* are configured by XOP.W.

- When XOP.W = 0, *src* is either an XMM register or a128-bit memory location specified by the ModRM.r/m field and *variable-count* is an XMM register specified by XOP.vvvv.
- When XOP.W = 1, *src* is an XMM register specified by XOP.vvvv and *variable-count* is either an XMM register or a 128-bit memory location specified by the ModRM.r/m field.

When the count value is positive, bits are rotated to the left (toward the more significant bit positions). The bits rotated out to the left of the most significant bit of each source doubleword operand are rotated back in at the right end (least-significant bit) of the doubleword.

When the count value is negative, bits are rotated to the right (toward the least significant bit positions). The bits rotated to the right out of the least significant bit of each source doubleword operand are rotated back in at the left end (most-significant bit) of the doubleword.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPROTD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPROTD *xmm1*, *xmm2*/*mem128, xmm3*	8F	$\overline{\text{RXB}}$.09	0.$\overline{count}$.0.00	92 /r
VPROTD *xmm1*, *xmm2, xmm3*/*mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	92 /r
VPROTD *xmm1, xmm2*/*mem128*, *imm8*	8F	$\overline{\text{RXB}}$.08	0.1111.0.00	C2 /r ib

Related Instructions

VPROTB, VPROTW, VPROTQ, VPSHLB, VPSHLW, VPSHLD, VPSHLQ, VPSHAB, VPSHAW, VPSHAD, VPSHAQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.vvvv ! = 1111b (for immediate operand variant only)
			X	XOP.L field = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPROTQ Packed Rotate Quadwords

Rotates each quadword of the source operand as specified by a count operand and writes the result to the corresponding quadword of the destination.

There are two versions of the instruction, one for each source of the count byte:

- VPROTQ *dest, src, fixed-count*
- VPROTQ *dest, src, variable-count*

For both versions of the instruction, the *dest* operand is an XMM register specified by ModRM.reg.

The fixed count version of the instruction rotates each quadword in the source the number of bits specified by the immediate *fixed-count* byte operand. All quadword elements of the source are rotated the same amount. The *src* XMM register or memory location is selected by the ModRM.r/m field.

The variable count version of the instruction rotates each quadword of the source the amount specified ny the low order byte of the corresponding quadword of the *variable-count* operand.

Both *src* and *variable-count* are configured by XOP.W.

- When XOP.W = 0, *src* is either an XMM register or a 128-bit memory location specified by ModRM.r/m and *variable-count* is an XMM register specified by XOP.vvvv.
- When XOP.W = 1, *src* is an XMM register specified by XOP.vvvv and *variable-count* is either an XMM register or a128-bit memory location specified by ModRM.r/m.

When the count value is positive, bits are rotated to the left (toward the more significant bit positions) of the operand element. The bits rotated out to the left of the most significant bit of the word element are rotated back in at the right end (least-significant bit).

When the count value is negative, operand element bits are rotated to the right (toward the least significant bit positions). The bits rotated to the right out of the least significant bit are rotated back in at the left end (most-significant bit) of the word element.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPROTQ	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPROTQ *xmm1*, *xmm2*/*mem128, xmm3*	8F	$\overline{\text{RXB}}$.09	0.$\overline{count}$.0.00	93 /r
VPROTQ *xmm1*, *xmm2, xmm3*/*mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	93 /r
VPROTQ *xmm1, xmm2*/*mem128*, *imm8*	8F	$\overline{\text{RXB}}$.08	0.1111.0.00	C3 /r ib

Related Instructions

VPROTB, VPROTW, VPROTD, VPSHLB, VPSHLW, VPSHLD, VPSHLQ, VPSHAB, VPSHAW, VPSHAD, VPSHAQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.vvvv ! = 1111b (for immediate operand variant only)
			X	XOP.L field = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPROTW Packed Rotate Words

Rotates each word of the source as specified by a count operand and writes the result to the corresponding word of the destination.

There are two versions of the instruction, one for each source of the count byte:

- VPROTW dest, *src*, *fixed-count*
- VPROTW dest, *src*, *variable-count*

For both versions of the instruction, the *dest* operand is an XMM register specified by ModRM.reg.

The fixed count version of the instruction rotates each word of the source the number of bits specified by the immediate *fixed-count* byte operand. All words of the source operand are rotated the same amount. The *src* XMM register or memory location is selected by the ModRM.r/m field.

The variable count version of this instruction rotates each word of the source operand by the amount specified in the low order byte of the corresponding word of the *variable-count* operand.

Both *src* and *variable-count* are configured by XOP.W.

- When XOP.W = 0, *src* is either an XMM register or a 128-bit memory location specified by ModRM.r/m and *variable-count* is an XMM register specified by XOP.vvvv.
- When XOP.W = 1, *src* is an XMM register specified by XOP.vvvv and *variable-count* is either an XMM register or a 128-bit memory location specified by ModRM.r/m.

When the count value is positive, bits are rotated to the left (toward the more significant bit positions). The bits rotated out to the left of the most significant bit of an element are rotated back in at the right end (least-significant bit) of the word element.

When the count value is negative, bits are rotated to the right (toward the least significant bit positions) of the element. The bits rotated to the right out of the least significant bit of an element are rotated back in at the left end (most-significant bit) of the word element.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPROTW	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPROTW *xmm1*, *xmm2*/*mem128, xmm3*	8F	$\overline{\text{RXB}}$.09	0.$\overline{count}$.0.00	91 /r
VPROTW *xmm1*, *xmm2, xmm3*/*mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	91 /r
VPROTW *xmm1, xmm2*/*mem128*, *imm8*	8F	$\overline{\text{RXB}}$.08	0.1111.0.00	C1 /r ib

Related Instructions

VPROTB, VPROTD, VPROTQ, VPSHLB, VPSHLW, VPSHLD, VPSHLQ, VPSHAB, VPSHAW, VPSHAD, VPSHAQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.vvvv ! = 1111b (for immediate operand variant only)
			X	XOP.L field = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPSHAB Packed Shift Arithmetic Bytes

Shifts each signed byte of the source as specified by a count byte and writes the result to the corresponding byte of the destination.

The count bytes are 8-bit signed two's-complement values in the corresponding bytes of the *count* operand.

When the count value is positive, bits are shifted to the left (toward the more significant bit positions). Zeros are shifted in at the right end (least-significant bit) of the byte.

When the count value is negative, bits are shifted to the right (toward the least significant bit positions). The most significant bit (sign bit) is replicated and shifted in at the left end (most-significant bit) of the byte.

There are three operands: VPSHAB *dest*, *src*, *count*

The destination (*dest*) is an XMM register specified by ModRM.reg.

Both *src* and *count* are configured by XOP.W.

- When XOP.W = 0, *count* is an XMM register specified by XOP.vvvv and *src* is either an XMM register or a128-bit memory location specified by ModRM.r/m.
- When XOP.W = 1, *count* is either an XMM register or a 128-bit memory location specified by ModRM.r/m and *src* is an XMM register specified by XOP.vvvv.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPSHAB	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPSHAB *xmm1*, *xmm2*/*mem128, xmm3*	8F	$\overline{\text{RXB}}$.09	0.$\overline{count}$.0.00	98 /r
VPSHAB *xmm1*, *xmm2, xmm3*/*mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	98 /r

Related Instructions

VPROTB, VPROTW, VPROTD, VPROTQ, VPSHLB, VPSHLW, VPSHLD, VPSHLQ, VPSHAW, VPSHAD, VPSHAQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPSHAD Packed Shift Arithmetic Doublewords

Shifts each signed doubleword of the source operand as specified by a count byte and writes the result to the corresponding doubleword of the destination.

The count bytes are 8-bit signed two's-complement values located in the low-order byte of the corresponding doubleword of the *count* operand.

When the count value is positive, bits are shifted to the left (toward the more significant bit positions). Zeros are shifted in at the right end (least-significant bit) of the doubleword.

When the count value is negative, bits are shifted to the right (toward the least significant bit positions). The most significant bit (sign bit) is replicated and shifted in at the left end (most-significant bit) of the doubleword.

There are three operands: VPSHAD *dest, src, count*

The destination (*dest*) is an XMM register specified by ModRM.reg.

Both *src* and *count* are configured by XOP.W.

- When XOP.W = 0, *count* is an XMM register specified by XOP.vvvv and *src* is either an XMM register or a memory location specified by ModRM.r/m.
- When XOP.W = 1, *count* is either an XMM register or a memory location specified by ModRM.r/m and *src* is an XMM register specified by XOP.vvvv.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPSHAD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPSHAD *xmm1*, *xmm2/mem128, xmm3*	8F	$\overline{\text{RXB}}$.09	0.$\overline{count}$.0.00	9A /r
VPSHAD *xmm1*, *xmm2, xmm3/mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	9A /r

Related Instructions

VPROTB, VPROTW, VPROTD, VPROTQ, VPSHLB, VPSHLW, VPSHLD, VPSHLQ, VPSHAB, VPSHAW, VPSHAQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPSHAQ Packed Shift Arithmetic Quadwords

Shifts each signed quadword of the source as specified by a count byte and writes the result to the corresponding quadword of the destination.

The count bytes are 8-bit signed two's-complement values located in the low-order byte of the corresponding quadword element of the *count* operand.

When the count value is positive, bits are shifted to the left (toward the more significant bit positions). Zeros are shifted in at the right end (least-significant bit) of the quadword.

When the count value is negative, bits are shifted to the right (toward the least significant bit positions). The most significant bit is replicated and shifted in at the left end (most-significant bit) of the quadword.

The shift amount is stored in two’s-complement form. The count is modulo 64.

There are three operands: VPSHAQ *dest*, *src*, *count*

The destination (*dest*) is an XMM register specified by ModRM.reg.

Both *src* and *count* are configured by XOP.W.

- When XOP.W = 0, *count* is an XMM register specified by XOP.vvvv and *src* is either an XMM register or a memory location specified by ModRM.r/m.
- When XOP.W = 1, *count* is either an XMM register or a memory location specified by ModRM.r/m and *src* is an XMM register specified by XOP.vvvv.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPSHAQ	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPSHAQ *xmm1*, *xmm2*/*mem128, xmm3*	8F	$\overline{\text{RXB}}$.09	0.$\overline{count}$.0.00	9B /r
VPSHAQ *xmm1*, *xmm2, xmm3*/*mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	9B /r

Related Instructions

VPROTB, VPROTW, VPROTD, VPROTQ, VPSHLB, VPSHLW, VPSHLD, VPSHLQ, VPSHAB, VPSHAW, VPSHAD

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPSHAW Packed Shift Arithmetic Words

Shifts each signed word of the source as specified by a count byte and writes the result to the corresponding word of the destination.

The count bytes are 8-bit signed two's-complement values located in the low-order byte of the corresponding word of the *count* operand.

When the count value is positive, bits are shifted to the left (toward the more significant bit positions). Zeros are shifted in at the right end (least-significant bit) of the word.

When the count value is negative, bits are shifted to the right (toward the least significant bit positions). The most significant bit (signed bit) is replicated and shifted in at the left end (most-significant bit) of the word.

The shift amount is stored in two’s-complement form. The count is modulo 16.

There are three operands: VPSHAW *dest, src, count*

The destination (*dest*) is an XMM register specified by ModRM.reg.

Both *src* and *count* are configured by XOP.W.

- When XOP.W = 0, *count* is an XMM register specified by XOP.vvvv and *src* is either an XMM register or a memory location specified by ModRM.r/m.
- When XOP.W = 1, *count* is either an XMM register or a memory location specified by ModRM.r/m and *src* is an XMM register specified by XOP.vvvv.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPSHAW	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPSHAW *xmm1*, *xmm2*/*mem128, xmm3*	8F	$\overline{\text{RXB}}$.09	0.$\overline{count}$.0.00	99 /r
VPSHAW *xmm1*, *xmm2, xmm3*/*mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	99 /r

Related Instructions

VPROTB, VPROTW, VPROTD, VPROTQ, VPSHLB, VPSHLW, VPSHLD, VPSHLQ, VPSHAB, VPSHAD, VPSHAQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPSHLB Packed Shift Logical Bytes

Shifts each packed byte of the source as specified by a count byte and writes the result to the corresponding byte of the destination.

The count bytes are 8-bit signed two's-complement values located in the corresponding byte element of the *count* operand.

When the count value is positive, bits are shifted to the left (toward the more significant bit positions). Zeros are shifted in at the right end (least-significant bit) of the byte.

When the count value is negative, bits are shifted to the right (toward the least significant bit positions). Zeros are shifted in at the left end (most-significant bit) of the byte.

There are three operands: VPSHLB *dest*, *src*, *count*

The destination (*dest*) is an XMM register specified by ModRM.reg.

Both *src* and *count* are configured by XOP.W.

- When XOP.W = 0, *count* is an XMM register specified by XOP.vvvv and *src* is either an XMM register or a memory location specified by ModRM.r/m.
- When XOP.W = 1, *count* is either an XMM register or a memory location specified by ModRM.r/m and *src* is an XMM register specified by XOP.vvvv.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPSHLB	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPSHLB *xmm1*, *xmm2/mem128, xmm3*	8F	$\overline{\text{RXB}}$.09	0.$\overline{count}$.0.00	94 /r
VPSHLB *xmm1*, *xmm2, xmm3/mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	94 /r

Related Instructions

VPROTB, VPROTW, VPROTD, VPROTQ, VPSHLW, VPSHLD, VPSHLQ, VPSHAB, VPSHAW, VPSHAD, VPSHAQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPSHLD Packed Shift Logical Doublewords

Shifts each doubleword of the source operand as specified by a count byte and writes the result to the corresponding doubleword of the destination.

The count bytes are 8-bit signed two's-complement values located in the low-order byte of the corresponding doubleword element of the *count* operand.

When the count value is positive, bits are shifted to the left (toward the more significant bit positions). Zeros are shifted in at the right end (least-significant bit) of the doubleword.

When the count value is negative, bits are shifted to the right (toward the least significant bit positions). Zeros are shifted in at the left end (most-significant bit) of the doubleword.

The shift amount is stored in two’s-complement form. The count is modulo 32.

There are three operands: VPSHLD *dest, src, count*

The destination (*dest*) is an XMM register specified by ModRM.reg.

Both *src* and *count* are configured by XOP.W.

- When XOP.W = 0, *count* is an XMM register specified by XOP.vvvv and *src* is either an XMM register or a memory location specified by ModRM.r/m.
- When XOP.W = 1, *count* is either an XMM register or a memory location specified by ModRM.r/m and *src* is an XMM register specified by XOP.vvvv.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPSHLD	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPSHLD *xmm1*, *xmm3*/*mem128, xmm2*	8F	$\overline{\text{RXB}}$.09	0.$\overline{count}$.0.00	96 /r
VPSHLD *xmm1*, *xmm2, xmm3*/*mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	96 /r

Related Instructions

VPROTB, VPROTW, VPROTD, VPROTQ, VPSHLB, VPSHLW, VPSHLQ, VPSHAB, VPSHAW, VPSHAD, VPSHAQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPSHLQ Packed Shift Logical Quadwords

Shifts each quadwords of the source by as specified by a count byte and writes the result in the corresponding quadword of the destination.

The count bytes are 8-bit signed two's-complement values located in the low-order byte of the corresponding quadword element of the *count* operand.

Bit 6 of the count byte is ignored.

When the count value is positive, bits are shifted to the left (toward the more significant bit positions). Zeros are shifted in at the right end (least-significant bit) of the quadword.

When the count value is negative, bits are shifted to the right (toward the least significant bit positions). Zeros are shifted in at the left end (most-significant bit) of the quadword.

There are three operands: VPSHLQ *dest, src, count*

The destination (*dest*) is an XMM register specified by ModRM.reg.

Both *src* and *count* are configured by XOP.W.

- When XOP.W = 0, *count* is an XMM register specified by XOP.vvvv and *src* is either an XMM register or a memory location specified by ModRM.r/m.
- When XOP.W = 1, *count* is either an XMM register or a memory location specified by ModRM.r/m and *src* is an XMM register specified by XOP.vvvv.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPSHLQ	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPSHLQ *xmm1*, *xmm3/mem128, xmm2*	8F	$\overline{\text{RXB}}$.09	0.$\overline{count}$.0.00	97 /r
VPSHLQ *xmm1*, *xmm2, xmm3/mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	97 /r

Related Instructions

VPROTB, VPROTW, VPROTD, VPROTQ, VPSHLB, VPSHLW, VPSHLD, VPSHAB, VPSHAW, VPSHAD, VPSHAQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPSHLW Packed Shift Logical Words

Shifts each word of the source operand as specified by a count byte and writes the result to the corresponding word of the destination.

The count bytes are 8-bit signed two's-complement values located in the low-order byte of the corresponding word element of the *count* operand.

When the count value is positive, bits are shifted to the left (toward the more significant bit positions). Zeros are shifted in at the right end (least-significant bit) of the word.

When the count value is negative, bits are shifted to the right (toward the least significant bit positions). Zeros are shifted in at the left end (most-significant bit) of the word.

There are three operands: VPSHLW *dest, src, count*

The destination (*dest*) is an XMM register specified by ModRM.reg.

Both *src* and *count* are configured by XOP.W.

- When XOP.W = 0, *count* is an XMM register specified by XOP.vvvv and *src* is either an XMM register or a memory location specified by ModRM.r/m.
- When XOP.W = 1, *count* is either an XMM register or a memory location specified by ModRM.r/m and *src* is an XMM register specified by XOP.vvvv.

Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
VPSHLW	XOP	CPUID Fn8000_0001_ECX[XOP] (bit 11)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VPSHLW *xmm1*, *xmm3/mem128, xmm2*	8F	$\overline{\text{RXB}}$.09	0.$\overline{count}$.0.00	95 /r
VPSHLW *xmm1*, *xmm2, xmm3/mem128*	8F	$\overline{\text{RXB}}$.09	1.$\overline{src}$.0.00	95 /r

Related Instructions

VPROTB, VPROLW, VPROTD, VPROTQ, VPSHLB, VPSHLD, VPSHLQ, VPSHAB, VPSHAW, VPSHAD, VPSHAQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

VPSLLVD Variable Shift Left Logical Doublewords

Left-shifts the bits of each doubleword in the first source operand by a count specified in the corresponding doubleword of a second source operand and writes the shifted values to the destination.

The second source operand is treated as an array of unsigned 32-bit integers. Each integer specifies the shift count of the corresponding doubleword of the first source operand. Each doubleword is shifted independently.

Low-order bits emptied by shifting are cleared. High-order bits shifted out of each doubleword are discarded. When the shift count for any doubleword is greater than 31, that doubleword is cleared in the destination.

This instruction has 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The shift count array is specified by either a second XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The shift count array is specified by either a second YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VPSLLVD	AVX2	CPUID Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSLLVD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.0.01	47 /r
VPSLLVD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.1.01	47 /r

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLQ, (V)PSLLW, (V)PSRAD, (V)PSRAW, (V)PSRLD, (V)PSRLDQ, (V)PSRLQ, (V)PSRLW, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX2 exception				

VPSLLVQ Variable Shift Left Logical Quadwords

Left-shifts the bits of each quadword in the first source operand by a count specified in the corresponding quadword of a second source operand and writes the shifted values to the destination.

The second source operand is treated as an array of unsigned 64-bit integers. Each integer specifies the shift count of the corresponding quadword of the first source operand. Each quadword is shifted independently.

Low-order bits emptied by shifting are cleared. High-order bits shifted out of each quadword are discarded. When the shift count for any quadword is greater than 63, that quadword is cleared in the destination.

This instruction has 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The shift count array is specified by either a second XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The shift count array is specified by either a second YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VPSLLVQ	AVX2	CPUID Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSLLVQ *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src1}$.0.01	47 /r
VPSLLVQ *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src1}$.1.01	47 /r

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLQ, (V)PSLLW, (V)PSRAD, (V)PSRAW, (V)PSRLD, (V)PSRLDQ, (V)PSRLQ, (V)PSRLW, VPSLLVD, VPSRAVD, VPSRLVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX2 exception				

VPSRAVD Variable Shift Right Arithmetic Doublewords

Performs a right arithmetic shift of each signed 32-bit integer in the first source operand by a count specified in the corresponding doubleword of a second source operand and writes the shifted values to the destination.

The second source operand is treated as an array of unsigned 32-bit integers. Each integer specifies the shift count of the corresponding doubleword of the first source operand. Each doubleword is shifted independently.

A copy of the sign bit is shifted into the most-significant bit of the element on each right-shift. Low-order bits shifted out of each element are discarded. If a doubleword contains a positive integer and the shift count is greater than 31, that doubleword is cleared in the destination. If a doubleword contains a negative integer and the shift count is greater than 31, that doubleword is set to -1 in the destination.

This instruction has 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The shift count array is specified by either a second XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The shift count array is specified by either a second YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VPSRAVD	AVX2	CPUID Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSRAVD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.0.01	46 /r
VPSRAVD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.1.01	46 /r

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLQ, (V)PSLLW, (V)PSRAD, (V)PSRAW, (V)PSRLD, (V)PSRLDQ, (V)PSRLQ, (V)PSRLW, VPSLLVD, VPSLLVQ, VPSRLVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX2 exception				

VPSRLVD Variable Shift Right Logical Doublewords

Right-shifts each doubleword in the first source operand by a count specified in the corresponding doubleword of a second source operand and writes the shifted values to the destination.

The second source operand is treated as an array of unsigned 32-bit integers. Each integer specifies the shift count of the corresponding doubleword of the first source operand. Each doubleword is shifted independently.

Zero is shifted into the most-significant bit of the element on each right-shift. Low-order bits shifted out of each element are discarded. If the shift count for any doubleword is greater than 31, that doubleword is cleared in the destination.

This instruction has 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The shift count array is specified by either a second XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The shift count array is specified by either a second YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VPSRLVD	AVX2	CPUID Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSRLVD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.0.01	45 /r
VPSRLVD *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src1}$.1.01	45 /r

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLQ, (V)PSLLW, (V)PSRAD, (V)PSRAW, (V)PSRLD, (V)PSRLDQ, (V)PSRLQ, (V)PSRLW, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX2 exception				

VPSRLVQ Variable Shift Right Logical Quadwords

Right-shifts each quadword in the first source operand by a count specified in the corresponding quadword of a second source operand and writes the shifted values to the destination.

The second source operand is treated as an array of unsigned 64-bit integers. Each integer specifies the shift count of the corresponding quadword of the first source operand. Each quadword is shifted independently.

Zero is shifted into the most-significant bit of the element on each right-shift. Low-order bits shifted out of each element are discarded. If the shift count for any quadword is greater than 63, that quadword is cleared in the destination.

This instruction has 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The shift count array is specified by either a second XMM register or a 128-bit memory location. The destination is an XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register. The shift count array is specified by either a second YMM register or a 256-bit memory location. The destination is a YMM register.

Instruction Support

Form	Subset	Feature Flag
VPSRLVQ	AVX2	CPUID Fn0000_00007_EBX[AVX2]_x0 (bit 5)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VPSRLVQ *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src1}$.0.01	45 /r
VPSRLVQ *ymm1*, *ymm2*, *ymm3/mem256*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src1}$.1.01	45 /r

Related Instructions

(V)PSLLD, (V)PSLLDQ, (V)PSLLQ, (V)PSLLW, (V)PSRAD, (V)PSRAW, (V)PSRLD, (V)PSRLDQ, (V)PSRLQ, (V)PSRLW, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX2 exception				

VTESTPD Packed Bit Test

Performs two different logical operations on the sign bits of the first and second packed floating-point operands and updates the ZF and CF flags based on the results.

First, performs a bitwise AND of the sign bits of each double-precision floating-point element of the first source operand with the sign bits of the corresponding elements of the second source operand. Sets rFLAGS.ZF when all bit operations = 0; else, clears ZF.

Second, performs a bitwise AND of the complements (NOT) of the sign bits of each double-precision floating-point element of the first source with the sign bits of the corresponding elements of the second source operand. Sets rFLAGS.CF when all bit operations = 0; else, clears CF.

Neither source operand is modified.

This extended-form instruction has both 128-bit and 256-bit encoding.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
VTESTPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VTESTPD *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	0.1111.0.01	0F /r
VTESTPD *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.02	0.1111.1.01	0F /r

Related Instructions

PTEST, VTESTPS

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3 and 1 are reserved. A flag set or cleared is M (modified). Unaffected flags are blank. Undefined flags are U.*

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		AVX instructions are only recognized in protected mode.
	X	X	X	CR0.EM = 1.
	X	X	X	CR4.OSFXSR = 0.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	VEX.W = 1.
			X	VEX.vvvv ! = 1111b.
			X	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	X	X	X	CR0.TS = 1.
Stack, #SS	X	X	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	X	X	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		X	X	Instruction execution caused a page fault.

X — AVX exception

VTESTPS Packed Bit Test

Performs two different logical operations on the sign bits of the first and second packed floating-point operands and updates the ZF and CF flags based on the results.

First, performs a bitwise AND of the sign bits of each single-precision floating-point element of the first source operand with the sign bits of the corresponding elements of the second source operand. Sets rFLAGS.ZF when all bit operations = 0; else, clears ZF.

Second, performs a bitwise AND of the complements (NOT) of the sign bits of each single-precision floating-point element of the first source with the sign bits of the corresponding elements of the second source operand. Sets rFLAGS.CF when all bit operations = 0; else, clears CF.

Neither source operand is modified.

This extended-form instruction has both 128-bit and 256-bit encoding.

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location.

YMM Encoding

The first source operand is a YMM register. The second source operand is either a YMM register or a 256-bit memory location.

Instruction Support

Form	Subset	Feature Flag
VTESTPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VTESTPS *xmm1*, *xmm2/mem128*	C4	$\overline{\text{RXB}}$.02	0.1111.0.01	0E /r
VTESTPS *ymm1*, *ymm2/mem256*	C4	$\overline{\text{RXB}}$.02	0.1111.1.01	0E /r

Related Instructions

PTEST, VTESTPD

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								0				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3 and 1 are reserved. A flag set or cleared is M (modified). Unaffected flags are blank. Undefined flags are U.*

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		AVX instructions are only recognized in protected mode.
	X	X	X	CR0.EM = 1.
	X	X	X	CR4.OSFXSR = 0.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	VEX.W = 1.
			X	VEX.vvvv ! = 1111b.
			X	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	X	X	X	CR0.TS = 1.
Stack, #SS	X	X	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	X	X	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		X	X	Instruction execution caused a page fault.

X — AVX exception

VZEROALL — Zero All YMM Registers

Clears all YMM registers.

In 64-bit mode, YMM0–15 are all cleared (set to all zeros). In legacy and compatibility modes, only YMM0–7 are cleared. The contents of the MXCSR is unaffected.

Instruction Support

Form	Subset	Feature Flag
VZEROALL	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VZEROALL	C4	$\overline{\text{RXB}}$.01	X.1111.1.00	77

Related Instructions

VZEROUPPER

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
A — AVX exception.				

VZEROUPPER Zero All YMM Registers Upper

Clears the upper octword of all YMM registers. The corresponding XMM registers (lower octword of each YMM register) are not affected.

In 64-bit mode, the instruction operates on registers YMM0–15. In legacy and compatibility mode, the instruction operates on YMM0–7. The contents of the MXCSR is unaffected.

Instruction Support

Form	Subset	Feature Flag
VZEROUPPER	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VZEROUPPER	C4	$\overline{\text{RXB}}$.01	X.1111.0.00	77

Related Instructions

VZEROUPPER

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
A — AVX exception.				

XGETBV Get Extended Control Register Value

Copies the content of the extended control register (XCR) specified by the ECX register into the EDX:EAX register pair. The high-order 32 bits of the XCR are loaded into EDX and the low-order 32 bits are loaded into EAX. The corresponding high-order 32 bits of RAX and RDX are cleared.

This instruction and associated data structures extend the FXSAVE/FXRSTOR memory image used to manage processor states and provide additional functionality. See the XSAVE instruction description for more information.

Values returned to EDX:EAX in unimplemented bit locations are undefined.

Specifying a reserved or unimplemented XCR in ECX causes a general protection exception.

Currently, only XCR0 (the XFEATURE_ENABLED_MASK register) is supported. If CPUID reports support for ECX=1 (see table below), then the XGETBV instruction supports an ECX value of 1. When ECX=1, XGETBV returns the logical and of XCR0 and the current value of the XINUSE state-component bitmap.

Instruction Support

Form	Subset	Feature Flag
XGETBV	XSAVE/XRSTOR	CPUID Fn0000_0001_ECX[XSAVE] (bit 26)
XGETBV	ECX=1 support	CPUID Fn0000_000D_EAX_x1[2] = 1

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
XGETBV	0F 01 D0	Copies content of the XCR specified by ECX into EDX:EAX.

Related Instructions

XSETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X	X	Lock prefix (F0h) preceding opcode.
	X	X	X	CR4.OSXSAVE = 0
General protection, #GP	X	X	X	ECX specifies a reserved or unimplemented XCR address.
X — exception generated				

XORPD VXORPD XOR Packed Double-Precision Floating-Point

Performs bitwise XOR of two packed double-precision floating-point values in the first source operand with the corresponding values of the second source operand and writes the results into the corresponding elements of the destination.

There are legacy and extended forms of the instruction:

XORPD

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VXORPD

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
XORPD	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VXORPD	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
XORPD *xmm1*, *xmm2/mem128*	66 0F 57 /r	Performs bitwise XOR of two packed double-precision floating-point values in *xmm1* with corresponding values in *xmm2* or *mem128*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VXORPD *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.01	57 /r
VXORPD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.01	57 /r

Related Instructions

(V)ANDNPS, (V)ANDPD, (V)ANDPS, (V)ORPD, (V)ORPS, (V)XORPS

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

XORPS XOR
VXORPS Packed Single-Precision Floating-Point

Performs bitwise XOR of four packed single-precision floating-point values in the first source operand with the corresponding values of the second source operand and writes the results into the corresponding elements of the destination.

There are legacy and extended forms of the instruction:

XORPS

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The first source register is also the destination. Bits [255:128] of the YMM register that corresponds to the destination are not affected.

VXORPS

The extended form of the instruction has both 128-bit and 256-bit encodings:

XMM Encoding

The first source operand is an XMM register. The second source operand is either an XMM register or a 128-bit memory location. The destination is a third XMM register. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

YMM Encoding

The first source operand is a YMM register and the second source operand is either a YMM register or a 256-bit memory location. The destination is a third YMM register.

Instruction Support

Form	Subset	Feature Flag
XORPS	SSE2	CPUID Fn0000_0001_EDX[SSE2] (bit 26)
VXORPS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
XORPS *xmm1*, *xmm2/mem128*	66 0F 57 /r	Performs bitwise XOR of four packed single-precision floating-point values in *xmm1* with corresponding values in *xmm2* or *mem128*. Writes the result to *xmm1*.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VXORPS *xmm1*, *xmm2*, *xmm3/mem128*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.0.00	57 /r
VXORPS *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.01	X.$\overline{src1}$.1.00	57 /r

Related Instructions

(V)ANDNPS, (V)ANDPD, (V)ANDPS, (V)ORPD, (V)ORPS, (V)XORPD

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

XRSTOR Restore Extended States

Restores a selected set of enabled processor state data from a save area at a specified address in memory. This instruction and associated data structures extend the FXSAVE/FXRSTOR memory image used to manage processor state and provide additional functionality. See the description of the XSAVE instruction for basic operational details.

The XRSTOR instruction may operate on the save area in standard form or a compact form. The compact form is indicated in the save area with XCOMP_BV[63]=1.

In either form, the instruction creates a Requested Feature Bit Map (RFBM) which is the logical AND of EDX:EAX and XCR0 which selects the processor components to be operated on. Then for each feature bit:

1. If RFBM = 0, XRSTOR does not update the component.
2. If RFBM = 1 but the corresponding XSTATE_BV bit is 0, the component is set to its reset state without reading anything out of the save area.
3. If RFBM =1 and XSTATE_BV =1, the component state is read from the save area.
4. XRSTOR loads an internal state value XRSTOR_INFO that can be used to further optimize a subsequent XSAVEOPT or XSAVES. This reflects the current privilege level and virtualization mode as well as the save area's base address and XCOMP_BV field.
5. If RFBM=1, the corresponding XINUSE bit is set to the state of XSTATE_BV.

For standard mode, MXCSR is loaded if RFBM[1]=1 or RFBM[2]=1. It is never initialized.

For compact mode, MXCSR is associated with RFBM[1].

In some generations, the FP error pointers were only restored if there was a Floating Point error logged. In newer generations, the FP error pointers are always restored. This is indicated by CPUID Fn8000_0008_EBX[2].

Refer to Volume 2, Section 11.5, "XSAVE/XRSTOR Instructions" for other operational detail including save area formats.

Instruction Support

Form	Subset	Feature Flag
XRSTOR	XRSTOR	CPUID Fn0000_00001_ECX[XSAVE] (bit 26)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
XRSTOR *mem*	0F AE /5	Restores user-specified processor state from memory.

Related Instructions

XGETBV, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, XSETBV

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X	X	CR4.OSXSAVE = 0.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	X	X	X	CR0.TS = 1.
Stack, #SS	X	X	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	X	X	X	Memory address exceeding data segment limit or non-canonical.
	X	X	X	Null data segment used to reference memory.
	X	X	X	Memory operand not aligned on 64-byte boundary.
	X	X	X	Any must be zero (MBZ) bits in the save area were set.
	X	X	X	Attempt to set reserved bits in MXCSR.
	X	X	X	XCOMP_BV[i] = 0 & XSTATE_BV[i] = 1
	X	X	X	XCOMP_BV[I] = 1 & XCR0[i] = 0
	X	X	X	Bytes 63:16 of header are non-zero
Page fault, #PF	X	X	X	Instruction execution caused a page fault.
X — exception generated				

XRSTORS Restore Extended States Supervisor

Restores a selected set of enabled processor state data from a save area at a specified address in memory, optionally including privileged state.

XRSTORS is very similar to the XRSTOR instruction in compacted form with the following differences:

1. XRSTORS must be executed at CPL=0
2. XRSTORS must read XCOMP_BV[63]=1, otherwise it will cause a #GP(0) exception
3. XRSTORS is able to restore state enabled from the IA32_XSS MSR.

All other behavior is the same as XRSTOR with the compact form.

Instruction Support

Form	Subset	Feature Flag
XRSTOR	XRSTOR	CPUID Fn0000_00001_ECX_X1[XSAVES] (bit 3)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
XRSTORS *mem*	0F C7 /3	Restores selected processor state from memory

Related Instructions

XGETBV, XRSTOR, XSAVE, XSAVEC, XSAVEOPT, XSAVES, XSETBV

rFLAGS Affected

None

MXCSR Flags Affected

None

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X	X	CR4.OSXSAVE = 0.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	X	X	X	CR0.TS = 1.
Stack, #SS	X	X	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	X	X	X	Memory address exceeding data segment limit or non-canonical.
	X	X	X	Null data segment used to reference memory.
	X	X	X	Memory operand not aligned on 64-byte boundary.
	X	X	X	Any must be zero (MBZ) bits in the save area were set.
	X	X	X	Attempt to set reserved bits in MXCSR.
	X	X	X	CPL != 0
	X	X	X	(XSTATE_BV[i] & ~IA321_XSS[i]) = 1
Page fault, #PF	X	X	X	Instruction execution caused a page fault.
X — exception generated				

XSAVE Save Extended States

Saves a selected set of enabled processor state data to a save area at a specified memory address.

This instruction and associated data structures extend the FXSAVE/FXRSTOR memory image used to manage numeric coprocessor state and provide additional functionality.

The XSAVE/XRSTOR save area consists of a header section, and individual save areas for each processor state component. A component is saved when both the corresponding bits in the mask operand (EDX:EAX) and the XFEATURE_ENABLED_MASK (XCR0) register are set. This bit-wise logical AND of EDX:EAX and XCR0 is known as the Requested Feature Bit Map (RFBM). A component is not saved when its corresponding RFBM bit is zero.

Software can set any bit in EDX:EAX, regardless of whether the bit position in XCR0 is valid for the processor. When the mask operand contains all 1's, all processor state components enabled in XCR0 are saved.

For each component saved, XSAVE sets the corresponding bit in the XSTATE_BV field of the save area header. XSAVE does not clear XSTATE_BV bits or modify individual save areas for components that are not saved. If a saved component is in the hardware-specified initialized state, XSAVE may clear the corresponding XSTATE_BV bit instead of setting it. This optimization is implementation-dependent.

The MXCSR register is saved if either of RFBM bits 0 or 1 are set to 1. If there is no floating point error present, some generations would not write out any of the FP error pointers. On newer generations, these fields are written to zeros. This is indicated by CPUID Fn8000_0008_EBX[2].

Refer to Volume 2, Section 11.5, "XSAVE/XRSTOR Instructions" for other operational detail including save area formats.

Instruction Support

Form	Subset	Feature Flag
XSAVE	XSAVE/XRSTOR	CPUID Fn0000_0001_ECX[XSAVE] (bit 26)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
XSAVE *mem*	0F AE /4	Saves selected processor state to memory.

Related Instructions

XGETBV, XRSTOR, XRSTORS, XSAVEC, XSAVEOPT, XSAVES, XSETBV

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X	X	CR4.OSXSAVE = 0.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	X	X	X	CR0.TS = 1.
Stack, #SS	X	X	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	X	X	X	Memory address exceeding data segment limit or non-canonical.
	X	X	X	Null data segment used to reference memory.
	X	X	X	Memory operand not aligned on 64-byte boundary.
	X	X	X	Attempt to write read-only memory.
Page fault, #PF	X	X	X	Instruction execution caused a page fault.
X — exception generated				

XSAVEC Save Extended States, Compacted

Saves a selected set of enabled processor state data to a save area at a specified memory address, possibly in a compacted form.

This instruction and associated data structures extend the FXSAVE/FXRSTOR memory image used to manage processor states and provides compaction functionality for more efficient context switching. See the XSAVE and XRSTOR instruction descriptions for basic operational details..

XSAVEC is very similar to XSAVE but provides the following alternate functionality:

1. XSAVEC differs from XSAVE by using the init optimization and compaction.
2. XSAVEC differs by only saving a component if its RFBM=1 and its XINUSE=1. XINUSE is a means by which the processor determines whether the feature is in its Initial state.
3. XSAVEC never writes bytes 511:464 of the legacy XSAVE data structure.
4. XSAVEC calculates XSTATE_BV by performing the logical AND of the RFBM and XINUSE bitmaps and writes it to the XSAVE area.
5. XSAVEC calculates XCOMP_BV as [63]=1 and 62:0 = RFBM, and writes it to the XSAVE area.
6. XSAVEC does not modify any other parts of the header except as indicated in 4 and 5.
7. XSAVEC uses the compacted format of the XSAVE extended region while saving state.

Instruction Support

Form	Subset	Feature Flag
XSAVE mem	XSAVEC	CPUID Fn0000_0000D_EAX_x1[XSAVEC] (bit 1)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
XSAVEC *mem*	0F C7 /4	Saves selected processor state to memory.

Related Instructions

XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEOPT, XSAVES, XSETBV

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X	X	CR4.OSXSAVE = 0.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	X	X	X	CR0.TS = 1.
Stack, #SS	X	X	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	X	X	X	Memory address exceeding data segment limit or non-canonical.
	X	X	X	Null data segment used to reference memory.
	X	X	X	Memory operand not aligned on 64-byte boundary.
	X	X	X	Attempt to write read-only memory.
Page fault, #PF	X	X	X	Instruction execution caused a page fault.
X — exception generated				

XSAVEOPT Save Extended States, Performance Optimized

Saves a selected set of enabled processor state data to a save area at a specified memory address.

This instruction and associated data structures extend the FXSAVE/FXRSTOR memory image used to manage processor states and provide additional functionality. See the XSAVE and XRSTOR instruction descriptions for basic operational details.

The XSAVE/XRSTOR save area consists of a header section, and individual save areas for each processor state component. A component is saved when both the corresponding bits in the mask operand (EDX:EAX) and the XFEATURE_ENABLED_MASK (XCR0) register are set. A component is not saved when either of the corresponding bits in EDX:EAX or XCR0 is cleared.

Software can set any bit in EDX:EAX, regardless of whether the bit position in XCR0 is valid for the processor. When the mask operand contains all 1's, all processor state components enabled in XCR0 are saved.

For each component saved, XSAVEOPT sets the corresponding bit in the XSTATE_BV field of the save area header. XSAVEOPT does not clear XSTATE_BV bits or modify individual save areas for components that are not saved. If a saved component is in the hardware-specified initialized state, XSAVEOPT may clear the corresponding XSTATE_BV bit instead of setting it. This optimization is implementation-dependent.

XSAVEOPT may provide other implementation-specific optimizations, such as the *modified* optimization described for XSAVES.

Instruction Support

Form	Subset	Feature Flag
XSAVEOPT	XSAVEOPT	CPUID Fn0000_0000D_EAX_x1[XSAVEOPT] (bit 0)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
XSAVEOPT *mem*	0F AE /6	Saves selected processor state to memory.

Related Instructions

XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVES, XSETBV

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X	X	CR4.OSXSAVE = 0.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	X	X	X	CR0.TS = 1.
Stack, #SS	X	X	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	X	X	X	Memory address exceeding data segment limit or non-canonical.
	X	X	X	Null data segment used to reference memory.
	X	X	X	Memory operand not aligned on 64-byte boundary.
	X	X	X	Attempt to write read-only memory.
Page fault, #PF	X	X	X	Instruction execution caused a page fault.
X — exception generated				

XSAVES Save Extended States Supervisor

Saves a selected set of enabled processor state data to a save area at a specified memory address, possibly in a compacted form, and optionally including privileged state.

This instruction and associated data structures extend the XSAVE/XRSTOR memory image used to manage processor states and provides compaction functionality. See the XSAVE and XRSTOR instruction descriptions for basic operational details.

XSAVES is very similar to XSAVEC but provides the following alternate functionality:

1. XSAVES must be executed at CPL=0
2. XSAVES can save state enabled in the IA32_XSS MSR. The specific state elements saved are determined by the logical AND of EDX:EAX with the logical OR of XCR0 with the IA32_XSS MSR.
3. XSAVES can use the *modified* optimization to not save components, even if RFBM=1 and XINUSE=1 for the stated component. If the component state has not been modified internally since the last execution of XRSTOR or XRSTORS and the XRSTOR_INFO state (an execution environment signature created by the last XRSTOR) matches the current execution state of this XSAVES, the state save can be skipped.

Instruction Support

Form	Subset	Feature Flag
XSAVES	XSAVES	CPUID Fn0000_0000D_EAX_x1[XSAVES] (bit 3)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
XSAVES mem	0F C7 /5	Saves selected processor state to memory

Related Instructions

XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSETBV

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X	X	CR4.OSXSAVE = 0.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	X	X	X	CR0.TS = 1.
Stack, #SS	X	X	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	X	X	X	Memory address exceeding data segment limit or non-canonical.
	X	X	X	Null data segment used to reference memory.
	X	X	X	Memory operand not aligned on 64-byte boundary.
	X	X	X	Attempt to write read-only memory.
Page fault, #PF	X	X	X	Instruction execution caused a page fault.
X — exception generated				

XSETBV Set Extended Control Register Value

Writes the content of the EDX:EAX register pair into the extended control register (XCR) specified by the ECX register. The high-order 32 bits of the XCR are loaded from EDX and the low-order 32 bits are loaded from EAX. The corresponding high-order 32 bits of RAX and RDX are ignored.

This instruction and associated data structures extend the FXSAVE/FXRSTOR memory image used to manage processor states and provide additional functionality. See the XSAVE instruction description for more information.

Currently, only the XFEATURE_ENABLED_MASK register (XCR0) is supported. Specifying a reserved or unimplemented XCR in ECX causes a general protection exception (#GP).

Executing XSETBV at a privilege level other than 0 causes a general-protection exception. A general protection exception also occurs when software attempts to write to reserved bits of an XCR.

Instruction Support

Form	Subset	Feature Flag
XSETBV	XSAVE/XRSTOR	CPUID Fn0000_0001_ECX[XSAVE] (bit 26)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

Mnemonic	Opcode	Description
XSETBV	0F 01 D1	Writes the content of the EDX:EAX register pair to the XCR specified by the ECX register.

Related Instructions

XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X		X	Instruction not supported, as indicated by CPUID feature identifier.
	X		X	CR4.OSXSAVE = 0.
	X		X	Lock prefix (F0h) preceding opcode.

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
General protection, #GP		X	X	CPL != 0.
	X		X	ECX specifies a reserved or unimplemented XCR address.
	X		X	Any must be zero (MBZ) bits in the XCR were set.
	X		X	Setting XCR0[2:1] to 10b.
	X		X	Writing 0 to XCR[0].
X — exception generated				

3 Exception Summary

This chapter provides a ready reference to instruction exceptions. Table 3-1 shows instructions grouped by exception class, with the extended and legacy instruction type (if applicable).

Hyperlinks in the table point to the exception tables which follow.

Table 3-1. Instructions By Exception Class

Mnemonic	Extended Type	Legacy Type
Class 1 — AVX / SSE Vector Aligned (VEX.vvvv != 1111)		
MOVAPD VMOVAPD	AVX	SSE2
MOVAPS VMOVAPS	AVX	SSE
MOVDQA VMOVDQA	AVX	SSE2
MOVNTDQ VMOVNTDQ	AVX	SSE2
MOVNTPD VMOVNTPD	AVX	SSE2
MOVNTPS VMOVNTPS	AVX	SSE
Class 1X — SSE / AXV / AVX2 Vector (VEX.vvvv != 1111b or VEX.L=1 && !AVX2)		
MOVNTDQA VMOVNTDQA	AVX, AVX2	SSE4.1
Class 2 — AVX / SSE Vector (SIMD 111111)		
DIVPD VDIVPD	AVX	SSE2
DIVPS VDIVPS	AVX	SSE
Class 2-1 — AVX / SSE Vector (SIMD 111011)		
ADDPD VADDPD	AVX	SSE2
ADDPS VADDPS	AVX	SSE
ADDSUBPD VADDSUBPD	AVX	SSE2
ADDSUBPS VADDSUBPS	AVX	SSE
DPPS VDPPS	AVX	SSE4.1
HADDPD VHADDPD	AVX	SSE3
HADDPS VHADDPS	AVX	SSE3
HSUBPD VHSUBPD	AVX	SSE3
HSUBPS VHSUBPS	AVX	SSE3
SUBPD VSUBPD	AVX	SSE2
SUBPS VSUBPS	AVX	SSE
Class 2-2 — AVX / SSE Vector (SIMD 000011)		
CMPPD VCMPPD	AVX	SSE2
CMPPS VCMPPS	AVX	SSE
MAXPD VMAXPD	AVX	SSE2
MAXPS VMAXPS	AVX	SSE
MINPD VMINPD	AVX	SSE2
MINPS VMINPS	AVX	SSE
MULPD VMULPD	AVX	SSE2
MULPS VMULPS	AVX	SSE
Class 2-3 — AVX / SSE Vector (SIMD 100001)		
(unused)	—	—

Table 3-1. Instructions By Exception Class (continued)

Mnemonic	Extended Type	Legacy Type
Class 2A — AVX / SSE Vector (SIMD 111111, VEX.L = 1)		
(unused)	—	—
Class 2A-1 — AVX / SSE Vector (SIMD 111011, VEX.L = 1)		
DPPD VDPPD	AVX	SSE4.1
Class 2B — AVX / SSE Vector (SIMD 111111, VEX.vvvv != 1111b)		
(unused)	—	—
Class 2B-1 — AVX / SSE Vector (SIMD 100000, VEX.vvvv != 1111b)		
CVTDQ2PS VCVTDQ2PS	AVX	SSE2
Class 2B-2 — AVX / SSE Vector (SIMD 100001, VEX.vvvv != 1111b)		
CVTPD2DQ VCVTPD2DQ	AVX	SSE2
CVTPS2DQ VCVTPS2DQ	AVX	SSE2
CVTTPS2DQ VCVTTPS2DQ	AVX	SSE2
CVTTPD2DQ VCVTTPD2DQ	AVX	SSE2
ROUNDPD, VROUNDPD	AVX	SSE4.1
ROUNDPS, VROUNDPS	AVX	SSE4.1
Class 2B-3 — AVX / SSE Vector (SIMD 111011, VEX.vvvv != 1111b)		
CVTPD2PS VCVTPD2PS	AVX	SSE2
Class 2B-4 — AVX / SSE Vector (SIMD 100011, VEX.vvvv != 1111b)		
SQRTPD VSQRTPD	AVX	SSE2
SQRTPS VSQRTPS	AVX	SSE
Class 3 — AVX / SSE Scalar (SIMD 111111)		
DIVSD VDIVSD	AVX	SSE2
DIVSS VDIVSS	AVX	SSE
Class 3-1 — AVX / SSE Scalar (SIMD 111011)		
ADDSD VADDSD	AVX	SSE2
ADDSS VADDSS	AVX	SSE
CVTSD2SS VCVTSD2SS	AVX	SSE2
SUBSD VSUBSD	AVX	SSE2
SUBSS VSUBSS	AVX	SSE
Class 3-2 — AVX / SSE Scalar (SIMD 000011)		
CMPSD VCMPSD	AVX	SSE2
CMPSS VCMPSS	AVX	SSE
CVTSS2SD VCVTSS2SD	AVX	SSE2
MAXSD VMAXSD	AVX	SSE2
MAXSS VMAXSS	AVX	SSE
MINSD VMINSD	AVX	SSE2
MINSS VMINSS	AVX	SSE
MULSD VMULSD	AVX	SSE2
MULSS VMULSS	AVX	SSE
UCOMISD VUCOMISD	AVX	SSE2
UCOMISS VUCOMISS	AVX	SSE

Table 3-1. Instructions By Exception Class (continued)

Mnemonic	Extended Type	Legacy Type
Class 3-3 — AVX / SSE Scalar (SIMD 100000)		
CVTSI2SD VCVTSI2SD	AVX	SSE2
CVTSI2SS VCVTSI2SS	AVX	SSE
Class 3-4 — AVX / SSE Scalar (SIMD 100001)		
ROUNDSD, VROUNDSD	AVX	SSE4.1
ROUNDSS, VROUNDSS	AVX	SSE4.1
Class 3-5 — AVX / SSE Scalar (SIMD 100011)		
SQRTSD VSQRTSD	AVX	SSE2
SQRTSS VSQRTSS	AVX	SSE
Class 3A — AVX / SSE Scalar (SIMD 111111, VEX.vvvv != 1111b)		
(unused)	—	—
Class 3A-1 — AVX / SSE Scalar (SIMD 000011, VEX.vvvv != 1111b)		
COMISD VCOMISD	AVX	SSE2
COMISS VCOMISS	AVX	SSE
CVTPS2PD VCVTPS2PD	AVX	SSE2
Class 3A-2 — AVX / SSE Scalar (SIMD 100001, VEX.vvvv != 1111b)		
CVTSD2SI VCVTSD2SI	AVX	SSE2
CVTSS2SI VCVTSS2SI	AVX	SSE
CVTTSD2SI VCVTTSD2SI	AVX	SSE2
CVTTSS2SI VCVTTSS2SI	AVX	SSE
Class 4 — AVX / SSE Vector		
AESDEC VAESDEC	AVX	AES
AESDECLAST VAESDECLAST	AVX	AES
AESENC VAESENC	AVX	AES
AESENCLAST VAESENCLAST	AVX	AES
AESIMC VAESIMC	AVX	AES
AESKEYGENASSIST VAESKEYGENASSIST	AVX	AES
ANDNPD VANDNPD	AVX	SSE2
ANDNPS VANDNPS	AVX	SSE
ANDPD VANDPD	AVX	SSE2
ANDPS VANDPS	AVX	SSE
BLENDPD VBLENDPD	AVX	SSE4.1
BLENDPS VBLENDPS	AVX	SSE4.1
ORPD VORPD	AVX	SSE2
ORPS VORPS	AVX	SSE
PCLMULQDQ VPCLMULQDQ	AVX	CLMUL
SHUFPD VSHUFPD	AVX	SSE2
SHUFPS VSHUFPS	AVX	SSE2
UNPCKHPD VUNPCKHPD	AVX	SSE2
UNPCKHPS VUNPCKHPS	AVX	SSE
UNPCKLPD VUNPCKLPD	AVX	SSE2
UNPCKLPS VUNPCKLPS	AVX	SSE

Table 3-1. Instructions By Exception Class (continued)

Mnemonic	Extended Type	Legacy Type
XORPD VXORPD	AVX	SSE2
XORPS VXORPS	AVX	SSE
Class 4A — AVX / SSE Vector (VEX.W = 1)		
BLENDVPD VBLENDVPD	AVX	SSE4.1
BLENDVPS VBLENDVPS	AVX	SSE4.1
Class 4B — AVX / SSE Vector (VEX.L = 1)		
(unused)	—	—
Class 4B-X — SSE / AVX / AVX2 (VEX.L = 1 && !AVX2)		
MPSADBW VMPSADBW	AVX, AVX2	SSE4.1
PACKSSDW VPACKSSDW	AVX, AVX2	SSE2
PACKSSWB VPACKSSWB	AVX, AVX2	SSE2
PACKUSDW VPACKUSDW	AVX, AVX2	SSE4.1
PACKUSWB VPACKUSWB	AVX, AVX2	SSE2
PADDB VPADDB	AVX, AVX2	SSE2
PADDD VPADDD	AVX, AVX2	SSE2
PADDQ VPADDQ	AVX, AVX2	SSE2
PADDSB VPADDSB	AVX, AVX2	SSE2
PADDSW VPADDSW	AVX, AVX2	SSE2
PADDUSB VPADDUSB	AVX, AVX2	SSE2
PADDUSW VPADDUSW	AVX, AVX2	SSE2
PADDW VPADDW	AVX, AVX2	SSE2
PALIGNR VPALIGNR	AVX, AVX2	SSSE3
PAND VPAND	AVX, AVX2	SSE2
PANDN VPANDN	AVX, AVX2	SSE2
PAVGB VPAVGB	AVX, AVX2	SSE
PAVGW VPAVGW	AVX, AVX2	SSE
PBLENDW VPBLENDW	AVX, AVX2	SSE4.1
PCMPEQB VPCMPEQB	AVX, AVX2	SSE2
PCMPEQD VPCMPEQD	AVX, AVX2	SSE2
PCMPEQQ VPCMPEQQ	AVX, AVX2	SSE4.1
PCMPEQW VPCMPEQW	AVX, AVX2	SSE2
PCMPGTB VPCMPGTB	AVX, AVX2	SSE2
PCMPGTD VPCMPGTD	AVX, AVX2	SSE2
PCMPGTQ VPCMPGTQ	AVX, AVX2	SSE4.2
PCMPGTW VPCMPGTW	AVX, AVX2	SSE2
PHADDD VPHADDD	AVX, AVX2	SSSE3
PHADDSW VPHADDSW	AVX, AVX2	SSSE3
PHADDW VPHADDW	AVX, AVX2	SSSE3
PHSUBD VPHSUBD	AVX, AVX2	SSSE3
PHSUBW VPHSUBW	AVX, AVX2	SSSE3
PHSUBSW VPHSUBSW	AVX, AVX2	SSSE3
PMADDUBSW VPMADDUBSW	AVX, AVX2	SSSE3

Table 3-1. Instructions By Exception Class (continued)

Mnemonic	Extended Type	Legacy Type
PMADDWD VPMADDWD	AVX, AVX2	SSE2
PMAXSB VPMAXSB	AVX, AVX2	SSE4.1
PMAXSD VPMAXSD	AVX, AVX2	SSE4.1
PMAXSW VPMAXSW	AVX, AVX2	SSE
PMAXUB VPMAXUB	AVX, AVX2	SSE
PMAXUD VPMAXUD	AVX, AVX2	SSE4.1
PMAXUW VPMAXUW	AVX, AVX2	SSE4.1
PMINSB VPMINSB	AVX, AVX2	SSE4.1
PMINSD VPMINSD	AVX, AVX2	SSE4.1
PMINSW VPMINSW	AVX, AVX2	SSE
PMINUB VPMINUB	AVX, AVX2	SSE
PMINUD VPMINUD	AVX, AVX2	SSE4.1
PMINUW VPMINUW	AVX, AVX2	SSE4.1
PMULDQ VPMULDQ	AVX, AVX2	SSE4.1
PMULHRSW VPMULHRSW	AVX, AVX2	SSSE3
PMULHUW VPMULHUW	AVX, AVX2	SSE2
PMULHW VPMULHW	AVX, AVX2	SSE2
PMULLD VPMULLD	AVX, AVX2	SSE4.1
PMULLW VPMULLW	AVX, AVX2	SSE2
PMULUDQ VPMULUDQ	AVX, AVX2	SSE2
POR VPOR	AVX, AVX2	SSE2
PSADBW VPSADBW	AVX, AVX2	SSE
PSHUFB VPSHUFB	AVX, AVX2	SSSE3
PSIGNB VPSIGNB	AVX, AVX2	SSSE3
PSIGND VPSIGND	AVX, AVX2	SSSE3
PSIGNW VPSIGNW	AVX, AVX2	SSSE3
PSUBB VPSUBB	AVX, AVX2	SSE2
PSUBD VPSUBD	AVX, AVX2	SSE2
PSUBQ VPSUBQ	AVX, AVX2	SSE2
PSUBSB VPSUBSB	AVX, AVX2	SSE2
PSUBSW VPSUBSW	AVX, AVX2	SSE2
PSUBUSB VPSUBUSB	AVX, AVX2	SSE2
PSUBUSW VPSUBUSW	AVX, AVX2	SSE2
PSUBW VPSUBW	AVX, AVX2	SSE2
PUNPCKHBW VPUNPCKHBW	AVX, AVX2	SSE2
PUNPCKHDQ VPUNPCKHDQ	AVX, AVX2	SSE2
PUNPCKHQDQ VPUNPCKHQDQ	AVX, AVX2	SSE2
PUNPCKHWD VPUNPCKHWD	AVX, AVX2	SSE2
PUNPCKLBW VPUNPCKLBW	AVX, AVX2	SSE2
PUNPCKLDQ VPUNPCKLDQ	AVX, AVX2	SSE2
PUNPCKLQDQ VPUNPCKLQDQ	AVX, AVX2	SSE2
PUNPCKLWD VPUNPCKLWD	AVX, AVX2	SSE2

Table 3-1. Instructions By Exception Class (continued)

Mnemonic	Extended Type	Legacy Type
PXOR VPXOR	AVX, AVX2	SSE2
Class 4C — AVX / SSE Vector (VEX.vvvv != 1111b)		
MOVSHDUP VMOVSHDUP	AVX	SSE3
MOVSLDUP VMOVSLDUP	AVX	SSE3
PTEST VPTEST	AVX	SSE4.1
RCPPS VRCPPS	AVX	SSE
RSQRTPS VRSQRTPS	AVX	SSE
Class 4C-1 — AVX / SSE Vector (write to RO memory, VEX.vvvv != 1111b)		
LDDQU VLDDQU	AVX	SSE3
MOVDQU VMOVDQU	AVX	SSE2
MOVUPD VMOVUPD	AVX	SSE2
MOVUPS VMOVUPS	AVX	SSE
Class 4D — AVX / SSE Vector (VEX.vvvv != 1111b, VEX.L = 1)		
MASKMOVDQU VMASKMOVDQU	AVX	SSE2
PCMPESTRI VPCMPESTRI	AVX	SSE4.2
PCMPESTRM VPCMPESTRM	AVX	SSE4.2
PCMPISTRI VPCMPISTRI	AVX	SSE4.2
PCMPISTRM VPCMPISTRM	AVX	SSE4.2
PHMINPOSUW VPHMINPOSUW	AVX	SSE4.1
Class 4D-X — SSE / AVX / AVX2 Vector (VEX.vvvv != 1111b, (VEX.L = 1 && !AVX2))		
PABSB VPABSB	AVX, AVX2	SSSE3
PABSD VPABSD	AVX, AVX2	SSSE3
PABSW VPABSW	AVX, AVX2	SSSE3
PSHUFD VPSHUFD	AVX, AVX2	SSE2
PSHUFHW VPSHUFHW	AVX, AVX2	SSE2
PSHUFLW VPSHUFLW	AVX, AVX2	SSE2
Class 4E — AVX / SSE Vector (VEX.W = 1, VEX.L = 1)		
(unused)	—	—
Class 4E-X — SSE / AVX / AVX2 Vector (VEX.W = 1, (VEX.L = 1 && !AVX2))		
PBLENDVB VPBLENDVB	AVX	SSE4.1
Class 4F — AVX / SSE (VEX.L = 1)		
(unused)	—	—
Class 4F-X — SSE / AVX / AVX2 Vector (VEX.L = 1 && !AVX2)		
PSLLD VPSLLD	AVX, AVX2	SSE2
PSLLQ VPSLLQ	AVX, AVX2	SSE2
PSLLW VPSLLW	AVX, AVX2	SSE2
PSRAD VPSRAD	AVX, AVX2	SSE2
PSRAW VPSRAW	AVX, AVX2	SSE2
PSRLD VPSRLD	AVX, AVX2	SSE2
PSRLQ VPSRLQ	AVX, AVX2	SSE2
PSRLW VPSRLW	AVX, AVX2	SSE2
Class 4G — AVX Vector (VEX.W = 1, VEX.vvvv != 1111b)		

Table 3-1. Instructions By Exception Class (continued)

Mnemonic	Extended Type	Legacy Type
VTESTPD	AVX	—
VTESTPS	AVX	—
Class 4H — AVX, 256-bit only (VEX.L = 0; No SIMD Exceptions)		
VPERMD	AVX2	—
VPERMPS	AVX2	—
Class 4H-1 — AVX2, 256-bit only (VEX.L = 0, VEX.vvvv != 1111b)		
VPERMPD	AVX2	—
VPERMQ	AVX2	—
Class 4J — AVX2 (VEX.W = 1)		
VPBLENDD	AVX2	—
VPSRAVD	AVX2	—
Class 4K — AVX2		
VPMASKMOVD	AVX2	—
VPMASKMOVQ	AVX2	—
VPSLLVD	AVX2	—
VPSLLVQ	AVX2	—
VPSRLVD	AVX2	—
VPSRLVQ	AVX2	—
Class 5 — AVX / SSE Scalar		
RCPSS VRCPSS	AVX	SSE
RSQRTSS VRSQRTSS	AVX	SSE
Class 5A — AVX / SSE Scalar (VEX.L = 1)		
INSERTPS VINSERTPS	AVX	SSE4.1
Class 5B — AVX / SSE Scalar (VEX.vvvv != 1111b)		
CVTDQ2PD VCVTDQ2PD	AVX	SSE2
MOVDDUP VMOVDDUP	AVX	SSE3
Class 5C — AVX /SSE Scalar (VEX.vvvv != 1111b, VEX.L = 1)		
PINSRB VPINSRB	AVX	SSE4.1
PINSRD VPINSRD	AVX	SSE4.1
PINSRQ VPINSRQ	AVX	SSE4.1
PINSRW VPINSRW	AVX	SSE
Class 5C-X — SSE / AVX / AVX2 Scalar (VEX.vvvv != 1111b, (VEX.L = 1 && !AVX2))		
PMOVSXBD VPMOVSXBD	AVX, AVX2	SSE4.1
PMOVSXBQ VPMOVSXBQ	AVX, AVX2	SSE4.1
PMOVSXBW VPMOVSXBW	AVX, AVX2	SSE4.1
PMOVSXDQ VPMOVSXDQ	AVX, AVX2	SSE4.1
PMOVSXWD VPMOVSXWD	AVX, AVX2	SSE4.1
PMOVSXWQ VPMOVSXWQ	AVX, AVX2	SSE4.1
PMOVZXBD VPMOVZXBD	AVX, AVX2	SSE4.1
PMOVZXBQ VPMOVZXBQ	AVX, AVX2	SSE4.1
PMOVZXBW VPMOVZXBW	AVX, AVX2	SSE4.1
PMOVZXDQ VPMOVZXDQ	AVX, AVX2	SSE4.1

Table 3-1. Instructions By Exception Class (continued)

Mnemonic	Extended Type	Legacy Type
PMOVZXWD VPMOVZXWD	AVX, AVX2	SSE4.1
PMOVZXWQ VPMOVZXWQ	AVX, AVX2	SSE4.1
Class 5C-1 — AVX / SSE Scalar (write to RO memory, VEX.vvvv != 1111b, VEX.L = 1)		
EXTRACTPS VEXTRACTPS	AVX	SSE4.1
MOVD VMOVD	AVX	SSE2
MOVQ VMOVQ	AVX	SSE2
PEXTRB VPEXTRB	AVX	SSE4.1
PEXTRD VPEXTRD	AVX	SSE4.1
PEXTRQ VPEXTRQ	AVX	SSE4.1
PEXTRW VPEXTRW	AVX	SSE4.1
Class 5D — AVX / SSE Scalar (write to RO memory, VEX.vvvv != 1111b (variant))		
MOVSD VMOVSD	AVX	SSE2
MOVSS VMOVSS	AVX	SSE
Class 5E — AVX / SSE Scalar (write to RO, VEX.vvvv != 1111b (variant), VEX.L = 1)		
MOVHPD VMOVHPD	AVX	SSE2
MOVHPS VMOVHPS	AVX	SSE
MOVLPD VMOVLPD	AVX	SSE2
MOVLPS VMOVLPS	AVX	SSE
Class 6 — AVX Mixed Memory Argument		
(unused)	—	—
Class 6A — AVX Mixed Memory Argument (VEX.W = 1)		
(unused)	—	—
Class 6A-1 — AVX Mixed Memory Argument (write to RO memory, VEX.W = 1)		
VMASKMOVPD	AVX	—
VMASKMOVPS	AVX	—
Class 6B — AVX Mixed Memory Argument (VEX.W = 1, VEX.L = 0)		
VINSERTF128	AVX	—
VINSERTI128	AVX2	—
VPERM2F128	AVX	—
VPERM2I128	AVX2	—
Class 6B-1 — AVX Mixed Memory Argument (write to RO, VEX.W = 1, VEX.L = 0)		
VEXTRACTF128	AVX	—
Class 6C — AVX Mixed Memory Argument (VEX.W = 1, VEX.L = 0, VEX.vvvv != 1111b)		
VBROADCASTF128	AVX	—
VBROADCASTI128	AVX2	—
VEXTRACTI128	AVX2	—
Class 6C-X — AVX / AVX2 (W=1, vvvv!=1111b, L=0, (reg src op specified && !AVX2))		
VBROADCASTSD	AVX, AVX2	—
Class 6D — AVX Mixed Memory Argument (VEX.W = 1, VEX.vvvv != 1111b)		
VPBROADCASTB	AVX2	—
VPBROADCASTD	AVX2	—
VPBROADCASTQ	AVX2	—

Table 3-1. Instructions By Exception Class (continued)

Mnemonic	Extended Type	Legacy Type
VPBROADCASTW	AVX2	—
Class 6D-X — AVX / AVX2 (W = 1, vvvv != 1111b, (ModRM.mod = 11b && !AVX2))		
VBROADCASTSS	AVX, AVX2	—
Class 6E — AVX Mixed Memory Argument (VEX.W = 1, VEX.vvvv != 1111b (variant))		
VPERMILPD	AVX	—
VPERMILPS	AVX	—
Class 6F — AVX2 (VEX.W = 1, VEX.vvvv != 1111b, VEX.L = 0, ModRM.mod = 11b)		
VBROADCASTI128	AVX2	—
Class 7 — AVX / SSE No Memory Argument		
(unused)	—	—
Class 7A — AVX /SSE No Memory Argument (VEX.L = 1)		
MOVHLPS VMOVHLPS	AVX	SSE
MOVLHPS VMOVLHPS	AVX	SSE
Class 7A-X SSE / AVX / AVX2 Vector (VEX.L = 1 && !AVX2)		
PSLLDQ VPSLLDQ	AVX, AVX2	SSE2
PSRLDQ VPSRLDQ	AVX, AVX2	SSE2
Class 7B — AVX /SSE No Memory Argument (VEX.vvvv != 1111b)		
MOVMSKPD VMOVMSKPD	AVX	SSE2
MOVMSKPS VMOVMSKPS	AVX	SSE
Class 7C — AVX / SSE No Memory Argument (VEX.vvvv != 1111b, VEX.L = 1)		
(not used)	—	—
Class 7C-X SSE / AVX / AVX2 Vector (VEX.vvvv != 1111b, (VEX.L = 1 && !AVX2))		
PMOVMSKB VPMOVMSKB	AVX, AVX2	SSE2
Class 8 — AVX No Memory Argument (VEX.vvvv != 1111b, VEX.W = 1)		
VZEROALL	AVX	—
VZEROUPPER	AVX	—
Class 9 — AVX 4-byte Argument (write to RO memory, VEX.vvvv != 1111b, VEX.L = 1)		
STMXCSR VSTMXCSR	AVX	SSE
Class 9A — AVX 4-byte argument (reserved MBZ = 1, VEX.vvvv != 1111b, VEX.L = 1)		
LDMXCSR VLDMXCSR	AVX	SSE

Table 3-1. Instructions By Exception Class (continued)

Mnemonic	Extended Type	Legacy Type
Class 10 — XOP Base		
VPCMOV	XOP	
VPCOMB	XOP	—
VPCOMD	XOP	—
VPCOMQ	XOP	—
VPCOMUB	XOP	—
VPCOMUD	XOP	—
VPCOMUQ	XOP	—
VPCOMUW	XOP	—
VPCOMW	XOP	—
VPERMIL2PS	XOP	—
VPERMIL2PD	XOP	—
Class 10A — XOP Base (XOP.L = 1)		
VPPERM	XOP	—
VPSHAB	XOP	—
VPSHAD	XOP	—
VPSHAQ	XOP	—
VPSHAW	XOP	—
VPSHLB	XOP	—
VPSHLD	XOP	—
VPSHLQ	XOP	—
VPSHLW	XOP	—
Class 10B — XOP Base (XOP.W = 1, XOP.L = 1)		
VPMACSDD	XOP	—
VPMACSDQH	XOP	—
VPMACSDQL	XOP	—
VPMACSSDD	XOP	—
VPMACSSDQH	XOP	—
VPMACSSDQL	XOP	—
VPMACSSWD	XOP	—
VPMACSSWW	XOP	—
VPMACSWD	XOP	—
VPMACSWW	XOP	—
VPMADCSSWD	XOP	—
VPMADCSWD	XOP	—
Class 10C — XOP Base (XOP.W = 1, XOP.vvvv != 1111b, XOP.L = 1)		
VPHADDBD	XOP	—
VPHADDBQ	XOP	—
VPHADDBW	XOP	—
VPHADDD	XOP	—
VPHADDDQ	XOP	—
VPHADDUBD	XOP	—

Table 3-1. Instructions By Exception Class (continued)

Mnemonic	Extended Type	Legacy Type
VPHADDUBQ	XOP	—
VPHADDUBW	XOP	—
VPHADDUDQ	XOP	—
VPHADDUWD	XOP	—
VPHADDUWQ	XOP	—
VPHADDWD	XOP	—
VPHADDWQ	XOP	—
VPHSUBBW	XOP	—
VPHSUBDQ	XOP	—
VPHSUBWD	XOP	—
Class 10D — XOP Base (SIMD 110011, XOP.vvvv != 1111b, XOP.W = 1)		
VFRCZPD	XOP	—
VFRCZPS	XOP	—
VFRCZSD	XOP	—
VFRCZSS	XOP	—
Class 10E — XOP Base (XOP.vvvv != 1111b (variant), XOP.L = 1)		
VPROTB	XOP	—
VPROTD	XOP	—
VPROTQ	XOP	—
VPROTW	XOP	—
Class 11 — F16C Instructions		
VCVTPH2PS	F16C	—
VCVTPS2PH	F16C	—
Class 12 — AVX2 VSID (ModRM.mod = 11b, ModRM.rm != 100b)		
VGATHERDPD	AVX2	—
VGATHERDPS	AVX2	—
VGATHERQPD	AVX2	—
VGATHERQPS	AVX2	—
VPGATHERDD	AVX2	—
VPGATHERDQ	AVX2	—
VPGATHERQD	AVX2	—
VPGATHERQQ	AVX2	—
Class FMA-2 — FMA / FMA4 Vector (SIMD Exceptions PE, UE, OE, DE, IE)		
VFMADDPD	FMA4	—
VFMADDPS	FMA4	—
VFMADDSUBPD	FMA4	—
VFMADDSUBPS	FMA4	—
VFMSUBADDPD	FMA4	—
VFMSUBADDPS	FMA4	—
VFMSUBPD	FMA4	—
VFMSUBPS	FMA4	—
VFNMADDPD	FMA4	—

Table 3-1. Instructions By Exception Class (continued)

Mnemonic	Extended Type	Legacy Type
VFNMADDPS	FMA4	—
VFNMSUBPD	FMA4	—
VFNMSUBPS	FMA4	—
Class FMA-3 — FMA / FMA4 Scalar (SIMD Exceptions PE, UE, OE, DE, IE)		
VFMADDSD	FMA4	—
VFMADDSS	FMA4	—
VFMSUBSD	FMA4	—
VFMSUBSS	FMA4	—
VFNMADDSD	FMA4	—
VFNMADDSS	FMA4	—
VFNMSUBSD	FMA4	—
VFNMSUBSS	FMA4	—
Unique Cases		
XGETBV	—	—
XRSTOR	—	—
XSAVE/XSAVEOPT	—	—
XSETBV	—	—

Class 1 — AVX / SSE Vector Aligned (VEX.vvvv != 1111)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not aligned on a 16-byte boundary.
	S	S	X	Write to a read-only data segment.
			A	VEX256: Memory operand not 32-byte aligned. VEX128: Memory operand not 16-byte aligned.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 1X — SSE / AXV / AVX2 Vector (VEX.vvvv != 1111b or VEX.L=1 && !AVX2)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not aligned on a 16-byte boundary.
	S	S	X	Write to a read-only data segment.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX, AVX2, and SSE exception
A — AVX, AVX2 exception
S — SSE exception

Class 2 — AVX / SSE Vector (SIMD 111111)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Division by zero, ZE	S	S	X	Division of finite dividend by zero-value divisor.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 2-1 — AVX / SSE Vector (SIMD 111011)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 2-2 — AVX / SSE Vector (SIMD 000011)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 2-3 — AVX / SSE Vector (SIMD 100001)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 2A — AVX / SSE Vector (SIMD 111111, VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Division by zero, ZE	S	S	X	Division of finite dividend by zero-value divisor.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 2A-1 — AVX / SSE Vector (SIMD 111011, VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 2B — AVX / SSE Vector (SIMD 111111, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Division by zero, ZE	S	S	X	Division of finite dividend by zero-value divisor.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 2B-1 — AVX / SSE Vector (SIMD 100000, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

Class 2B-2 — AVX / SSE Vector (SIMD 100001, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 2B-3 — AVX / SSE Vector (SIMD 111011, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 2B-4 — AVX / SSE Vector (SIMD 100011, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Non-aligned memory operand while MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 3 — AVX / SSE Scalar (SIMD 111111)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Division by zero, ZE	S	S	X	Division of finite dividend by zero-value divisor.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 3-1 — AVX / SSE Scalar (SIMD 111011)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 3-2 — AVX / SSE Scalar (SIMD 000011)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 3-3 — AVX / SSE Scalar (SIMD 100000)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

Class 3-4 — AVX / SSE Scalar (SIMD 100001)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 3-5 — AVX / SSE Scalar (SIMD 100011)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 3A — AVX / SSE Scalar (SIMD 111111, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Division by zero, ZE	S	S	X	Division of finite dividend by zero-value divisor.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 3A-1 — AVX / SSE Scalar (SIMD 000011, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 3A-2 — AVX / SSE Scalar (SIMD 100001, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.
X — AVX and SSE exception *A — AVX exception* *S — SSE exception*				

Class 4 — AVX / SSE Vector

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Memory operand not 16-byte aligned and MXCSR.MM = 0.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 4A — AVX / SSE Vector (VEX.W = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 4B — AVX / SSE Vector (VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 4B-X — SSE / AVX / AVX2 (VEX.L = 1 && !AVX2)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

Class 4C — AVX / SSE Vector (VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 4C-1 — AVX / SSE Vector (write to RO memory, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	X	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 4D — AVX / SSE Vector (VEX.vvvv != 1111b, VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 4D-X — SSE / AVX / AVX2 Vector (VEX.vvvv != 1111b, (VEX.L = 1 && !AVX2))

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

Class 4E — AVX / SSE Vector (VEX.W = 1, VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 4E-X — SSE / AVX / AVX2 Vector (VEX.W = 1, (VEX.L = 1 && !AVX2))

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		S	X	Instruction execution caused a page fault.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

Class 4F — AVX / SSE (VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF			A	Instruction execution caused a page fault.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 4F-X — SSE / AVX / AVX2 Vector (VEX.L = 1 && !AVX2)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	When alignment checking enabled: • 128-bit memory operand not 16-byte aligned. • 256-bit memory operand not 32-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.

X — AVX, AVX2, and SSE exception
A — AVX and AVX2 exception
S — SSE exception

Class 4G — AVX Vector (VEX.W = 1, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		AVX instructions are only recognized in protected mode.
	X	X	X	CR0.EM = 1.
	X	X	X	CR4.OSFXSR = 0.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	VEX.W = 1.
			X	VEX.vvvv ! = 1111b.
			X	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	X	X	X	CR0.TS = 1.
Stack, #SS	X	X	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	X	X	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Alignment check, #AC	S	S	S	Memory operand not 16-byte aligned when alignment checking enabled and MXCSR.MM = 1.
			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		X	X	Instruction execution caused a page fault.
X — AVX exception				

Class 4H — AVX, 256-bit only (VEX.L = 0; No SIMD Exceptions)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	A	A	A	CR0.EM = 1.
	A	A	A	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L= 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	A	A	A	Lock prefix (F0h) preceding opcode.
Device not available, #NM	A	A	A	CR0.TS = 1.
Stack, #SS	A	A	A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	A	A	A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

Class 4H-1 — AVX2, 256-bit only (VEX.L = 0, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	A	A	A	CR0.EM = 1.
	A	A	A	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L= 0.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	A	A	A	Lock prefix (F0h) preceding opcode.
Device not available, #NM	A	A	A	CR0.TS = 1.
Stack, #SS	A	A	A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	A	A	A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

Class 4J — AVX2 (VEX.W = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX2 exception				

Class 4K — AVX2

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX2 exception				

Class 5 — AVX / SSE Scalar

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 5A — AVX / SSE Scalar (VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 5B — AVX / SSE Scalar (VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference with alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 5C — AVX /SSE Scalar (VEX.vvvv != 1111b, VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 5C-X — SSE / AVX / AVX2 Scalar (VEX.vvvv != 1111b, (VEX.L = 1 && !AVX2))

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

Class 5C-1 — AVX / SSE Scalar (write to RO memory, VEX.vvvv != 1111b, VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 5D — AVX / SSE Scalar (write to RO memory, VEX.vvvv != 1111b (variant))

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b (for memory destination enoding only).
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 5E — AVX / SSE Scalar (write to RO, VEX.vvvv != 1111b (variant), VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b (for memory destination encoding only).
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 6 — AVX Mixed Memory Argument

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

Class 6A — AVX Mixed Memory Argument (VEX.W = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

Class 6A-1 — AVX Mixed Memory Argument (write to RO memory, VEX.W = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
	S	S	X	Write to a read-only data segment.
Page fault, #PF			A	Instruction execution caused a page fault.
A — AVX exception.				

Class 6B — AVX Mixed Memory Argument (VEX.W = 1, VEX.L = 0)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.L = 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
A — AVX exception.				

Class 6B-1 — AVX Mixed Memory Argument (write to RO, VEX.W = 1, VEX.L = 0)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.L = 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Write to a read-only data segment.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Memory operand not 16-byte aligned when alignment checking enabled.
A — AVX exception.				

Class 6C — AVX Mixed Memory Argument (VEX.W = 1, VEX.L = 0, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 0.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

Class 6C-X — AVX / AVX2 (W=1, vvvv!=1111b, L=0, (reg src op specified && !AVX2))

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 0.
			A	Register-based source operand specified when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX, AVX2 exception.				

Class 6D — AVX Mixed Memory Argument (VEX.W = 1, VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

Class 6D-X — AVX / AVX2 (W = 1, vvvv != 1111b, (ModRM.mod = 11b && !AVX2))

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	MODRM.mod = 11b when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX, AVX2 exception.				

Class 6E — AVX Mixed Memory Argument (VEX.W = 1, VEX.vvvv != 1111b (variant))

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b (for versions with immediate byte operand only).
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

Class 6F — AVX2 (VEX.W = 1, VEX.vvvv != 1111b, VEX.L = 0, ModRM.mod = 11b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 0.
			A	Register-based source operand specified (MODRM.mod = 11b)
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Page fault, #PF			A	Instruction execution caused a page fault.
Alignment check, #AC			A	Unaligned memory reference when alignment checking enabled.
A — AVX exception.				

Class 7 — AVX / SSE No Memory Argument

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 7A — AVX /SSE No Memory Argument (VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 7A-X SSE / AVX / AVX2 Vector (VEX.L = 1 && !AVX2)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.L = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — SSE, AVX, and AVX2 exception
A — AVX, AVX2 exception
S — SSE exception

Class 7B — AVX /SSE No Memory Argument (VEX.vvvv != 1111b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 7C — AVX / SSE No Memory Argument (VEX.vvvv != 1111b, VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv field ! = 1111b.
			A	VEX.L field = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 7C-X SSE / AVX / AVX2 Vector (VEX.vvvv != 1111b, (VEX.L = 1 && !AVX2))

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv field ! = 1111b.
			A	VEX.L field = 1 when AVX2 not supported.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.

X — SSE, AVX and AVX2 exception
A — AVX, AVX2exception
S — SSE exception

Class 8 — AVX No Memory Argument (VEX.vvvv != 1111b, VEX.W = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.W = 1.
			A	VEX.vvvv ! = 1111b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
Device not available, #NM			A	CR0.TS = 1.
A — AVX exception.				

Class 9 — AVX 4-byte Argument (write to RO memory, VEX.vvvv != 1111b, VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	X	Write to a read-only data segment.
	S	S	S	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 9A — AVX 4-byte argument (reserved MBZ = 1, VEX.vvvv != 1111b, VEX.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	VEX.vvvv ! = 1111b.
			A	VEX.L = 1.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
	S	S	S	Null data segment used to reference memory.
	S	S	X	Attempt to load non-zero values into reserved MXCSR bits
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Class 10 — XOP Base

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

Class 10A — XOP Base (XOP.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

Class 10B — XOP Base (XOP.W = 1, XOP.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

Class 10C — XOP Base (XOP.W = 1, XOP.vvvv != 1111b, XOP.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			A	XOP.vvvv ! = 1111b.
			X	XOP.L = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

Class 10D — XOP Base (SIMD 110011, XOP.vvvv != 1111b, XOP.W = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.W = 1.
			X	XOP.vvvv ! = 1111b.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0. See *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			X	A source operand was an SNaN value.
			X	Undefined operation.
Denormalized operand, DE			X	A source operand was a denormal value.
Underflow, UE			X	Rounded result too small to fit into the format of the destination operand.
Precision, PE			X	A result could not be represented exactly in the destination format.
X — XOP exception				

Class 10E — XOP Base (XOP.vvvv != 1111b (variant), XOP.L = 1)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X		XOP instructions are only recognized in protected mode.
			X	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			X	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			X	XOP.vvvv ! = 1111b (for immediate operand variant only)
			X	XOP.L field = 1.
			X	REX, F2, F3, or 66 prefix preceding XOP prefix.
			X	Lock prefix (F0h) preceding opcode.
Device not available, #NM			X	CR0.TS = 1.
Stack, #SS			X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF			X	Instruction execution caused a page fault.
Alignment check, #AC			X	Memory operand not 16-byte aligned when alignment checking enabled.
X — XOP exception				

Class 11 — F16C Instructions

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		AVX instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	VEX.W field = 1.
			A	VEX.vvvv ! = 1111b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Alignment check, #AC			F	Unaligned memory reference when alignment checking enabled.
Page fault, #PF			F	Instruction execution caused a page fault.
SIMD Floating-Point Exception, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid-operation exception (IE)			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized-operand exception (DE)			F	A source operand was a denormal value.
Overflow exception (OE)			F	Rounded result too large to fit into the format of the destination operand.
Underflow exception (UE)			F	Rounded result too small to fit into the format of the destination operand.
Precision exception (PE)			F	A result could not be represented exactly in the destination format.
F — F16C exception.				

Class 12 — AVX2 VSID (ModRM.mod = 11b, ModRM.rm != 100b)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	A	A	A	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
			A	Lock prefix (F0h) preceding opcode.
			A	MODRM.mod = 11b
			A	MODRM.rm ! = 100b
			A	YMM/XMM registers specified for destination, mask, and index not unique.
Device not available, #NM			A	CR0.TS = 1.
Stack, #SS			A	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			A	Memory address exceeding data segment limit or non-canonical.
			A	Null data segment used to reference memory.
Alignment check, #AC			A	Alignment checking enabled and: 256-bit memory operand not 32-byte aligned or 128-bit memory operand not 16-byte aligned.
Page fault, #PF		A	A	Instruction execution caused a page fault.
A — AVX2 exception				

Class FMA-2 — FMA / FMA4 Vector (SIMD Exceptions PE, UE, OE, DE, IE)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Memory operand not 16-byte aligned when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

Class FMA-3 — FMA / FMA4 Scalar (SIMD Exceptions PE, UE, OE, DE, IE)

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD			F	Instruction not supported, as indicated by CPUID feature identifier.
	F	F		FMA instructions are only recognized in protected mode.
			F	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			F	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			F	REX, F2, F3, or 66 prefix preceding VEX prefix.
			F	Lock prefix (F0h) preceding opcode.
			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM			F	CR0.TS = 1.
Stack, #SS			F	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP			F	Memory address exceeding data segment limit or non-canonical.
			F	Null data segment used to reference memory.
Page fault, #PF			F	Instruction execution caused a page fault.
Alignment check, #AC			F	Non-aligned memory reference when alignment checking enabled.
SIMD floating-point, #XF			F	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE			F	A source operand was an SNaN value.
			F	Undefined operation.
Denormalized operand, DE			F	A source operand was a denormal value.
Overflow, OE			F	Rounded result too large to fit into the format of the destination operand.
Underflow, UE			F	Rounded result too small to fit into the format of the destination operand.
Precision, PE			F	A result could not be represented exactly in the destination format.
F — FMA, FMA4 exception				

XGETBV

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X	X	Lock prefix (F0h) preceding opcode.
General protection, #GP	X	X	X	ECX specifies a reserved or unimplemented XCR address.
X — exception generated				

XRSTOR

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X	X	CR4.OSFXSR = 0.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	X	X	X	CR0.TS = 1.
Stack, #SS	X	X	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	X	X	X	Memory address exceeding data segment limit or non-canonical.
	X	X	X	Null data segment used to reference memory.
	X	X	X	Memory operand not aligned on 64-byte boundary.
	X	X	X	Any must be zero (MBZ) bits in the save area were set.
	X	X	X	Attempt to set reserved bits in MXCSR.
Page fault, #PF	X	X	X	Instruction execution caused a page fault.
X — exception generated				

XSAVE/XSAVEOPT

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X	X	CR4.OSFXSR = 0.
	X	X	X	Lock prefix (F0h) preceding opcode.
Device not available, #NM	X	X	X	CR0.TS = 1.
Stack, #SS	X	X	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	X	X	X	Memory address exceeding data segment limit or non-canonical.
	X	X	X	Null data segment used to reference memory.
	X	X	X	Memory operand not aligned on 64-byte boundary.
	X	X	X	Attempt to write read-only memory.
Page fault, #PF	X	X	X	Instruction execution caused a page fault.
X — exception generated				

XSETBV

Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	X	X	X	CR4.OSFXSR = 0.
	X	X	X	Lock prefix (F0h) preceding opcode.
General protection, #GP	X	X	X	CPL != 0.
	X	X	X	ECX specifies a reserved or unimplemented XCR address.
	X	X	X	Any must be zero (MBZ) bits in the save area were set.
	X	X	X	Writing 0 to XCR0.

X — exception generated

Note:
In virtual mode, only #UD for Instruction not supported and #GP for CPL != 0 are supported.

Appendix A AES Instructions

This appendix gives background information concerning the use of the AES instruction subset in the implementation of encryption compliant to the Advanced Encryption Standard (AES).

A.1 AES Overview

This section provides an overview of AMD64 instructions that support AES software implementation.

The U.S. National Institute of Standards and Technology has adopted the Rijndael algorithm, a block cipher that processes 16-byte data blocks using a shared key of variable length, as the Advanced Encryption Standard (AES). The standard is defined in Federal Information Processing Standards Publication 197 (FIPS 197), *Specification for the Advanced Encryption Standard (AES)*. There are three versions of the algorithm, based on key widths of 16 (AES-128), 24 (AES-192), and 32 (AES-256) bytes.

The following AMD64 instructions support AES implementation:

- AESDEC/VAESDEC and AESDECLAST/VAESDECLAST
  Perform one round of AES decryption
- AESENC/VAESENC and AESENCLAST/VAESENCLAST
  Perform one round of AES encryption
- AESIMC/VAESIMC
  Perform the AES InvMixColumn transformation
  - AESKEYGENASSIST/VAESKEYGENASSIST
    Assist AES round key generation
  - PCLMULQDQ, VPCLMULQDQ
    Perform carry-less multiplication

See Chapter 2, “Instruction Reference” for detailed descriptions of the instructions.

A.2 Coding Conventions

This overview uses descriptive code that has the following basic characteristics.

- Syntax and notation based on the C language
- Four numerical data types:
  - bool: The numbers 0 and 1, the values of the Boolean constants false and true
  - nat: The infinite set of all natural numbers, including bool as a subtype
  - int: The infinite set of all integers, including nat as a subtype
  - rat: The infinite set of all rational numbers, including int as a subtype

- Standard logical and arithmetic operators
- Enumeration (enum) types, arrays, structures (struct), and union types
- Global and local variable and constant declarations, initializations, and assignments
- Standard control constructs (if, then, else, for, while, switch, break, and continue)
- Function subroutines
- Macro definitions (#define)

A.3 AES Data Structures

The AES instructions operate on 16-byte blocks of text called the *state*. Each block is represented as a 4×4 matrix of bytes which is assigned the Galois field matrix data type (GFMatrix). In the AMD64 implementation, the matrices are formatted as 16-byte vectors in XMM registers or 128-bit memory locations. This overview represents each matrix as a sequence of 16 bytes in little-endian format (least significant byte on the right and most significant byte on the left).

Figure A-1 shows a state block in 4×4 matrix representation.

$$GFMatrix = \begin{vmatrix} X_{3,0} & X_{2,0} & X_{1,0} & X_{0,0} \\ X_{3,1} & X_{2,1} & X_{1,1} & X_{0,1} \\ X_{3,2} & X_{2,2} & X_{1,2} & X_{0,2} \\ X_{3,3} & X_{2,3} & X_{1,3} & X_{0,3} \end{vmatrix}$$

Figure A-1. GFMatrix Representation of 16-byte Block

Figure A-2 shows the AMD64 AES format, with the corresponding mapping of FIPS 197 AES "words" to operand bytes.

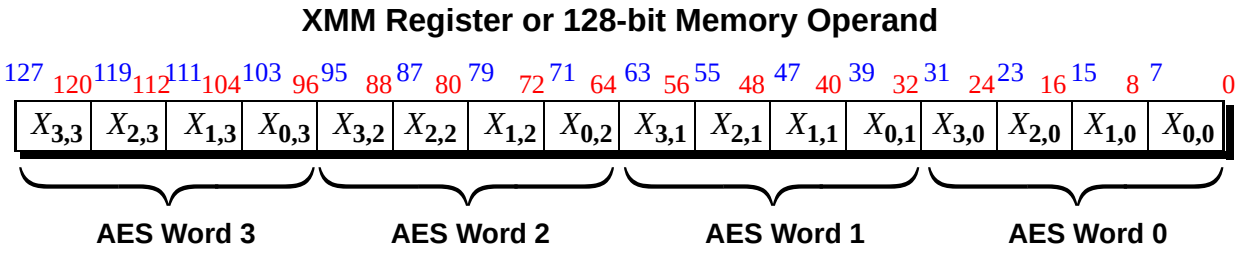


Figure A-2. GFMatrix to Operand Byte Mappings

A.4 Algebraic Preliminaries

AES operations are based on the Galois field $GF = GF(2^8)$, of order 256, constructed by adjoining a root of the irreducible polynomial

$$p(X) = X^8 + X^4 + X^3 + X + 1$$

to the field of two elements, $\mathbb{Z}_2$. Equivalently, *GF* is the quotient field $\mathbb{Z}_2[X]/p(X)$ and thus may be viewed as the set of all polynomials of degree less than 8 in $\mathbb{Z}_2[X]$ with the operations of addition and multiplication modulo $p(X)$. These operations may be implemented efficiently by exploiting the mapping from $\mathbb{Z}_2[X]$ to the natural numbers given by

$$a_nX^n + \ldots + a_1X + a_0 \rightarrow 2^n a_n + \ldots + 2a_1 + a_0 \rightarrow a_n \ldots a_1a_0\text{b}$$

For example:

$$1 \rightarrow 01\text{h}$$

$$X \rightarrow 02\text{h}$$

$$X^2 \rightarrow 04\text{h}$$

$$X^4 + X^3 + 1 \rightarrow 19\text{h}$$

$$p(X) \rightarrow 11\text{Bh}$$

Thus, each element of *GF* is identified with a unique byte. This overview uses the data type **GF256** as an alias of **nat**, to identify variables that are to be thought of as elements of *GF*.

The operations of addition and multiplication in *GF* are denoted by $\oplus$ and $\odot$, respectively. Since $\mathbb{Z}_2$ is of characteristic 2, addition is simply the "exclusive or" operation:

$$x \oplus y = x \wedge y$$

In particular, every element of *GF* is its own additive inverse.

Multiplication in *GF* may be computed as a sequence of additions and multiplications by 2. Note that this operation may be viewed as multiplication in $\mathbb{Z}_2[X]$ followed by a possible reduction modulo $p(X)$. Since 2 corresponds to the polynomial X and 11B corresponds to $p(X)$, for any $x \in GF$,

$$2 \odot x = \begin{cases} x << 1 & \text{if } x < 80\text{h} \\ (x << 1) \oplus 11\text{Bh} & \text{if } x \geq 80\text{h} \end{cases}$$

Now, if $y = b_7 \ldots b_1 b_0 b$, then

$$x \odot y = 2 \odot (\ldots (2 \odot (2 \odot (b_7 \odot x) \oplus b_6 \odot x) \oplus b_5 \odot x) \ldots b_0.$$

This computation is performed by the **GFMul( )** function.

A.4.1 Multiplication in the Field GF

The **GFMul( )** function operates on GF256 elements in SRC1 and SRC2 and returns a GF256 matrix in the destination.

```
GF256 GFMul(GF256 x, GF256 y) {
  nat sum = 0;
```

```
  for (int i=7; i>=0; i--) {
    // Multiply sum by 2. This amounts to a shift followed
    // by reduction mod 0x11B:
    sum <<= 1;
    if (sum > 0xFF) {sum = sum ^ 0x11B;}
    // Add y[i]*x:
    if (y[i]) {sum = sum ^ x;}
  }
  return sum;
}
```

Because the multiplicative group *GF** is of order 255, the inverse of an element *x* of *GF* may be computed by repeated multiplication as $x^{-1} = x^{254}$. A more efficient computation, however, is performed by the **GFInv( )** function as an application of Euclid's greatest common divisor algorithm. See Section A.11, "Computation of GFInv with Euclidean Greatest Common Divisor" for an analysis of this computation and the **GFInv( )** function.

The AES algorithms operate on the vector space GF^4, of dimension 4 over *GF*, which is represented by the array type **GFWord**. FIPS 197 refers to an object of this type as a *word*. This overview uses the term *GF word* in order to avoid confusion with the AMD64 notion of a 16-bit word.

A **GFMatrix** is an array of four *GF* words, which are viewed as the rows of a 4×4 matrix over *GF*.

The field operation symbols $\oplus$ and $\odot$ are used to denote addition and multiplication of matrices over *GF* as well. The **GFMatrixMul( )** function computes the product $A \odot B$ of 4×4 matrices.

A.4.2 Multiplication of 4x4 Matrices Over GF

```
, GFMatrix GFMatrixMul(GFMatrix a, GFMatrix b) {
  GFMatrix c;
  for (nat i=0; i<4; i++) {
    for (nat j=0; j<4; j++) {
      c[i][j] = 0;
      for (nat k=0; k<4; k++) {
        c[i][j] = c[i][j] ^ GFMul(a[i][k], b[k][j]);
      }
    }
  }
  return c;
}
```

A.5 AES Operations

The AES encryption and decryption procedures may be specified as follows, in terms of a set of basic operations that are defined later in this section. See the alphabetic instruction reference for detailed descriptions of the instructions that are used to implement the procedures.

Call the **Encrypt** or **Decrypt** procedure, which pass the same expanded key to the functions

TextBlock Cipher(TextBlock in, ExpandedKey w, nat Nk)

and

TextBlock InvCipher(TextBlock in, ExpandedKey w, nat Nk)

In both cases, the input text is converted by

GFMatrix Text2Matrix(TextBlock A)

to a matrix, which becomes the initial state of the process. This state is transformed through the sequence of N_r + 1 rounds and ultimately converted back to a linear array by

TextBlock Matrix2Text(GFMatrix M).

In each round *i*, the round key K_i is extracted from the expanded key *w* and added to the state by

GFMatrix AddRoundKey(GFMatrix state, ExpandedKey w, nat round).

Note that **AddRoundKey** does not explicitly construct K_i , but operates directly on the bytes of *w*.

The rounds of **Cipher** are numbered 0,…N_r. Let *X* be the initial state an an execution, i.e., the input in matrix format, let S_i be the state produced by round *i*, and let $Y = S_{Nr}$ be the final state. Let Σ , *R* , and *C* denote the operations performed by **SubBytes**, **ShiftRows**, **MixColumns**, respectively. Then

The initial round is a simple addition:

$$S_0 = X \oplus K_0;$$

Each of the next N_r + 1 rounds is a composition of four operations:

$$S_i = \mathcal{C}(\mathcal{R}(\Sigma(S_{i-1}))) \oplus K_i \quad \text{for} \quad i = 1, \ldots, N_r - 1;$$

The **MixColumns** transformation is omitted from the final round:

$$Y = S_{N_r} = \mathcal{R}(\Sigma(S_{N_r-1})) \oplus K_{N_r}.$$

Composing these expressions yields

$$Y = \mathcal{R}(\Sigma(\mathcal{C}(\mathcal{R}(\Sigma(\cdots(\mathcal{C}(\mathcal{R}(\Sigma(X \oplus K_0))) \oplus K_1)\cdots))) \oplus K_{N_r-1})) \oplus K_{N_r}.$$

Note that the rounds of **InvCipher** are numbered in reverse order, N_r,…,0. If *Σ*' and *Y*' are the initial and final states and S'_i is the state following round *i* , then

$$S'_{N_r} = X' \oplus K_{N_r};$$

$$S'_i = \mathcal{C}^{-1}(\Sigma^{-1}(\mathcal{R}^{-1}(S'_{i+1})) \oplus K_i) \quad \text{for} \quad i = N_r - 1, \ldots, 1;$$

$$Y' = \Sigma^{-1}(\mathcal{R}^{-1}(S'_1)) \oplus K_0.$$

Composing these expressions yields

$$Y' = \Sigma^{-1}(\mathcal{R}^{-1}(\mathcal{C}^{-1}(\Sigma^{-1}(\mathcal{R}^{-1}(\cdots(\mathcal{C}^{-1}(\Sigma^{-1}(\mathcal{R}^{-1}(X' \oplus K_{n_r}) \oplus K_{N_r-1}))\cdots)) \oplus K_1))) \oplus K_0.$$

In order to show that **InvCipher** is the inverse of **Cipher**, it is only necessary to combine these expanded expressions by replacing *X'* with *Y* and cancel inverse operations to yield *Y'* = *X*.

A.5.1 Sequence of Operations

- Use predefined **SBox** and **InvSBox** matrices or initialize the matrices using the **ComputeSBox** and **ComputeInvSBox** functions.
- Call the **Encrypt** or **Decrypt** procedure.
- For the **Encrypt** procedure:

1. Load the input **TextBlock** and **CipherKey**.
2. Expand the cipher key using the **KeyExpansion** function.
3. Call the **Cipher** function to perform the number of rounds determined by the cipher key length.
4. Perform round entry operations.
   a. Convert input text block to state matrix using the **Text2Matrix** function.
   b. Combine state and round key bytes by bitwise XOR using the **AddRoundKey** function.
5. Perform round iteration operations.
   a. Replace each state byte with another by non-linear substitution using the **SubBytes** function.
   b. Shift each row of the state cyclically using the **ShiftRows** function.
   c. Combine the four bytes in each column of the state using the **MixColumns** function.
   d. Perform **AddRoundKey**.
6. Perform round exit operations.
   a. Perform **SubBytes**.
   b. Perform **ShiftRows**.
   c. Perform **AddRoundKey**.
   d. Convert state matrix to output text block using the **Matrix2Text** function and return **TextBlock**.

- For the **Decrypt** procedure:

1. Load the input **TextBlock** and **CipherKey**.

2. Expand the cipher key using the **KeyExpansion** function.
3. Call the **InvCipher** function to perform the number of rounds determined by the cipher key length.
4. Perform round entry operations.
   a. Convert input text block to state matrix using the **Text2Matrix** function.
   b. Combine state and round key bytes by bitwise XOR using the **AddRoundKey** function.
5. Perform round iteration operations.
   a. Shift each row of the state cyclically using the **InvShiftRows** function.
   b. Replace each state byte with another by non-linear substitution using the **InvSubBytes** function.
   c. Perform **AddRoundKey**.
   d. Combine the four bytes in each column of the state using the **InvMixColumns** function.
6. Perform round exit operations.
   a. Perform **InvShiftRows**.
   b. Perform **InvSubBytes** (**InvSubWord**).
   c. Perform **AddRoundKey**.
   d. Convert state matrix to output text block using the **Matrix2Text** function and return **TextBlock**.

A.6 Initializing the Sbox and InvSBox Matrices

The AES makes use of a bijective mapping $\sigma : GF \rightarrow GF$, which is encoded, along with its inverse mapping, in the 16×16 arrays **SBox** (for encryption) and **InvSBox** (for decryption), as follows:

for all $x \in G$,

$$\sigma(x) = \text{SBox}[x[7:4], x[3:0]]$$

and

$$\sigma^{-1}(x) = \text{InvSBox}[x[7:4], x[3:0]]$$

While the FIPS 197 standard defines the contents of the **SBox[ ]** and **InvSbox [ ]** matrices, the matrices may also be initialized algebraically (and algorithmically) by means of the **ComputeSBox( )** and **ComputeInvSBox( )** functions, discussed below.

The bijective mappings for encryption and decryption are computed by the **SubByte( )** and **InvSubByte ( )** functions, respectively:

SubByte( ) computation:

```
GF256 SubByte(GF256 x) {
  return SBox[x[7:4]][x[3:0]];
}
```

InvSubByte ( ) computation:

```
GF256 InvSubByte(GF256 x) {
  return InvSBox[x[7:4]][x[3:0]];
}
```

A.6.1 Computation of SBox and InvSBox

Computation of SBox and InvSBox elements has a direct relationship to the cryptographic properties of the AES, but not to the algorithms that use the tables. Readers who prefer to view σ as a primitive operation may skip the remainder of this section.

The algorithmic definition of the bijective mapping σ is based on the consideration of *GF* as an 8-dimensional vector space over the subfield $\mathbb{Z}_2$. Let φ be a linear operator on this vector space and let $M = [a_{ij}]$ be the matrix representation of φ with respect to the ordered basis {1, 2, 4, 10, 20, 40, 80}. Then φ may be encoded concisely as an array of bytes *A* of dimension 8, each entry of which is the concatenation of the corresponding row of *M*:

$$A[i] = a_{i\mathbf{8}}\, a_{i\mathbf{7}} \ldots a_{i\mathbf{0}}$$

This expression may be represented algorithmically by means of the **ApplyLinearOp( )** function, which applies a linear operator to an element of GF. The **ApplyLinear Op( )** function is used in the initialization of both the **sBox[]** and I**nvSBox[ ]** matrices.

```
// The following function takes the array A representing a linear operator phi and
// an element x of G and returns phi(x):

GF256 ApplyLinearOp(GF256 A[8], GF256 x) {
  GF256 result = 0;
  for (nat i=0; i<8; i++) {
    bool sum = 0;
    for (nat j=0; j<8; j++) {
      sum = sum ^ (A[i][j] & x[j]);
    }
    result[i] = sum;
  }
  return result;
}
```

The definition of σ involves the linear operator φ with matrix

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

In this case,

$$A = \{F1, E3, C7, 8F, 1F, 3E, 7C, F8\}.$$

Initialization of SBox[ ]

The mapping $\sigma : G \rightarrow G$ is defined by

$$\sigma(x) = \varphi\,(x^{-1}) \oplus 63$$

This computation is performed by **ComputeSBox( )**.

ComputeSBox( )

```
GF256[16][16] ComputeSBox() {
  GF256 result[16][16];
  GF256 A[8] = {0xF1, 0xE3, 0xC7, 0x8F, 0x1F, 0x3E, 0x7C, 0xF8};
  for (nat i=0; i<16; i++) {
    for (nat j=0; j<16; j++) {
      GF256 x = (i << 4) | j;
      result[i][j] = ApplyLinearOp(A, GFInv(x)) ^ 0x63;
    }
  }
  return result;
}

const GF256 SBox[16][16] = ComputeSBox();
```

Table A-1 shows the resulting **SBox[ ]**, as defined in FIPS 197.

Table A-1. SBox Definition

	S[3:0]																
S[7:4]		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	a5
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

A.6.2 Initialization of InvSBox[ ]

A straightforward calculation confirms that the matrix M is nonsingular with inverse.

Thus, φ is invertible and φ^{-1} is encoded as the array

$$M^{-1} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

$$B = \{A4, 49, 92, 25, 4A, 94, 29, 52\}.$$

If $y = \sigma(x)$, then

$$
\begin{aligned}
(\varphi^{-1}((y) \oplus 5)^{-1} &= (\varphi^{-1}(y \oplus \varphi(5))^{-1} \\
&= (\varphi^{-1}(y \oplus 63))^{-1} \\
&= (\varphi^{-1}(\varphi(x^{-1}) \oplus 63 \oplus 63))^{-1} \\
&= (\varphi^{-1}(\varphi(x^{-1})))^{-1} \\
&= x,
\end{aligned}
$$

and σ is a permutation of *GF* with

$$\sigma^{-1}(y) = (\varphi^{-1}(y) \oplus 5)^{-1}$$

This computation is performed by **ComputeInvSBox( )**.

ComputeInvSBox( )

```
GF256[16][16] ComputeInvSBox() {
  GF256 result[16][16];
  GF256 B[8] = {0xA4, 0x49, 0x92, 0x25, 0x4A, 0x94, 0x29, 0x52};
  for (nat i=0; i<16; i++) {
    for (nat j=0; j<16; j++) {
      GF256 y = (i << 4) | j;
      result[i][j] = GFInv(ApplyLinearOp(B, y) ^ 0x5);
    }
  }
  return result;
}

const GF256 InvSBox[16][16] = ComputeInvSBox();
```

Table A-2 shows the resulting **InvSBox[ ]**, as defined in the FIPS 197.

Table A-2. InvSBox Definition

	S[3:0]																
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
S[7:4]	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

A.7 Encryption and Decryption

The AMD64 architecture implements the AES algorithm by means of an iterative function called a *round* for both encryption and the inverse operation, decryption.

The top-level encryption and decryption procedures **Encrypt( )** and **Decrypt( )** set up the rounds and invoke the functions that perform them. Each of the procedures takes two 128-bit binary arguments:

- *input data* — a 16-byte block of text stored in a source 128-bit XMM register
- *cipher key* — a 16-, 24-, or 32-byte cipher key stored in either a second 128-bit XMM register or 128-bit memory location

A.7.1 The Encrypt( ) and Decrypt( ) Procedures

```
TextBlock Encrypt(TextBlock in, CipherKey key, nat Nk) {
  return Cipher(in, ExpandKey(key, Nk), Nk);
}

TextBlock Decrypt(TextBlock in, CipherKey key, nat Nk) {
  return InvCipher(in, ExpandKey(key, Nk), Nk);
```

```
}
```

The array types **TextBlock** and **CipherKey** are introduced to accommodate the text and key parameters. The 16-, 24-, or 32-byte cipher keys correspond to AES-128, AES-192, or AES-256 key sizes. The cipher key is logically partitioned into N_k = 4, 6, or 8 AES 32-bit words. N_k is passed as a parameter to determine the AES version to be executed, and the number of rounds to be performed.

Both the **Encrypt( )** and **Decrypt( )** procedures invoke the **ExpandKey( )** function to expand the cipher key for use in round key generation. When key expansion is complete, either the **Cipher( )** or **InvCipher( )** functions are invoked.

The **Cipher( )** and **InvCipher( )** functions are the key components of the encryption and decryption process. See Section A.8, “The Cipher Function” and Section A.9, “The InvCipher Function” for detailed information.

A.7.2 Round Sequences and Key Expansion

Encryption and decryption are performed in a sequence of rounds indexed by 0, …, N_r, where N_r is determined by the number N_k of *GF* words in the cipher key. A key matrix called a *round key* is generated for each round. The number of *GF* words required to form N_r + 1 round keys is equal to , 4(N_r + 1). Table A-3 shows the relationship between cipher key length, round sequence length, and round key length.

Table A-3. Cipher Key, Round Sequence, and Round Key Length

N_k	N_r	$4(N_r + 1)$
4	10	44
6	12	52
8	14	60

Expanded keys are generated from the cipher key by the **ExpandKey( )** function, where the array type **ExpandedKey** is defined to accommodate 60 words (the maximum required) corresponding to $N_k = 8$.

The ExpandKey( ) Function

```
ExpandedKey ExpandKey(CipherKey key, nat Nk) {
  assert((Nk == 4) || (Nk == 6) || (Nk == 8));
  nat Nr = Nk + 6;
  ExpandedKey w;

  // Copy key into first Nk rows of w:
  for (nat i=0; i<Nk; i++) {
    for (nat j=0; j<4; j++) {
      w[i][j] = key[4*i+j];
    }
  }
```

```
  // Write next row of w:
  for (nat i=Nk; i<4*(Nr+1); i++) {

    // Encode preceding row:
    GFWord tmp = w[i-1];
    if (mod(i, Nk) == 0) {
      tmp = SubWord(RotWord(tmp));
      tmp[0] = tmp[0] ^ RCON[i/Nk];
    }
    else if ((Nk == 8) && (mod(i, Nk) == 4)) {
      tmp = SubWord(tmp);
    }

    // XOR tmp with w[i-Nk]:
    for (nat j=0; j<4; j++) {
      w[i][j] = w[i-Nk][j] ^ tmp[j];
    }
  }
  return w;
}
```

ExpandKey( ) begins by copying the input cipher key into the first N_k *GF* words of the expanded key *w*. The remaining $4(N_r + 1) – N_k$ GF words are computed iteratively. For each $i \geq N_k$, $w[i]$ is derived from the two *GF* words $w[i – 1]$ and $w[i – N_k]$. In most cases, $w[i]$ is simply the sum $w[i – 1] \oplus w[i – N_k]$. There are two exceptions:

- If *i* is divisible by N_k, then before adding it to $w[i – N_k]$, $w[i – 1]$ is first rotated by one position to the left by **RotWord( )**, then transformed by the substitution **SubWord( )**, and an element of the array **RCON** is added to it.

  RCON[11] = {00h, 01h, 02h, 04h, 08h, 10h, 20h, 40h, 80h, 1Bh, 36h}

- In the case $N_k = 8$, if *i* is divisible by 4 but not 8, then $w[i – 1]$ is transformed by the substitution **SubWord( )**.

The i^{th} round key K_i comprises the four *GF* words $w[4i], \ldots, w[4i + 3]$. More precisely, let W_i be the matrix

$$W = \{w[4i], w[4i + 1], w[4i + 2], w[4i + 3]\}$$

Then $K_i = W_i^t$, the transpose of W_i. Thus, the entries of the array *w* are the columns of the round keys.

A.8 The Cipher Function

This function performs encryption. It converts the input text to matrix form, generates the round key from the expanded key matrix, and iterates through the transforming functions the number of times determined by encryption key size to produce a 128-bit binary cipher matrix. As a final step, it converts the matrix to an output text block.

```
TextBlock Cipher(TextBlock in, ExpandedKey w, nat Nk) {
  assert((Nk == 4) || (Nk == 6) || (Nk == 8));
  nat Nr = Nk + 6;
  GFMatrix state = Text2Matrix(in);
  state = AddRoundKey(state, w, 0);
  for (nat round=1; round<Nr; round++) {
    state = SubBytes(state);
    state = ShiftRows(state);
    state = MixColumns(state);
    state = AddRoundKey(state, w, round);
  }
  state = SubBytes(state);
  state = ShiftRows(state);
  state = AddRoundKey(state, w, Nr);
  return Matrix2Text(state);
}
```

A.8.1 Text to Matrix Conversion

Prior to processing, the input text block must be converted to matrix form. The **Text2Matrix( )** function stores a **TextBlock** in a GFMatrix in column-major order as follows.

```
GFMatrix Text2Matrix(TextBlock A) {
  GFMatrix result;
  for (nat j=0; j<4; j++) {
    for (nat i=0; i<4; i++) {
      result[i][j] = A[4*j+i];
    }
  }
  return result;
}
```

A.8.2 Cipher Transformations

The Cipher function employs the following transformations.

SubBytes( ) — Applies a non-linear substitution table (SBox) to each byte of the state.

SubWord( ) — Uses a non-linear substitution table (SBox) to produce a four-byte AES output word from the four bytes of an AES input word.

ShiftRows( ) — Cyclically shifts the last three rows of the state by various offsets.

RotWord( ) — Rotates an AES (4-byte) word to the right.

MixColumns( ) — Mixes data in all the state columns independently to produce new columns.

AddRoundKey( ) — Extracts a 128-bit round key from the expanded key matrix and adds it to the 128-bit state using an XOR operation.

Inverses of **SubBytes( )**, **SubWord( )**, **ShiftRows( )** and **MixColumns( )** are used in decryption. See Section A.9, “The InvCipher Function” for more information.

SubBytes( ) Function

Performs a byte substitution operation using the invertible substitution table (**SBox**) to convert input text to an intermediate encryption state.

```
GFMatrix SubBytes(GFMatrix M) {
  GFMatrix result;
  for (nat i=0; i<4; i++) {
    result[i] = SubWord(M[i]);
  }
  return result;
}
```

SubWord( ) Function

Applies **SubBytes** to each element of a vector or a matrix:

```
GFWord SubWord(GFWord x) {
  GFWord result;
  for (nat i=0; i<4; i++) {
    result[i] = SubByte(x[i]);
  }
  return result;
}
```

ShiftRows( ) Function

Cyclically shifts the last three rows of the state by various offsets.

```
GFMatrix ShiftRows(GFMatrix M) {
  GFMatrix result;
  for (nat i=0; i<4; i++) {
    result[i] = RotateLeft(M[i], -i);
  }
  return result;
```

RotWord( ) Function

Performs byte-wise cyclic permutation of a 32-bit AES word.

```
GFWord RotWord(GFWord x)
{ return RotateLeft(x, 1); }
```

MixColumns( ) Function

Performs a byte-oriented column-by-column matrix multiplication

$$M \rightarrow \mathrm{C} \odot M \text{, where C is the predefined fixed matrix}$$

$$C = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

The function is implemented as follows:

```
GFMatrix MixColumns(GFMatrix M) {
  GFMatrix C = {
    {0x02,0x03,0x01,0x01},
    {0x01,0x02,0x03,0x01},
    {0x01,0x01,0x02,0x03},
    {0x03,0x01,0x01,0x02}
  };
  return GFMatrixMul(C, M);
}
```

AddRoundKey( ) Function

Extracts the round key from the expanded key and adds it to the state using a bitwise XOR operation.

```
GFMatrix AddRoundKey(GFMatrix state, ExpandedKey w, nat round) {
  GFMatrix result = state;
  for (nat i=0; i<4; i++) {
    for (nat j=0; j<4; j++) {
      result[i][j] = result[i][j] ^ w[4*round+j][i];
    }
  }
  return result;
}
```

A.8.3 Matrix to Text Conversion

After processing, the output matrix must be converted to a text block. The **Matrix2Text( )** function converts a GFMatrix in column-major order to a **TextBlock** as follows.

```
TextBlock Matrix2Text(GFMatrix M) {
  TextBlock result;
  for (nat j=0; j<4; j++) {
    for (nat i=0; i<4; i++) {
      result[4*j+i] = M[i][j];
    }
  }
  return result;
}
```

A.9 The InvCipher Function

This function performs decryption. It iterates through the round function the number of times determined by encryption key size and produces a 128-bit block of text as output.

```
TextBlock InvCipher(TextBlock in, ExpandedKey w, nat Nk) {
  assert((Nk == 4) || (Nk == 6) || (Nk == 8));
  nat Nr = Nk + 6;
  GFMatrix state = Text2Matrix(in);
  state = AddRoundKey(state, w, Nr);
  for (nat round=Nr-1; round>0; round--) {
    state = InvShiftRows(state);
    state = InvSubBytes(state);
```

```
    state = AddRoundKey(state, w, round);
    state = InvMixColumns(state);
  }
  state = InvShiftRows(state);
  state = InvSubBytes(state);
  state = AddRoundKey(state, w, 0);
  return Matrix2Text(state);
}
```

A.9.1 Text to Matrix Conversion

Prior to processing, the input text block must be converted to matrix form. The **Text2Matrix( )** function stores a **TextBlock** in a GFMatrix in column-major order as follows.

```
GFMatrix Text2Matrix(TextBlock A) {
  GFMatrix result;
  for (nat j=0; j<4; j++) {
    for (nat i=0; i<4; i++) {
      result[i][j] = A[4*j+i];
    }
  }
  return result;
}
```

A.9.2 InvCypher Transformations

The following functions are used in decryption:

InvShiftRows( ) — The inverse of **ShiftRows( )**.

InvSubBytes( ) — The inverse of **SubBytes( )**.

InvSubWord( ) — The inverse of **SubWord( )**.

InvMixColumns( ) — The inverse of **MixColumns( )**.

AddRoundKey( ) — Is its own inverse.

Decryption is the inverse of encryption and is accomplished by means of the inverses of the, **SubBytes( )**, **SubWord( )**, **ShiftRows( )** and **MixColumns( )** transformations used in encryption.

SubWord( ), **SubBytes( )**, and **ShiftRows( )** are injective. This is also the case with **MixColumns( )**. A simple computation shows that *C* is invertible with

$$C^{-1} = \begin{bmatrix} E & B & D & 9 \\ 9 & E & B & D \\ D & 9 & E & B \\ B & D & 9 & E \end{bmatrix}$$

InvShiftRows( ) Function

The inverse of **ShiftRows( )**.

```
GFMatrix InvShiftRows(GFMatrix M) {
  GFMatrix result;
```

```
  for (nat i=0; i<4; i++) {
    result[i] = RotateLeft(M[i], -i);
  }
  return result;
```

InvSubBytes( ) Function

The inverse of *SubBytes*( ).

```
GFMatrix InvSubBytes(GFMatrix M) {
  GFMatrix result;
  for (nat i=0; i<4; i++) {
    result[i] = InvSubWord(M[i]);
  }
  return result;
}
```

InvSubWord( ) Function

The inverse of **SubWord( )**, **InvSubBytes( )** applied to each element of a vector or a matrix.

```
GFWord InvSubWord(GFWord x) {
  GFWord result;
  for (nat i=0; i<4; i++) {
    result[i] = InvSubByte(x[i]);
  }
  return result;
}
```

InvMixColumns( ) Function

The inverse of the **MixColumns( )** function. Multiplies by the inverse of the predefined fixed matrix, C, C^{-1}, as discussed previously.

```
GFMatrix InvMixColumns(GFMatrix M) {
  GFMatrix D = {
    {0x0e,0x0b,0x0d,0x09},
    {0x09,0x0e,0x0b,0x0d},
    {0x0d,0x09,0x0e,0x0b},
    {0x0b,0x0d,0x09,0x0e}
  };
  return GFMatrixMul(D, M);
}
```

AddRoundKey( ) Function

Extracts the round key from the expanded key and adds it to the state using a bitwise XOR operation.

```
GFMatrix AddRoundKey(GFMatrix state, ExpandedKey w, nat round) {
  GFMatrix result = state;
  for (nat i=0; i<4; i++) {
    for (nat j=0; j<4; j++) {
      result[i][j] = result[i][j] ^ w[4*round+j][i];
    }
  }
  return result;
```

```
}
```

A.9.3 Matrix to Text Conversion

After processing, the output matrix must be converted to a text block. The **Matrix2Text( )** function converts a GFMatrix in column-major order to a **TextBlock** as follows.

```
TextBlock Matrix2Text(GFMatrix M) {
  TextBlock result;
  for (nat j=0; j<4; j++) {
    for (nat i=0; i<4; i++) {
      result[4*j+i] = M[i][j];
    }
  }
  return result;
}
```

A.10 An Alternative Decryption Procedure

This section outlines an alternative decrypting procedure,

TextBlock EqDecrypt(TextBlock in, CipherKey key, nat Nk):

```
TextBlock EqDecrypt(TextBlock in, CipherKey key, nat Nk) {
  return EqInvCipher(in, MixRoundKeys(ExpandKey(key, Nk), Nk), Nk);
}
```

The procedure is based on a variation of **InvCipher**,

TextBlock EqInvCipher(TextBlock in, ExpandedKey w, nat Nk):

```
TextBlock EqInvCipher(TextBlock in, ExpandedKey dw, nat Nk) {
  assert((Nk == 4) || (Nk == 6) || (Nk == 8));
  nat Nr = Nk + 6;
  GFMatrix state = Text2Matrix(in);
  state = AddRoundKey(state, dw, Nr);
  for (nat round=Nr-1; round>0; round--) {
    state = InvSubBytes(state);
    state = InvShiftRows(state);
    state = InvMixColumns(state);
    state = AddRoundKey(state, dw, round);
  }
  state = InvSubBytes(state);
  state = InvShiftRows(state);
  state = AddRoundKey(state, dw, 0);
  return Matrix2Text(state);
}
```

The variant structure more closely resembles that of **Cipher**. This requires a modification of the expanded key generated by **ExpandKey**,

ExpandedKey MixRoundKeys(ExpandedKey w, nat Nk):

```
ExpandedKey MixRoundKeys(ExpandedKey w, nat Nk) {
  assert((Nk == 4) || (Nk == 6) || (Nk == 8));
  nat Nr = Nk + 6;
  ExpandedKey result;
  GFMatrix roundKey;
  for (nat round=0; round<Nr+1; round++) {
    for (nat i=0; i<4; i++) {
      roundKey[i] = w[4*round+i];
    }
    if ((round > 0) && (round < Nr)) {
      roundKey = InvMixRows(roundKey);
    }
    for (nat i=0; i<4; i++) {
      result[4*round+i] = roundKey[i];
    }
  }
  return result;
}
```

The transformation **MixRoundKeys** leaves *K0* and K_{Nr} unchanged, but for $i = 1,\ldots,N_r - 1$, it replaces W_i with the matrix product $W_i \odot Q$, where

$$Q = \begin{bmatrix} \mathrm{E} & 9 & \mathrm{D} & \mathrm{B} \\ \mathrm{B} & \mathrm{E} & 9 & \mathrm{D} \\ \mathrm{D} & \mathrm{B} & \mathrm{E} & 9 \\ 9 & \mathrm{D} & \mathrm{B} & \mathrm{E} \end{bmatrix} = \begin{bmatrix} \mathrm{E} & \mathrm{B} & \mathrm{D} & 9 \\ 9 & \mathrm{E} & \mathrm{B} & \mathrm{D} \\ \mathrm{D} & 9 & \mathrm{E} & \mathrm{B} \\ \mathrm{B} & \mathrm{D} & 9 & \mathrm{E} \end{bmatrix}^t = (C^{-1})^t.$$

The effect of this is to replace K_i with

$$(W_i \odot Q)^t = Q^t \odot W_i^t = C^{-1} \odot K_i = \mathcal{C}^{-1}(K_i)$$

for $i = 1,\ldots,N_r - 1$.

The equivalence of **EqDecrypt** and **Decrypt** follows from two properties of the basic operations:

C is a linear transformation and therefore, so is C^{-1};

Σ and R commute, and hence so do Σ^{-1} and R^{-1}, for if

$$S = \begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix},$$

then

$$\Sigma(\mathcal{R}(S)) = \begin{bmatrix} \sigma(s_{00}) & \sigma(s_{01}) & \sigma(s_{02}) & \sigma(s_{03}) \\ \sigma(s_{11}) & \sigma(s_{12}) & \sigma(s_{13}) & \sigma(s_{10}) \\ \sigma(s_{22}) & \sigma(s_{23}) & \sigma(s_{20}) & \sigma(s_{21}) \\ \sigma(s_{33}) & \sigma(s_{30}) & \sigma(s_{31}) & \sigma(s_{32}) \end{bmatrix} = \mathcal{R}(\Sigma(S)).$$

Now let *X"* and *Y"* be the initial and final states of an execution of **EqDecrypt** and let $S"_i$ be the state following round *i* . Suppose *X"* = *X'*. Appealing to the definitions of **EqDecrypt** and **EqInvCipher**, we have

$$S''_{N_r} = X'' \oplus K_{N_r} = X' \oplus K_{N_r} = S'_{N_r},$$

and for $i = N_r - 1,\ldots,1$, by induction,

$$\begin{aligned} S''_i &= \mathcal{C}^{-1}(\Sigma^{-1}(\mathcal{R}^{-1}(S''_{i+1}))) \oplus \mathcal{C}^{-1}(K_{N_r}) \\ &= \mathcal{C}^{-1}(\Sigma^{-1}(\mathcal{R}^{-1}(S''_{i+1})) \oplus K_{N_r}) \\ &= \mathcal{C}^{-1}(\mathcal{R}^{-1}(\Sigma^{-1}(S''_{i+1})) \oplus K_{N_r}) \\ &= \mathcal{C}^{-1}(\mathcal{R}^{-1}(\Sigma^{-1}(S'_{i+1})) \oplus K_{N_r}) \\ &= S'_i. \end{aligned}$$

Finally,

$$\begin{aligned} Y'' = S''_0 &= \mathcal{R}^{-1}(\Sigma^{-1}(S''_1)) \oplus K_0 \\ &= \Sigma^{-1}(\mathcal{R}^{-1}(S''_1)) \oplus K_0 \\ &= \Sigma^{-1}(\mathcal{R}^{-1}(S'_1)) \oplus K_0 \\ &= S'_0 = Y'. \end{aligned}$$

A.11 Computation of GFInv with Euclidean Greatest Common Divisor

Note that the operations performed by **GFInv( )** are in the ring $\mathbb{Z}_2[X]$ rather than the quotient field *GF*.

The initial values of the variables x_1 and x_2 are the inputs *x* and 11b, the latter representing the polynomial *p*(*X*). The variables a_1 and a_2 are initialized to 1 and 0.

On each iteration of the loop, a multiple of the lesser of x1 and x2 is added to the other. If x1 ≤ x2, then the values of x2 and a2 are adjusted as follows:

$x_2 \rightarrow x_2 \oplus 2^s \odot x_1$

$a_2 \rightarrow a_2 \oplus 2^s \odot a_1$

where *s* is the difference in the exponents (*i.e.*, degrees) of x_1 and x_2 . In the remaining case, x_1 and a_1 are similarly adjusted. This step is repeated until either $x_1 = 0$ or $x_2 = 0$.

We make the following observations:

- On each iteration, the value added to *xi* has the same exponent as x_i, and hence the sum has lesser exponent. Therefore, termination is guaranteed.
- Since $p(X)$ is irreducible and x is of smaller degree than $p(X)$, the initial values of x_1 and x_2 have no non-trivial common factor. This property is clearly preserved by each step.
- Initially,

$$x_1 \oplus a_1 \odot x = x \oplus x = 0$$

and

$$x_2 \oplus a_2 \odot x = 11b \oplus 0 = 11b$$

are both divisible by 11b. This property is also invariant, since, for example, the above assignments result in

$$x_2 \oplus a_2 \odot x \rightarrow (x_2 \oplus 2^s \odot x_1) \oplus (a_2 \oplus 2^s \odot a_1) \odot x = (x_2 \oplus a2 \odot x) \oplus 2^s \odot (x_1 \oplus a_1 \odot x).$$

Now suppose that the loop terminates with $x_2 = 0$. Then x_1 has no non-trivial factor and, hence, $x_1 = 1$. Thus, $1 \oplus a_1 \odot x$ is divisible by 11b. Since the final result y is derived by reducing a_1 modulo 11b, it follows that $1 \oplus y \odot x$ is also divisible by 11b and, hence, in the quotient field *GF*, $1 + y \odot x = 0$, which implies $y \odot x = 1$.

The computation of the multiplicative inverse utilizing Euclid's algorithm is as follows:

```
// Computation of multiplicative inverse based on Euclid's algorithm:

GF256 GFInv(GF256 x) {
  if (x == 0) {
    return 0;
  }
  // Initialization:
  nat x1 = x;
  nat x2 = 0x11B; // the irreducible polynomial p(X)
  nat a1 = 1;
  nat a2 = 0;
  nat shift; // difference in exponents
  while ((x1 != 0) && (x2!= 0)) {

  // Termination is guaranteed, since either x1 or x2 decreases on each iteration.
  // We have the following loop invariants, viewing natural numbers as elements of
  // the polynomial ring Z2[X]:
  // (1) x1 and x2 have no common divisor other than 1.
  // (2) x1 ^ GFMul(a1, x) and x2 ^ GFMul(a2, x) are both divisible by p(X).

    if (x1 <= x2) {
      shift = expo(x2) - expo(x1);
      x2 = x2 ^ (x1 << shift);
      a2 = a2 ^ (a1 << shift);
    }
    else {
      shift = expo(x1) - expo(x2);
      x1 = x1 ^ (x2 << shift);
      a1 = a1 ^ (a2 << shift);
    }
  }
  nat y;

  // Since either x1 or x2 is 0, it follows from (1) above that the other is 1.

  if (x1 == 1) { // x2 == 0
    y = a1;
  }
  else if (x2 == 1) { // x1 == 0
    y = a2;
  }
  else {
    assert(false);
  }

  // Now it follows from (2) that GFMul(y, x) ^ 1 is divisible by 0x11b.
  // We need only reduce y modulo 0x11b:

  nat e = expo(y);
  while (e >= 8) {
    y = y ^ (0x11B << (e - 8));
    e = expo(y);
  }
  return y;
}
```

Index

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

AMD

AMD64 Technology

AMD64 Architecture Programmer's Manual

Volume 5: 64-Bit Media and x87 Floating-Point Instructions

Publication No.	Revision	Date
26569	3.16	November 2021

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

Trademarks

AMD, the AMD Arrow logo, AMD Athlon, and AMD Opteron, and combinations thereof, and 3DNow! are trademarks, and AMD-K6 is a registered trademark of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Reverse engineering or disassembly is prohibited.

MMX is a trademark and Pentium is a registered trademark of Intel Corporation.

Dolby Laboratories, Inc.

Manufactured under license from Dolby Laboratories.

Rovi Corporation

This device is protected by U.S. patents and other intellectual property rights. The use of Rovi Corporation's copy protection technology in the device must be authorized by Rovi Corporation and is intended for home and other limited pay-per-view uses only, unless otherwise authorized in writing by Rovi Corporation.

USE OF THIS PRODUCT IN ANY MANNER THAT COMPLIES WITH THE MPEG ACTUAL OR DE FACTO VIDEO AND/OR AUDIO STANDARDS IS EXPRESSLY PROHIBITED WITHOUT ALL NECESSARY LICENSES UNDER APPLICABLE PATENTS. SUCH LICENSES MAY BE ACQUIRED FROM VARIOUS THIRD PARTIES INCLUDING, BUT NOT LIMITED TO, IN THE MPEG PATENT PORTFOLIO, WHICH LICENSE IS AVAILABLE FROM MPEG LA,

Contents

Figures

Tables

Revision History

Date	Revision	Description
November 2021	3.16	Added Alignment Check, #AC to FXRSTOR and FXSAVE and additional details of alignment check behavior.
May 2018	3.15	Added clarification to CVTPI2PS. Corrected PSHUFW detail.
September 2016	3.14	Modified Exceptions for Floating-Point Load under x87 Floating-Point Exception Generated, #MF.
May 2013	3.13	Corrected CPUID function and feature bit called out in text for FXSAVE/FXRSTOR optimization. Made other corrections and clarifications related to the specification of feature bits.
March 2012	3.12	Clarified exception and trap behavior for MASKMOVQ
December 2009	3.11	Revised FCOM, FCOMP, FCOMPP and FCOMI, FCOMIP instruction pages. Corrected exception tables for FPREM and FPREM1.
April 2009	3.10	Revised FCOM, FCOMP, FCOMPP description. Corrected FISTTP exception table.
September 2007	3.09	Added minor clarifications and corrected typographical and formatting errors.
July 2007	3.08	Added misaligned access support to applicable instructions. Deprecated 3DNow!™ instructions. Added Appendix A, ”Recommended Substitutions for 3DNow!™ Instructions,” on page 333. Added minor clarifications and corrected typographical and formatting errors.
September 2006	3.07	Added minor clarifications and corrected typographical and formatting errors.
December 2005	3.06	Added minor clarifications and corrected typographical and formatting errors.

Date	Revision	Description
December 2004	3.05	Added FISTTP instruction (SSE3). Updated CPUID information in exception tables. Corrected several typographical and formatting errors.
September 2003	3.04	Clarified x87 condition codes for FPREM and FPREM1 instructions. Corrected tables of numeric ranges for results of PF2ID and PF2IW instructions.
April 2003	3.03	Corrected numerous typos and stylistic errors. Corrected description of FYL2XP1 instruction. Clarified the description of the FXRSTOR instruction.

Preface

About This Book

This book is part of a multivolume work entitled the *AMD64 Architecture Programmer's Manual*. This table lists each volume and its order number.

Title	Order No.
Volume 1: Application Programming	24592
Volume 2: System Programming	24593
Volume 3: General-Purpose and System Instructions	24594
Volume 4: 128-Bit and 256-Bit Media Instructions	26568
Volume 5: 64-Bit Media and x87 Floating-Point Instructions	26569

Audience

This volume (Volume 5) is intended for all programmers writing application or system software for a processor that implements the AMD64 processor architecture.

Organization

Volumes 3, 4, and 5 describe the AMD64 instruction set in detail. Together, they cover each instruction's mnemonic syntax, opcodes, functions, affected flags, and possible exceptions.

The AMD64 instruction set is divided into five subsets:

- General-purpose instructions
- System instructions
- 128-bit and 256-bit media instructions (Streaming SIMD Extensions – SSE)
- 64-bit media instructions (MMX™)
- x87 floating-point instructions

A number of instructions belong to—and are described identically in—multiple instruction subsets.

This volume describes the 64-bit media and x87 floating-point instructions. The index at the end cross-references topics within this volume. For other topics relating to the AMD64 architecture, and for information on instructions in other subsets, see the tables of contents and indexes of the other volumes.

Conventions and Definitions

The following section **Notational Conventions** describes notational conventions used in this volume and in the remaining volumes of this *AMD64 Architecture Programmer's Manual*. This is followed by a **Definitions** section which lists a number of terms used in the manual along with their technical definitions. Finally, the **Registers** section lists the registers which are a part of the application programming model.

Notational Conventions

#GP(0)

An instruction exception—in this example, a general-protection exception with error code of 0.

1011b

A binary value—in this example, a 4-bit value.

F0EA_0B02h

A hexadecimal value. Underscore characters may be inserted to improve readability.

128

Numbers without an alpha suffix are decimal unless the context indicates otherwise.

7:4

A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first. Commas may be inserted to indicate gaps.

CPUID Fn*XXXX_XXXX_RRR*[*FieldName*]

Support for optional features or the value of an implementation-specific parameter of a processor can be discovered by executing the CPUID instruction on that processor. To obtain this value, software must execute the CPUID instruction with the function code *XXXX_XXXX*h in EAX and then examine the field *FieldName* returned in register *RRR*. If the "*_RRR*" notation is followed by "_x*YYY*", register ECX must be set to the value *YYY*h before executing CPUID. When *FieldName* is not given, the entire contents of register *RRR* contains the desired value. When determining optional feature support, if the bit identified by *FieldName* is set to a one, the feature is supported on that processor.

CR0–CR4

A register range, from register CR0 through CR4, inclusive, with the low-order register first.

CR0[PE], CR0.PE

Notation for referring to a field within a register—in this case, the PE field of the CR0 register.

CR0[PE] = 1, CR0.PE = 1

The PE field of the CR0 register is set (contains the value 1).

EFER[LME] = 0, EFER.LME = 0

The LME field of the EFER register is cleared (contains a value of 0).

DS:SI

A far pointer or logical address. The real address or segment descriptor specified by the segment register (DS in this example) is combined with the offset contained in the second register (SI in this example) to form a real or virtual address.

RFLAGS[13:12]

A field within a register identified by its bit range. In this example, corresponding to the IOPL field.

Definitions

Many of the following definitions assume an in-depth knowledge of the legacy x86 architecture. See “Related Documents” on page xxviii for descriptions of the legacy x86 architecture.

128-bit media instructions

Instructions that operate on the various 128-bit vector data types. Supported within both the *legacy SSE* and *extended SSE* instruction sets.

256-bit media instructions

Instructions that operate on the various 256-bit vector data types. Supported within the *extended SSE* instruction set.

64-bit media instructions

Instructions that operate on the 64-bit vector data types. These are primarily a combination of MMX and 3DNow!™ instruction sets and their extensions, with some additional instructions from the *SSE1* and *SSE2* instruction sets.

16-bit mode

Legacy mode or compatibility mode in which a 16-bit address size is active. See *legacy mode* and *compatibility mode*.

32-bit mode

Legacy mode or compatibility mode in which a 32-bit address size is active. See *legacy mode* and *compatibility mode*.

64-bit mode

A submode of *long mode*. In 64-bit mode, the default address size is 64 bits and new features, such as register extensions, are supported for system and application software.

absolute

Said of a displacement that references the base of a code segment rather than an instruction pointer. Contrast with *relative*.

AES

Advance Encryption Standard (AES) algorithm acceleration instructions; part of *Streaming SIMD Extensions (SSE)*.

ASID

Address space identifier.

AVX

Extension of the SSE instruction set supporting 256-bit vector (packed) operands. See *Streaming SIMD Extensions*.

biased exponent

The sum of a floating-point value's exponent and a constant bias for a particular floating-point data type. The bias makes the range of the biased exponent always positive, which allows reciprocation without overflow.

byte

Eight bits.

clear

To write a bit value of 0. Compare *set*.

compatibility mode

A submode of *long mode*. In compatibility mode, the default address size is 32 bits, and legacy 16-bit and 32-bit applications run without modification.

commit

To irreversibly write, in program order, an instruction's result to software-visible storage, such as a register (including flags), the data cache, an internal write buffer, or memory.

CPL

Current privilege level.

direct

Referencing a memory location whose address is included in the instruction's syntax as an immediate operand. The address may be an absolute or relative address. Compare *indirect*.

dirty data

Data held in the processor's caches or internal buffers that is more recent than the copy held in main memory.

displacement

A signed value that is added to the base of a segment (absolute addressing) or an instruction pointer (relative addressing). Same as *offset*.

doubleword

Two words, or four bytes, or 32 bits.

double quadword

Eight words, or 16 bytes, or 128 bits. Also called *octword*.

effective address size

The address size for the current instruction after accounting for the default address size and any address-size override prefix.

effective operand size

The operand size for the current instruction after accounting for the default operand size and any operand-size override prefix.

element

See *vector*.

exception

An abnormal condition that occurs as the result of executing an instruction. The processor's response to an exception depends on the type of the exception. For all exceptions except SSE floating-point exceptions and x87 floating-point exceptions, control is transferred to the handler (or service routine) for that exception, as defined by the exception's vector. For floating-point exceptions defined by the IEEE 754 standard, there are both masked and unmasked responses. When unmasked, the exception handler is called, and when masked, a default response is provided instead of calling the handler.

extended SSE

Enhanced set of SIMD instructions supporting 256-bit vector data types and allowing the specification of up to four operands. A subset of the *Streaming SIMD Extensions (SSE)*. Includes the *AVX*, *FMA*, *FMA4*, and *XOP* instructions. Compare *legacy SSE*.

flush

An often ambiguous term meaning (1) writeback, if modified, and invalidate, as in "flush the cache line," or (2) invalidate, as in "flush the pipeline," or (3) change a value, as in "flush to zero."

FMA4

Fused Multiply Add, four operand. Part of the *extended SSE* instruction set.

FMA

Fused Multiply Add. Part of the *extended SSE* instruction set.

GDT

Global descriptor table.

GIF

Global interrupt flag.

IDT

Interrupt descriptor table.

IGN

Ignored. Value written is ignored by hardware. Value returned on a read is indeterminate. See *reserved*.

indirect

Referencing a memory location whose address is in a register or other memory location. The address may be an absolute or relative address. Compare *direct*.

IRB

The virtual-8086 mode interrupt-redirection bitmap.

IST

The long-mode interrupt-stack table.

IVT

The real-address mode interrupt-vector table.

LDT

Local descriptor table.

legacy x86

The legacy x86 architecture. See “Related Documents” on page xxviii for descriptions of the legacy x86 architecture.

legacy mode

An operating mode of the AMD64 architecture in which existing 16-bit and 32-bit applications and operating systems run without modification. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Legacy mode has three submodes, *real mode*, *protected mode*, and *virtual-8086 mode*.

legacy SSE

A subset of the *Streaming SIMD Extensions (SSE)* composed of the *SSE1*, *SSE2*, *SSE3*, *SSSE3*, *SSE4.1*, *SSE4.2*, and *SSE4A* instruction sets. Compare *extended SSE*.

long mode

An operating mode unique to the AMD64 architecture. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Long mode has two submodes, *64-bit mode* and *compatibility mode*.

lsb

Least-significant bit.

LSB

Least-significant byte.

main memory

Physical memory, such as RAM and ROM (but not cache memory) that is installed in a particular computer system.

mask

(1) A control bit that prevents the occurrence of a floating-point exception from invoking an exception-handling routine. (2) A field of bits used for a control purpose.

MBZ

Must be zero. If software attempts to set an MBZ bit to 1, a general-protection exception (#GP) occurs. See *reserved*.

memory

Unless otherwise specified, *main memory*.

msb

Most-significant bit.

MSB

Most-significant byte.

multimedia instructions

Those instructions that operate simultaneously on multiple elements within a vector data type. Comprises the *256-bit media instructions*, *128-bit media instructions*, and *64-bit media instructions*.

octword

Same as *double quadword.*

offset

Same as *displacement*.

overflow

The condition in which a floating-point number is larger in magnitude than the largest, finite, positive or negative number that can be represented in the data-type format being used.

packed

See *vector*.

PAE

Physical-address extensions.

physical memory

Actual memory, consisting of *main memory* and cache.

probe

A check for an address in a processor's caches or internal buffers. *External probes* originate outside the processor, and *internal probes* originate within the processor.

protected mode

A submode of *legacy mode*.

quadword

Four words, or eight bytes, or 64 bits.

RAZ

Read as zero. Value returned on a read is always zero (0) regardless of what was previously written. (See *reserved*)

real-address mode

See *real mode*.

real mode

A short name for *real-address mode*, a submode of *legacy mode*.

relative

Referencing with a displacement (also called offset) from an instruction pointer rather than the base of a code segment. Contrast with *absolute*.

reserved

Fields marked as reserved may be used at some future time.

To preserve compatibility with future processors, reserved fields require special handling when read or written by software. Software must not depend on the state of a reserved field (unless qualified as RAZ), nor upon the ability of such fields to return a previously written state.

If a field is marked reserved without qualification, software must not change the state of that field; it must reload that field with the same value returned from a prior read.

Reserved fields may be qualified as IGN, MBZ, RAZ, or SBZ (see definitions).

REX

An instruction encoding prefix that specifies a 64-bit operand size and provides access to additional registers.

RIP-relative addressing

Addressing relative to the 64-bit RIP instruction pointer.

SBZ

Should be zero. An attempt by software to set an SBZ bit to 1 results in undefined behavior. See *reserved*.

scalar

An atomic value existing independently of any specification of location, direction, etc., as opposed to *vectors*.

set

To write a bit value of 1. Compare *clear*.

SIB

A byte following an instruction opcode that specifies address calculation based on scale (S), index (I), and base (B).

SIMD

Single instruction, multiple data. See *vector*.

Streaming SIMD Extensions (SSE)

Instructions that operate on scalar or vector (packed) integer and floating point numbers. The SSE instruction set comprises the *legacy SSE* and *extended SSE* instruction sets.

SSE1

Original SSE instruction set. Includes instructions that operate on vector operands in both the MMX and the XMM registers.

SSE2

Extensions to the SSE instruction set.

SSE3

Further extensions to the SSE instruction set.

SSSE3

Further extensions to the SSE instruction set.

SSE4.1

Further extensions to the SSE instruction set.

SSE4.2

Further extensions to the SSE instruction set.

SSE4A

A minor extension to the SSE instruction set adding the instructions EXTRQ, INSERTQ, MOVNTSS, and MOVNTSD.

sticky bit

A bit that is set or cleared by hardware and that remains in that state until explicitly changed by software.

TOP

The x87 top-of-stack pointer.

TSS

Task-state segment.

underflow

The condition in which a floating-point number is smaller in magnitude than the smallest nonzero, positive or negative number that can be represented in the data-type format being used.

vector

(1) A set of integer or floating-point values, called *elements*, that are packed into a single operand. Most of the media instructions support vectors as operands. Vectors are also called *packed* or *SIMD* (single-instruction multiple-data) operands.

(2) An index into an interrupt descriptor table (IDT), used to access exception handlers. Compare *exception*.

VEX

An instruction encoding escape prefix that opens a new extended instruction encoding space, specifies a 64-bit operand size, and provides access to additional registers. See *XOP prefix*.

virtual-8086 mode

A submode of *legacy mode*.

VMCB

Virtual machine control block.

VMM

Virtual machine monitor.

word

Two bytes, or 16 bits.

XOP instructions

Part of the extended SSE instruction set using the XOP prefix. See *Streaming SIMD Extensions*.

XOP prefix

Extended instruction identifier prefix, used by XOP instructions allowing the specification of up to four operands and 128 or 256-bit operand widths.

Registers

In the following list of registers, the names are used to refer either to a given register or to the contents of that register:

AH–DH
: The high 8-bit AH, BH, CH, and DH registers. Compare *AL–DL*.

AL–DL
: The low 8-bit AL, BL, CL, and DL registers. Compare *AH–DH*.

AL–r15B
: The low 8-bit AL, BL, CL, DL, SIL, DIL, BPL, SPL, and R8B–R15B registers, available in 64-bit mode.

BP
: Base pointer register.

CR*n*
: Control register number *n*.

CS
: Code segment register.

eAX–eSP
: The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers. Compare *rAX–rSP*.

EBP
: Extended base pointer register.

EFER
: Extended features enable register.

eFLAGS
: 16-bit or 32-bit flags register. Compare *rFLAGS*.

EFLAGS
: 32-bit (extended) flags register.

eIP
: 16-bit or 32-bit instruction-pointer register. Compare *rIP*.

EIP
: 32-bit (extended) instruction-pointer register.

FLAGS
: 16-bit flags register.

GDTR
: Global descriptor table register.

GPRs
: General-purpose registers. For the 16-bit data size, these are AX, BX, CX, DX, DI, SI, BP, and SP. For the 32-bit data size, these are EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP. For the 64-bit data size, these include RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and R8–R15.

IDTR
: Interrupt descriptor table register.

IP
: 16-bit instruction-pointer register.

LDTR
: Local descriptor table register.

MSR
: Model-specific register.

r8–r15
: The 8-bit R8B–R15B registers, or the 16-bit R8W–R15W registers, or the 32-bit R8D–R15D registers, or the 64-bit R8–R15 registers.

rAX–rSP
: The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers, or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers, or the 64-bit RAX, RBX, RCX, RDX, RDI, RSI, RBP, and RSP registers. Replace the placeholder *r* with nothing for 16-bit size, “E” for 32-bit size, or “R” for 64-bit size.

RAX
: 64-bit version of the EAX register.

RBP
: 64-bit version of the EBP register.

RBX
: 64-bit version of the EBX register.

RCX
: 64-bit version of the ECX register.

RDI
: 64-bit version of the EDI register.

RDX
: 64-bit version of the EDX register.

rFLAGS
: 16-bit, 32-bit, or 64-bit flags register. Compare *RFLAGS*.

RFLAGS
: 64-bit flags register. Compare *rFLAGS*.

rIP
: 16-bit, 32-bit, or 64-bit instruction-pointer register. Compare *RIP*.

RIP
: 64-bit instruction-pointer register.

RSI
: 64-bit version of the ESI register.

RSP
: 64-bit version of the ESP register.

SP
: Stack pointer register.

SS
: Stack segment register.

TPR
: Task priority register (CR8), a new register introduced in the AMD64 architecture to speed interrupt management.

TR
: Task register.

Endian Order

The x86 and AMD64 architectures address memory using little-endian byte-ordering. Multibyte values are stored with their least-significant byte at the lowest byte address, and they are illustrated with their least significant byte at the right side. Strings are illustrated in reverse order, because the addresses of their bytes increase from right to left.

Related Documents

- Peter Abel, *IBM PC Assembly Language and Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Rakesh Agarwal, *80x86 Architecture & Programming: Volume II*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- AMD, *AMD-K6™ MMX™ Enhanced Processor Multimedia Technology*, Sunnyvale, CA, 2000.
- AMD, *3DNow!™ Technology Manual*, Sunnyvale, CA, 2000.
- AMD, *AMD Extensions to the 3DNow!™ and MMX™ Instruction Sets*, Sunnyvale, CA, 2000.
- Don Anderson and Tom Shanley, *Pentium Processor System Architecture*, Addison-Wesley, New York, 1995.
- Nabajyoti Barkakati and Randall Hyde, *Microsoft Macro Assembler Bible*, Sams, Carmel, Indiana, 1992.
- Barry B. Brey, *8086/8088, 80286, 80386, and 80486 Assembly Language Programming*, Macmillan Publishing Co., New York, 1994.
- Barry B. Brey, *Programming the 80286, 80386, 80486, and Pentium Based Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Ralf Brown and Jim Kyle, *PC Interrupts*, Addison-Wesley, New York, 1994.
- Penn Brumm and Don Brumm, *80386/80486 Assembly Language Programming*, Windcrest McGraw-Hill, 1993.
- Geoff Chappell, *DOS Internals*, Addison-Wesley, New York, 1994.
- Chips and Technologies, Inc. *Super386 DX Programmer's Reference Manual*, Chips and Technologies, Inc., San Jose, 1992.
- John Crawford and Patrick Gelsinger, *Programming the 80386*, Sybex, San Francisco, 1987.
- Cyrix Corporation, *5x86 Processor BIOS Writer's Guide*, Cyrix Corporation, Richardson, TX, 1995.
- Cyrix Corporation, *M1 Processor Data Book*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor MMX Extension Opcode Table*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor Data Book*, Cyrix Corporation, Richardson, TX, 1997.
- Ray Duncan, *Extending DOS: A Programmer's Guide to Protected-Mode DOS*, Addison Wesley, NY, 1991.
- William B. Giles, *Assembly Language Programming for the Intel 80xxx Family*, Macmillan, New York, 1991.
- Frank van Gilluwe, *The Undocumented PC*, Addison-Wesley, New York, 1994.
- John L. Hennessy and David A. Patterson, *Computer Architecture*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.
- Thom Hogan, *The Programmer's PC Sourcebook*, Microsoft Press, Redmond, WA, 1991.

- Hal Katircioglu, *Inside the 486, Pentium, and Pentium Pro*, Peer-to-Peer Communications, Menlo Park, CA, 1997.
- IBM Corporation, *486SLC Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *486SLC2 Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *80486DX2 Processor Floating Point Instructions*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *80486DX2 Processor BIOS Writer's Guide*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *Blue Lightning 486DX2 Data Book*, IBM Corporation, Essex Junction, VT, 1994.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, ANSI/IEEE Std 854-1987.
- Muhammad Ali Mazidi and Janice Gillispie Mazidi, *80X86 IBM PC and Compatible Computers*, Prentice-Hall, Englewood Cliffs, NJ, 1997.
- Hans-Peter Messmer, *The Indispensable Pentium Book,* Addison-Wesley, New York, 1995.
- Karen Miller, *An Assembly Language Introduction to Computer Architecture: Using the Intel Pentium*, Oxford University Press, New York, 1999.
- Stephen Morse, Eric Isaacson, and Douglas Albert, *The 80386/387 Architecture*, John Wiley & Sons, New York, 1987.
- NexGen Inc., *Nx586 Processor Data Book*, NexGen Inc., Milpitas, CA, 1993.
- NexGen Inc., *Nx686 Processor Data Book*, NexGen Inc., Milpitas, CA, 1994.
- Bipin Patwardhan, *Introduction to the Streaming SIMD Extensions in the Pentium III*, www.x86.org/articles/sse_pt1/ simd1.htm, June, 2000.
- Peter Norton, Peter Aitken, and Richard Wilton, *PC Programmer's Bible,* Microsoft Press, Redmond, WA, 1993.
- *PharLap 386|ASM Reference Manual*, Pharlap, Cambridge MA, 1993.
- *PharLap TNT DOS-Extender Reference Manual*, Pharlap, Cambridge MA, 1995.
- Sen-Cuo Ro and Sheau-Chuen Her, *i386/i486 Advanced Programming*, Van Nostrand Reinhold, New York, 1993.
- Jeffrey P. Royer, *Introduction to Protected Mode Programming*, course materials for an onsite class, 1992.
- Tom Shanley, *Protected Mode System Architecture*, Addison Wesley, NY, 1996.
- SGS-Thomson Corporation, *80486DX Processor SMM Programming Manual*, SGS-Thomson Corporation, 1995.

- Walter A. Triebel, *The 80386DX Microprocessor*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- John Wharton, *The Complete x86*, MicroDesign Resources, Sebastopol, California, 1994.
- Web sites and newsgroups:
  - www.amd.com
  - news.comp.arch
  - news.comp.lang.asm.x86
  - news.intel.microprocessors
  - news.microsoft

1 64-Bit Media Instruction Reference

This chapter describes the function, mnemonic syntax, opcodes, affected flags, and possible exceptions generated by the 64-bit media instructions. These instructions operate on data located in the 64-bit MMX registers. Most of the instructions operate in parallel on sets of packed elements called *vectors*, although some operate on scalars. The instructions define both integer and floating-point operations, and include the legacy MMX™ instructions, the 3DNow!™ instructions, and the AMD extensions to the MMX and 3DNow! instruction sets.

Each instruction that performs a vector (packed) operation is illustrated with a diagram. Figure 1-1 on page 1 shows the conventions used in these diagrams. The particular diagram shows the PSLLW (packed shift left logical words) instruction.

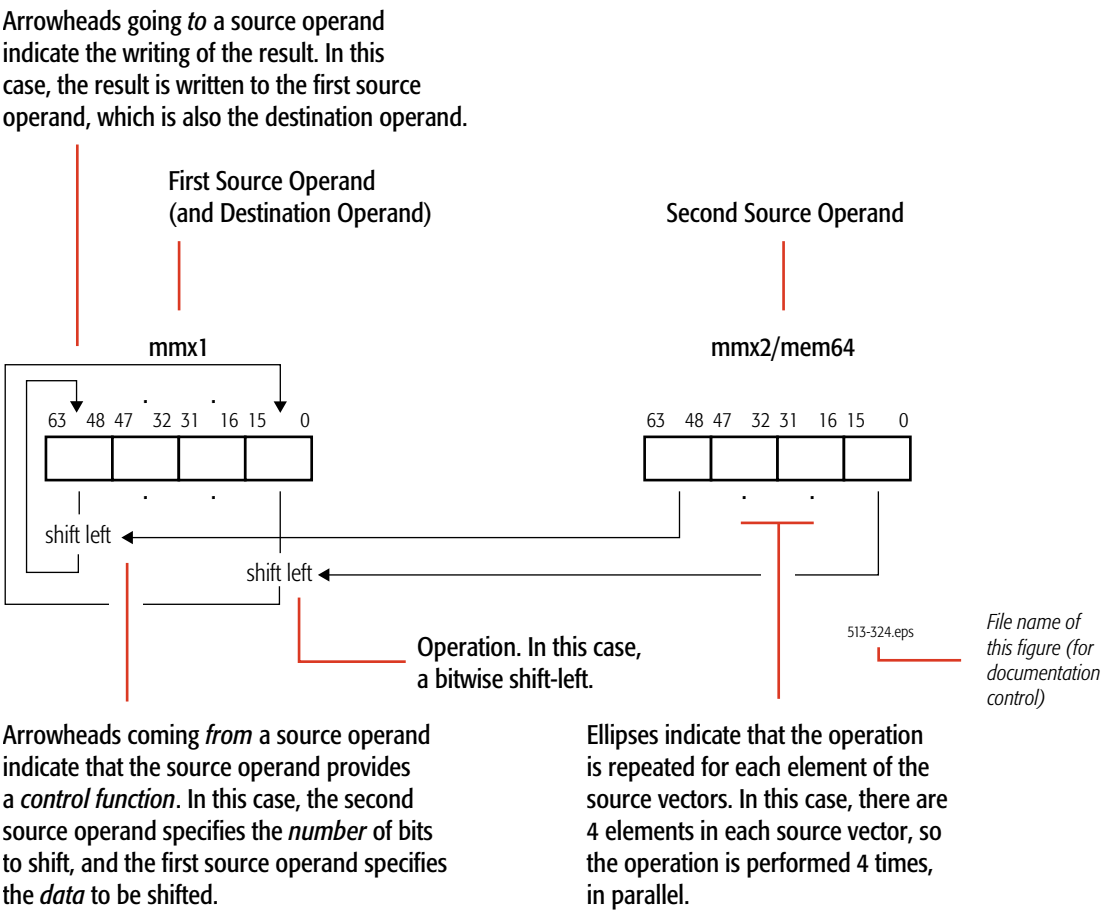


Figure 1-1. Diagram Conventions for 64-Bit Media Instructions

Gray areas in diagrams indicate unmodified operand bits.

Like the 128-bit media instructions, many of the 64-bit instructions independently and simultaneously perform a single operation on multiple elements of a vector and are thus classified as *single-instruction, multiple-data* (SIMD) instructions. A few 64-bit media instructions convert operands in MMX registers to operands in GPR, XMM, or x87 registers (or vice versa), or save or restore MMX state, or reset x87state.

Hardware support of the MMX instruction set and specific optional extensions can be determined by testing specific bits of the value returned in EDX by the CPUID instruction. If a specific bit is set in the return value, the feature is supported on the processor. The following lists the CPUID function numbers and feature bits indicating support for these features:

- MMX Instructions, indicated by EDX[23] returned by CPUID function 0000_0001h and function 8000_0001h.
- AMD Extensions to MMX Instructions, indicated by EDX[22] of CPUID function 8000_0001h.
- SSE1, indicated by EDX[25] of CPUID function 0000_0001h.
- SSE2, indicated by EDX[26] of CPUID function 0000_0001h.
- AMD 3DNow! Instructions, indicated by EDX[31] of CPUID function 8000_0001h.
- AMD Extensions to 3DNow! Instructions, indicated by EDX[30] of CPUID function 8000_0001h.
- FXSAVE and FXRSTOR, indicated by EDX[24] of CPUID function 0000_0001h and function 8000_0001h.

The 64-bit media instructions can be used in legacy mode or long mode. Their use in long mode is available if the following CPUID function return bit is set:

- Long Mode, indicated by EDX[29] of CPUID function 8000_0001h.

For more information on using the CPUID instruction, see the instruction description in Volume 3.

Compilation of 64-bit media programs for execution in 64-bit mode offers four primary advantages: access to the eight extended, 64-bit general-purpose registers (for a register set consisting of GPR0–GPR15), access to the eight extended XMM registers (for a register set consisting of XMM0–XMM15), access to the 64-bit virtual address space, and access to the RIP-relative addressing mode.

For further information, see:

- “64-Bit Media Programming” in Volume 1.
- “Summary of Registers and Data Types” in Volume 3.
- “Notation” in Volume 3.
- “Instruction Prefixes” in Volume 3.

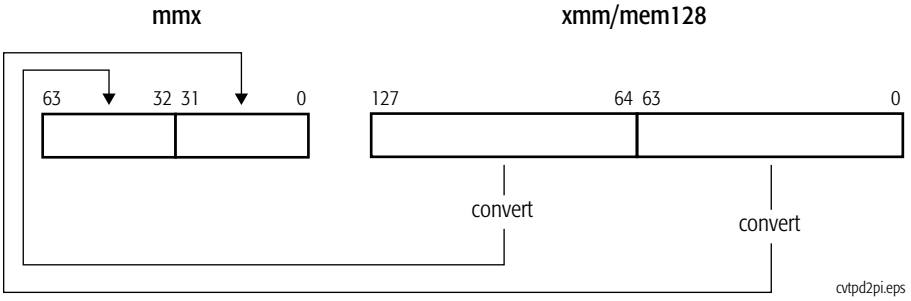
CVTPD2PI Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers

Converts two packed double-precision floating-point values in an XMM register or a 128-bit memory location to two packed 32-bit signed integer values and writes the converted values in an MMX register.

If the result of the conversion is an inexact value, the value is rounded as specified by the rounding control bits (RC) in the MXCSR register. If the floating-point value is a NaN, infinity, or if the result of the conversion is larger than the maximum signed doubleword (-2^{31} to $+2^{31} - 1$), the instruction returns the 32-bit indefinite integer value (8000_0000h) when the invalid-operation exception (IE) is masked.

The CVTPD2PI instruction is an SSE2 instruction. Support for this instruction set is indicated by CPUID Fn0000_0001_EDX[SSE2] = 1. Support for misaligned 16-byte memory accesses is indicated by CPUID Fn8000_0001_ECX[MisAlignSse] = 1.

Mnemonic	Opcode	Description
CVTPD2PI *mmx, xmm2/mem128*	66 0F 2D */r*	Converts packed double-precision floating-point values in an XMM register or 128-bit memory location to packed doubleword integers values in the destination MMX register.



Related Instructions

CVTDQ2PD, CVTPD2DQ, CVTPI2PD, CVTSD2SI, CVTSI2SD, CVTTPD2DQ, CVTTPD2PI, CVTTSD2SI

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that can be set to one or zero is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0.
	X	X	X	There was an unmasked SIMD floating-point exception while CR4.OSXMMEXCPT was cleared to 0. See *SIMD Floating-Point Exceptions*, below, for details.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
	X	X	X	The memory operand was not aligned on a 16-byte boundary while MXCSR.MM = 0.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled with MXCSR.MM = 1.
x87 floating-point exception pending, #MF	X	X	X	An exception is pending due to an x87 floating-point instruction.
SIMD Floating-Point Exception, #XF	X	X	X	There was an unmasked SIMD floating-point exception while CR4.OSXMMEXCPT was set to 1. See *SIMD Floating-Point Exceptions*, below, for details.

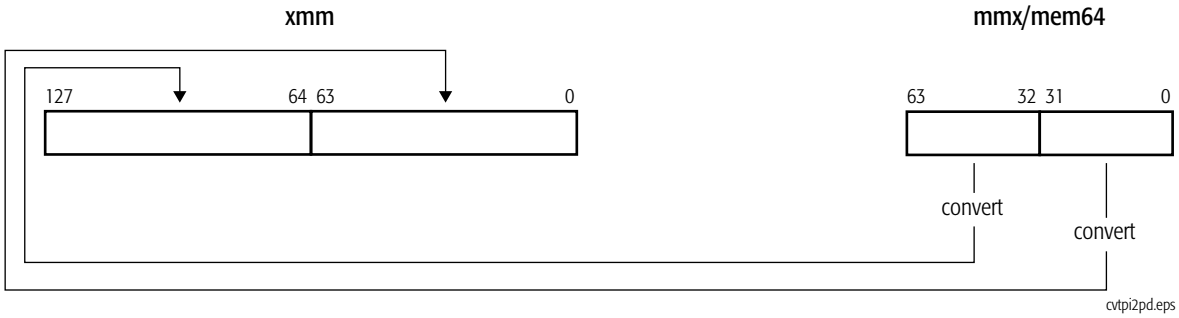
Exception	Real	Virtual 8086	Protected	Cause of Exception
SIMD Floating-Point Exceptions				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value, a QNaN value, or ±infinity.
	X	X	X	A source operand was too large to fit in the destination format.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

CVTPI2PD Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point

Converts two packed 32-bit signed integer values in an MMX register or a 64-bit memory location to two double-precision floating-point values and writes the converted values in an XMM register.

The CVTPI2PD instruction is an SSE2 instruction. Support for this instruction set is indicated by CPUID Fn0000_0001_EDX[SSE2] = 1. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
CVTPI2PD *xmm*, *mmx/mem64*	66 0F 2A */r*	Converts two packed doubleword integer values in an MMX register or 64-bit memory location to two packed double-precision floating-point values in the destination XMM register.



Related Instructions

CVTDQ2PD, CVTPD2DQ, CVTPD2PI, CVTSD2SI, CVTSI2SD, CVTTPD2DQ, CVTTPD2PI, CVTTSD2SI

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

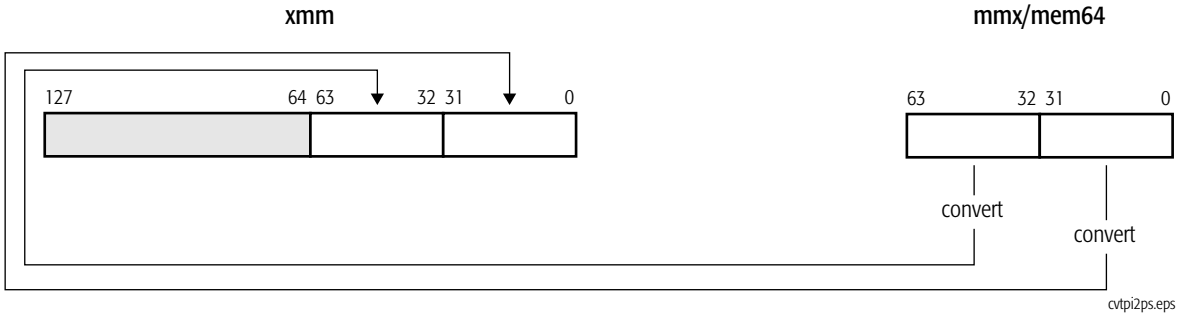
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

CVTPI2PS Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point

Converts two packed 32-bit signed integer values in an MMX register or a 64-bit memory location to two single-precision floating-point values and writes the converted values in the low-order 64 bits of an XMM register. The high-order 64 bits of the XMM register are not modified. If the result of the conversion is an inexact value, the value is rounded as specified by the rounding control bits (RC) in the MXCSR register.

The CVTPI2PS instruction is an SSE1 instruction. Support for this instruction set is indicated by CPUID Fn0000_0001_EDX[SSE] = 1. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
CVTPI2PS *xmm*, *mmx/mem64*	0F 2A */r*	Converts packed doubleword integer values in an MMX register or 64-bit memory location to single-precision floating-point values in the destination XMM register.



Related Instructions

CVTDQ2PS, CVTPS2DQ, CVTPS2PI, CVTSI2SS, CVTSS2SI, CVTTPS2DQ, CVTTPS2PI, CVTTSS2SI

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that can be set to one or zero is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0.
	X	X	X	There was an unmasked SIMD floating-point exception while CR4.OSXMMEXCPT was cleared to 0. See *SIMD Floating-Point Exceptions*, below, for details.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
SIMD Floating-Point Exception, #XF	X	X	X	There was an unmasked SIMD floating-point exception while CR4.OSXMMEXCPT was set to 1. See *SIMD Floating-Point Exceptions*, below, for details.
SIMD Floating-Point Exceptions				
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

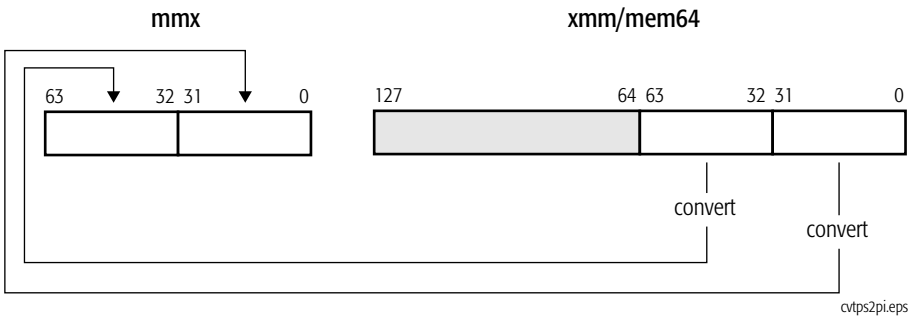
CVTPS2PI Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers

Converts two packed single-precision floating-point values in the low-order 64 bits of an XMM register or a 64-bit memory location to two packed 32-bit signed integers and writes the converted values in an MMX register.

If the result of the conversion is an inexact value, the value is rounded as specified by the rounding control bits (RC) in the MXCSR register. If the floating-point value is a NaN, infinity, or if the result of the conversion is larger than the maximum signed doubleword (-2^{31} to $+2^{31} - 1$), the instruction returns the 32-bit indefinite integer value (8000_0000h) when the invalid-operation exception (IE) is masked.

The CVTPS2PI instruction is an SSE1 instruction. Support for this instruction set is indicated by CPUID Fn0000_0001_EDX[SSE] = 1. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
CVTPS2PI *mmx, xmm/mem64*	0F 2D */r*	Converts packed single-precision floating-point values in an XMM register or 64-bit memory location to packed doubleword integers in the destination MMX register.



Related Instructions

CVTDQ2PS, CVTPI2PS, CVTPS2DQ, CVTSI2SS, CVTSS2SI, CVTTPS2DQ, CVTTPS2PI, CVTTSS2SI

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that can be set to one or zero is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0.
	X	X	X	There was an unmasked SIMD floating-point exception while CR4.OSXMMEXCPT was cleared to 0. See *SIMD Floating-Point Exceptions*, below, for details.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
SIMD Floating-Point Exception, #XF	X	X	X	There was an unmasked SIMD floating-point exception while CR4.OSXMMEXCPT was set to 1. See *SIMD Floating-Point Exceptions*, below, for details.
SIMD Floating-Point Exceptions				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value, a QNaN value, or ±infinity.
	X	X	X	A source operand was too large to fit in the destination format.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

CVTTPD2PI Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers, Truncated

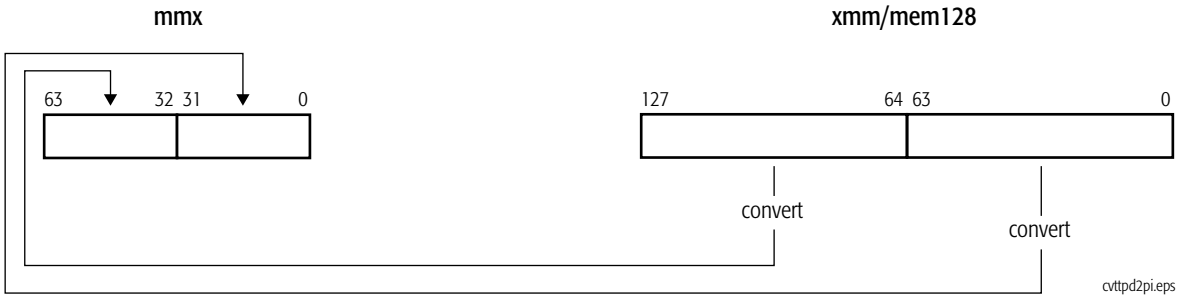
Converts two packed double-precision floating-point values in an XMM register or a 128-bit memory location to two packed 32-bit signed integer values and writes the converted values in an MMX register.

If the result of the conversion is an inexact value, the value is truncated (rounded toward zero). If the floating-point value is a NaN, infinity, or if the result of the conversion is larger than the maximum signed doubleword (-2^{31} to $+2^{31} - 1$), the instruction returns the 32-bit indefinite integer value (8000_0000h) when the invalid-operation exception (IE) is masked.

The CVTTPD2PI instruction is an SSE2 instruction. Support for this instruction set is indicated by CPUID Fn0000_0001_EDX[SSE2] = 1. Support for misaligned 16-byte memory accesses is indicated by CPUID Fn8000_0001_ECX[MisAlignSse] = 1.

See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
CVTTPD2PI *mmx, xmm/mem128*	66 0F 2C */r*	Converts packed double-precision floating-point values in an XMM register or 128-bit memory location to packed doubleword integer values in the destination MMX register. Inexact results are truncated.



Related Instructions

CVTDQ2PD, CVTPD2DQ, CVTPD2PI, CVTPI2PD, CVTSD2SI, CVTSI2SD, CVTTPD2DQ, CVTTSD2SI

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that can be set to one or zero is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0.
	X	X	X	There was an unmasked SIMD floating-point exception while CR4.OSXMMEXCPT was cleared to 0. See *SIMD Floating-Point Exceptions*, below, for details.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
	X	X	X	The memory operand was not aligned on a 16-byte boundary while MXCSR.MM = 0.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled with MXCSR.MM = 1.
x87 floating-point exception pending, #MF	X	X	X	An exception is pending due to an x87 floating-point instruction.
SIMD Floating-Point Exception, #XF	X	X	X	There was an unmasked SIMD floating-point exception while CR4.OSXMMEXCPT was set to 1. See *SIMD Floating-Point Exceptions*, below, for details.

Exception	Real	Virtual 8086	Protected	Cause of Exception
SIMD Floating-Point Exceptions				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value, a QNaN value, or ±infinity.
	X	X	X	A source operand was too large to fit in the destination format.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

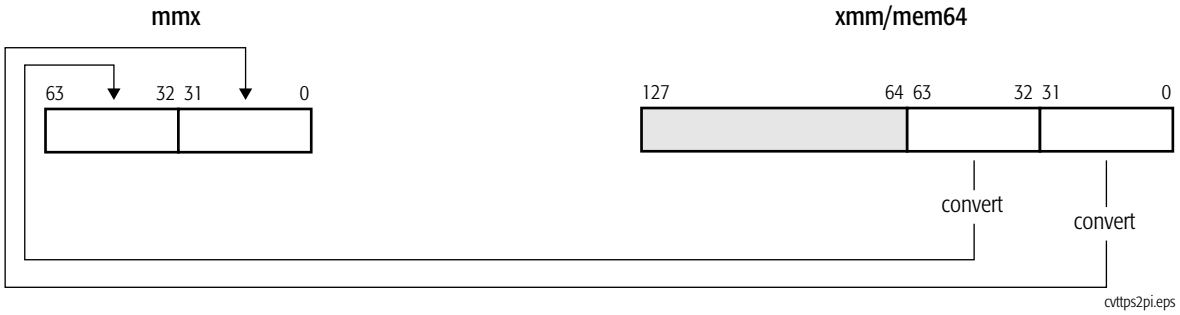
CVTTPS2PI Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers, Truncated

Converts two packed single-precision floating-point values in the low-order 64 bits of an XMM register or a 64-bit memory location to two packed 32-bit signed integer values and writes the converted values in an MMX register.

If the result of the conversion is an inexact value, the value is truncated (rounded toward zero). If the floating-point value is a NaN, infinity, or if the result of the conversion is larger than the maximum signed doubleword (-2^{31} to $+2^{31} - 1$), the instruction returns the 32-bit indefinite integer value (8000_0000h) when the invalid-operation exception (IE) is masked.

The CVTTPS2PI instruction is an SSE1 instruction. Support for this instruction set is indicated by CPUID Fn0000_0001_EDX[SSE] = 1. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
CVTTPS2PI *mmx, xmm/mem64*	0F 2C */r*	Converts packed single-precision floating-point values in an XMM register or 64-bit memory location to doubleword integer values in the destination MMX register. Inexact results are truncated.



Related Instructions

CVTDQ2PS, CVTPI2PS, CVTPS2DQ, CVTPS2PI, CVTSI2SS, CVTSS2SI, CVTTPS2DQ, CVTTSS2SI

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M					M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: *A flag that can be set to one or zero is M (modified). Unaffected flags are blank.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0.
	X	X	X	There was an unmasked SIMD floating-point exception while CR4.OSXMMEXCPT was cleared to 0. See *SIMD Floating-Point Exceptions*, below, for details.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
SIMD Floating-Point Exception, #XF	X	X	X	There was an unmasked SIMD floating-point exception while CR4.OSXMMEXCPT was set to 1. See *SIMD Floating-Point Exceptions*, below, for details.
SIMD Floating-Point Exceptions				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value, a QNaN value, or ±infinity.
	X	X	X	A source operand was too large to fit in the destination format.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

EMMS Exit Multimedia State

Clears the MMX state by setting the state of the x87 stack registers to *empty* (tag-bit encoding of all 1s for all MMX registers) indicating that the contents of the registers are available for a new procedure, such as an x87 floating-point procedure. This setting of the tag bits is referred to as “clearing the MMX state”.

Because the MMX registers and tag word are shared with the x87 floating-point instructions, software should execute an EMMS instruction to clear the MMX state before executing code that includes x87 floating-point instructions.

The functions of the EMMS and FEMMS instructions are identical.

For details about the setting of x87 tag bits, see “Media and x87 Processor State” in Volume 2.

The EMMS instruction is an MMX™ instruction. Support for the MMX instruction subset is indicated by CPUID Fn0000_0001_EDX[MMX] = 1 or CPUID Fn8000_0001_EDX[MMX] = 1. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
EMMS	0F 77	Clears the MMX state.

Related Instructions

FEMMS (a 3DNow! instruction)

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

FEMMS Fast Exit Multimedia State

Clears the MMX state by setting the state of the x87 stack registers to *empty* (tag-bit encoding of all 1s for all MMX registers) indicating that the contents of the registers are available for a new procedure, such as an x87 floating-point procedure. This setting of the tag bits is referred to as “clearing the MMX state”.

Because the MMX registers and tag word are shared with the x87 floating-point instructions, software should execute an EMMS or FEMMS instruction to clear the MMX state before executing code that includes x87 floating-point instructions.

The functions of the FEMMS and EMMS instructions are identical. The FEMMS instruction is supported for backward-compatibility with certain AMD processors. Software that must be both compatible with both AMD and non-AMD processors should use the EMMS instruction.

FEMMS is a 3DNow! instruction. Support for this instruction subset is indicated by CPUID Fn8000_0001_EDX[3DNow] = 1. See “CPUID” in Volume 3 for more information about the CPUID instruction.

For details about the setting of x87 tag bits, see “Media and x87 Processor State” in Volume 2.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

EMMS

Mnemonic	Opcode	Description
FEMMS	0F 0E	Clears MMX state.

Related Instructions

EMMS

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

FRSTOR Floating-Point Restore x87 and MMX™ State

Restores the complete x87 state from memory starting at the specified address, as stored by a previous call to FNSAVE. The x87 state occupies 94 or 108 bytes of memory depending on whether the processor is operating in real or protected mode and whether the operand-size attribute is 16-bit or 32-bit. Because the MMX registers are mapped onto the low 64 bits of the x87 floating-point registers, this operation also restores the MMX state.

If FRSTOR results in set exception flags in the loaded x87 status word register, and these exceptions are unmasked in the x87 control word register, a floating-point exception occurs when the next floating-point instruction is executed (except for the no-wait floating-point instructions).

To avoid generating exceptions when loading a new environment, use the FCLEX or FNCLEX instruction to clear the exception flags in the x87 status word before storing that environment.

For details about the memory image restored by FRSTOR, see “Media and x87 Processor State” in Volume 2.

Mnemonic	Opcode	Description
FRSTOR *mem94/108env*	DD /4	Load the x87 state from *mem94/108env*.

Related Instructions

FSAVE, FNSAVE, FXSAVE, FXRSTOR

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	M	Loaded from memory.
C1	M	Loaded from memory.
C2	M	Loaded from memory.
C3	M	Loaded from memory.

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

FSAVE (FNSAVE) Floating-Point Save x87 and MMX™ State

Stores the complete x87 state to memory starting at the specified address and reinitializes the x87 state. The x87 state requires 94 or 108 bytes of memory, depending upon whether the processor is operating in real or protected mode and whether the operand-size attribute is 16-bit or 32-bit. Because the MMX registers are mapped onto the low 64 bits of the x87 floating-point registers, this operation also saves the MMX state. For details about the memory image saved by FNSAVE, see “Media and x87 Processor State” in Volume 2.

The FNSAVE instruction does not wait for pending unmasked x87 floating-point exceptions to be processed. Processor interrupts should be disabled before using this instruction.

Assemblers usually provide an FSAVE macro that expands into the instruction sequence:

```
WAIT                    ; Opcode 9B
FNSAVE destination      ; Opcode DD /6
```

The WAIT (9Bh) instruction checks for pending x87 exceptions and calls an exception handler, if necessary. The FNSAVE instruction then stores the x87 state to the specified destination.

Mnemonic	Opcode	Description
FNSAVE *mem94/108env*	DD /6	Copy the x87 state to *mem94/108env* without checking for pending floating-point exceptions, then reinitialize the x87 state.
FSAVE *mem94/108env*	9B DD /6	Copy the x87 state to *mem94/108env* after checking for pending floating-point exceptions, then reinitialize the x87 state.

Related Instructions

FRSTOR, FXSAVE, FXRSTOR

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	0	
C1	0	
C2	0	
C3	0	

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

FXRSTOR Restore XMM, MMX™, and x87 State

Restores the XMM, MMX, and x87 state from a 16-byte aligned area in memory pointed to by the source operand. The data loaded from memory is the state information previously saved using the FXSAVE instruction. Restoring data with FXRSTOR that had been previously saved with an FSAVE (rather than FXSAVE) instruction results in an incorrect restoration. If the source pointer is not 16-byte aligned, execution may be inhibited either by an #AC exception at CPL=3 if alignment checking is enabled, otherwise by a #GP(0) exception.

If FXRSTOR results in set exception flags in the loaded x87 status word register, and these exceptions are unmasked in the x87 control word register, a floating-point exception occurs when the next floating-point instruction is executed (except for the no-wait floating-point instructions).

If the restored MXCSR register contains a set bit in an exception status flag, and the corresponding exception mask bit is cleared (indicating an unmasked exception), loading the MXCSR register does not cause a SIMD floating-point exception (#XF).

FXRSTOR does not restore the x87 error pointers (last instruction pointer, last data pointer, and last opcode), except when FXRSTOR sets FSW.ES=1 after recomputing it from the error mask bits in FCW and error status bits in FSW.

The architecture supports two 512-bit memory formats for FXRSTOR, a 64-bit format that loads XMM0-XMM15, and a 32-bit legacy format that loads only XMM0-XMM7. If FXRSTOR is executed in 64-bit mode, the 64-bit format is used, otherwise the 32-bit format is used. When the 64-bit format is used, if the operand-size is 64-bit, FXRSTOR loads the x87 pointer registers as *offset64*, otherwise it loads them as *sel:offset32*. For details about the memory format used by FXRSTOR, see "Saving Media and x87 Processor State" in Volume 2.

If the fast-FXSAVE/FXRSTOR (FFXSR) feature is enabled in EFER, FXRSTOR does not restore the XMM registers (XMM0-XMM15) when executed in 64-bit mode at CPL 0. MXCSR is restored whether fast-FXSAVE/FXRSTOR is enabled or not.

Support for the fast-FXSAVE/FXRSTOR feature is indicated by CPUID Fn8000_0001_EDX[FFXSR] = 1.

If the operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 is cleared to 0, the saved image of XMM0–XMM15 and MXCSR is not loaded into the processor. A general-protection exception occurs if the FXRSTOR instruction attempts to load non-zero values into reserved MXCSR bits. Software can use MXCSR_MASK to determine which bits of MXCSR are reserved. For details on the MXCSR_MASK, see “SSE, MMX, and x87 Programming” in Volume 2.

Support for this instruction is implementation-specific. CPUID Fn8000_0001_EDX[FXSR] = 1 or CPUID Fn0000_0001_EDX[FXSR] = 1 indicates support for the FXSAVE and FXRSTOR instructions. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
FXRSTOR *mem512env*	0F AE /1	Restores XMM, MMX™, and x87 state from 512-byte memory location.

Related Instructions

FWAIT, FXSAVE

rFLAGS Affected

None

MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Note: *A flag that can be set to one or zero is M (modified). Unaffected flags are blank. Shaded fields are reserved.*																

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The FXSAVE/FXRSTOR instructions are not supported, as indicated by EDX[FXSR] = 0, returned by CPUID Fn0000_0001 or CPUID Fn8000_0001.
Device not available, #NM	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit, or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded the data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
	X	X	X	The memory operand was not aligned on a 16-byte boundary. At CPL=3 (including virtual 8086 mode), this will be overridden by a #AC exception if alignment checking is enabled.
	X	X	X	Ones were written to the reserved bits in MXCSR.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment Check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

FXSAVE Save XMM, MMX, and x87 State

Saves the XMM, MMX, and x87 state to a 16-byte aligned area in memory pointed to by the source operand. If the destination pointer is not 16-byte aligned, execution may be inhibited by an #AC exception at CPL=3 if alignment checking is enabled, otherwise by a #GP(0) exception.

Unlike FSAVE and FNSAVE, FXSAVE does not alter the x87 tag bits. The contents of the saved MMX/x87 data registers are retained, thus indicating that the registers may be valid (or whatever other value the x87 tag bits indicated prior to the save). To invalidate the contents of the MMX/x87 data registers after FXSAVE, software must execute an FINIT instruction. Also, FXSAVE (like FNSAVE) does not check for pending unmasked x87 floating-point exceptions. An FWAIT instruction can be used for this purpose.

FXSAVE does not save the x87 pointer registers (last instruction pointer, last data pointer, and last opcode), except in the relatively rare cases in which the exception-summary (ES) bit in the x87 status word is set to 1, indicating that an unmasked x87 exception has occurred.

The architecture supports two 512-bit memory formats for FXSAVE, a 64-bit format that saves XMM0-XMM15, and a 32-bit legacy format that saves only XMM0-XMM7. If FXSAVE is executed in 64-bit mode, the 64-bit format is used, otherwise the 32-bit format is used. When the 64-bit format is used, if the operand-size is 64-bit, FXSAVE saves the x87 pointer registers as *offset64*, otherwise it saves them as *sel:offset32*. For more details about the memory format used by FXSAVE, see “Saving Media and x87 Execution Unit State” in Volume 2.

If the fast-FXSAVE/FXRSTOR (FFXSR) feature is enabled in EFER, FXSAVE does not save the XMM registers (XMM0-XMM15) when executed in 64-bit mode at CPL 0. MXCSR is saved whether fast-FXSAVE/FXRSTOR is enabled or not.

Support for the fast-FXSAVE/FXRSTOR feature is indicated by CPUID Fn8000_0001_EDX[FFXSR] = 1.

If the operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 is cleared to 0, FXSAVE does not save the image of XMM0–XMM15 or MXCSR. For details about the CR4.OSFXSR bit, see “FXSAVE/FXRSTOR Support (OSFXSR) Bit” in Volume 2.

Support for this instruction is implementation-specific. CPUID Fn8000_0001_EDX[FXSR] = 1 or CPUID Fn0000_0001_EDX[FXSR] = 1 indicates support for the FXSAVE and FXRSTOR instructions. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
FXSAVE *mem512env*	0F AE /0	Saves XMM, MMX, and x87 state to 512-byte memory location.

Related Instructions

FINIT, FNSAVE, FRSTOR, FSAVE, FXRSTOR, LDMXCSR, STMXCSR

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The FXSAVE/FXRSTOR instructions are not supported, as indicated by EDX[FXSR] = 0, returned by CPUID Fn0000_0001 or CPUID Fn8000_0001.
Device not available, #NM	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit, or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded the data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
	X	X	X	The memory operand was not aligned on a 16-byte boundary. At CPL=3 (including virtual 8086 mode), this will be overridden by an #AC exception if alignment checking is enabled.
			X	The destination operand was in a non-writable segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment Check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MASKMOVQ Masked Move Quadword

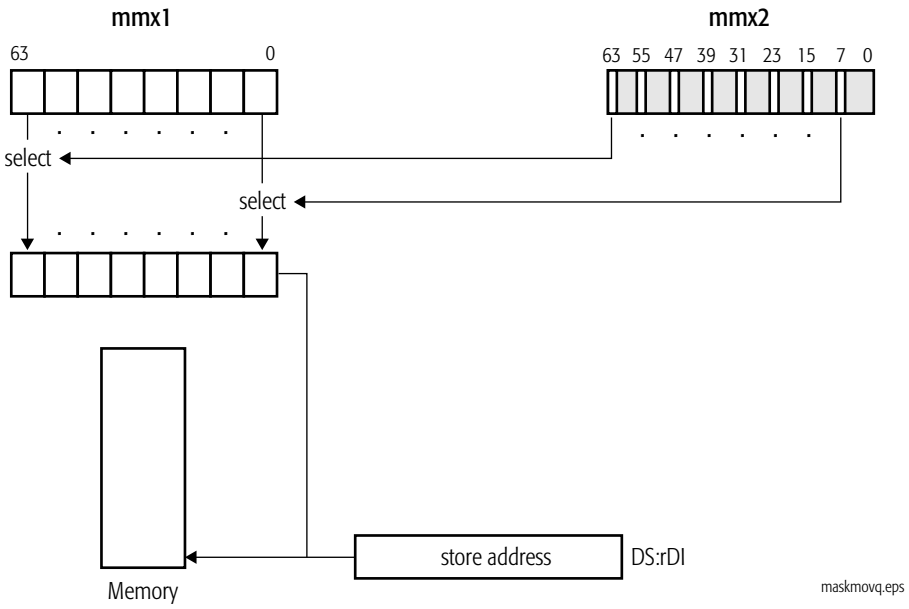
Stores bytes from the first source operand, as selected by the second source operand, to a memory location specified in the DS:rDI registers (except that DS is ignored in 64-bit mode). The first source operand is an MMX register, and the second source operand is another MMX register. The most-significant bit (msb) of each byte in the second source operand specifies the store (1 = store, 0 = no store) of the corresponding byte of the first source operand.

Exception and trap behavior for the elements not selected for storage to memory are implementation dependent. For instance, a given implementation may signal a data breakpoint or a page fault for bytes that are zero-masked and not actually written.

MASKMOVQ implicitly uses weakly-ordered, write-combining buffering for the data, as described in "Buffering and Combining Memory Writes" in Volume 2. If the stored data is shared by multiple processors, this instruction should be used together with a fence instruction in order to ensure data coherency (refer to "Cache and TLB Management" in Volume 2).

The MASKMOVQ instruction is an AMD extension to MMX™ instruction set and is an SSE1 instruction. Support for AMD extensions to the MMX instruction subset is indicated by CPUID Fn8000_0001_EDX[MmxExt] = 1. See "CPUID" in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
MASKMOVQ *mmx1*, *mmx2*	0F F7 */r*	Store bytes from an MMX register, selected by the most-significant bit of the corresponding byte in another MMX register, to DS:rDI.



Related Instructions

MASKMOVDQU

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0 and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.

Exception	Real	Virtual 8086	Protected	Cause of Exception
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MOVD Move Doubleword or Quadword

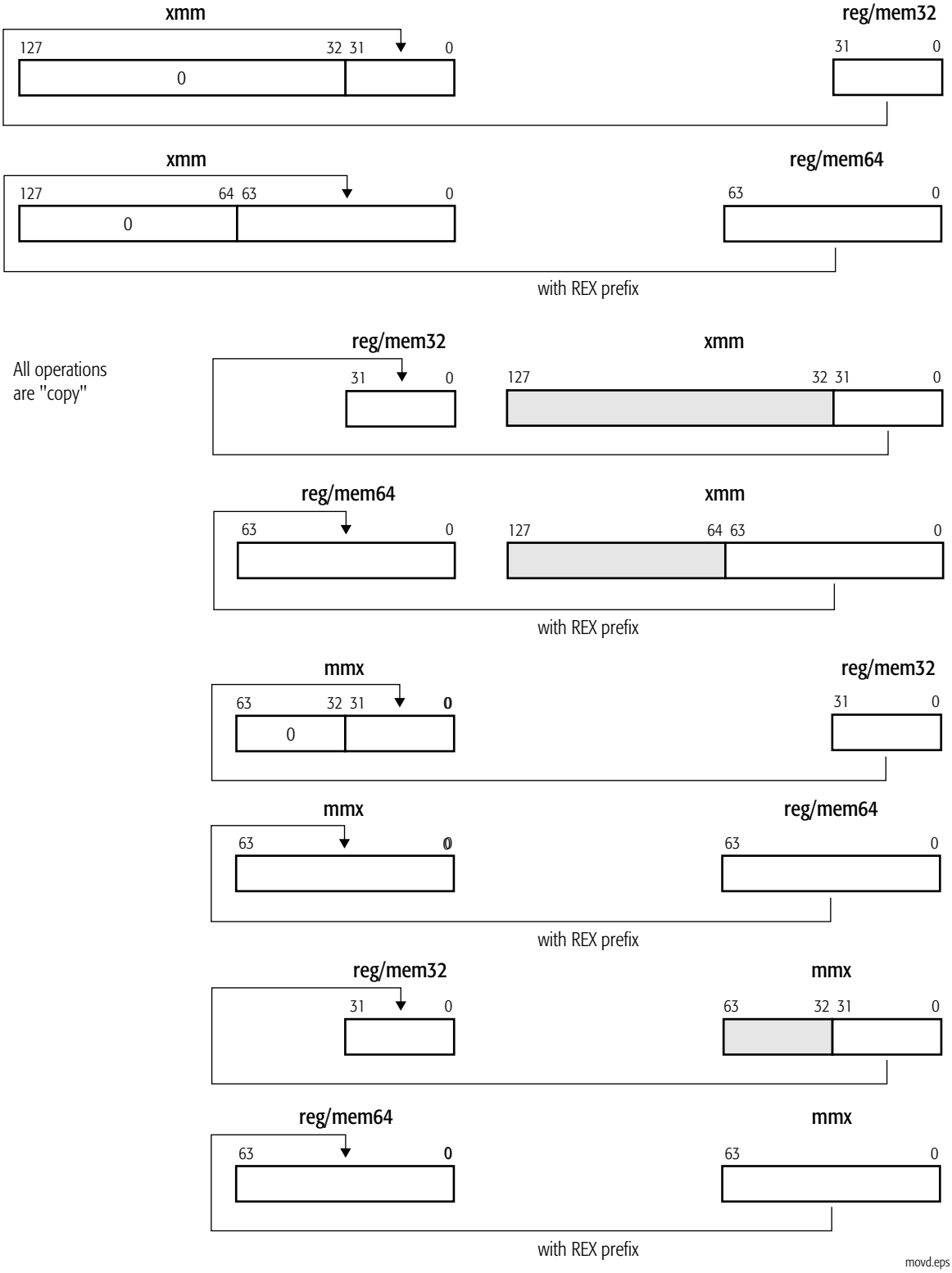
Moves a 32-bit or 64-bit value in one of the following ways:

- from a 32-bit or 64-bit general-purpose register or memory location to the low-order 32 or 64 bits of an XMM register, with zero-extension to 128 bits
- from the low-order 32 or 64 bits of an XMM to a 32-bit or 64-bit general-purpose register or memory location
- from a 32-bit or 64-bit general-purpose register or memory location to the low-order 32 bits (with zero-extension to 64 bits) or the full 64 bits of an MMX register
- from the low-order 32 or the full 64 bits of an MMX register to a 32-bit or 64-bit general-purpose register or memory location.

The MOVD instruction is a member of both the MMX and the SSE2 instruction sets. The presence of this instruction set is indicated by EDX[MMX] = 1 returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
MOVD *mmx*, *reg/mem32*	0F 6E */r*	Move 32-bit value from a general-purpose register or 32-bit memory location to an MMX register.
MOVD *mmx*, *reg/mem64*	0F 6E */r*	Move 64-bit value from a general-purpose register or 64-bit memory location to an MMX register.
MOVD *reg/mem32*, *mmx*	0F 7E */r*	Move 32-bit value from an MMX register to a 32-bit general-purpose register or memory location.
MOVD *reg/mem64*, *mmx*	0F 7E */r*	Move 64-bit value from an MMX register to a 64-bit general-purpose register or memory location.

The following diagrams illustrate the operation of the MOVD instruction.



Related Instructions

MOVDQA, MOVDQU, MOVDQ2Q, MOVQ, MOVQ2DQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

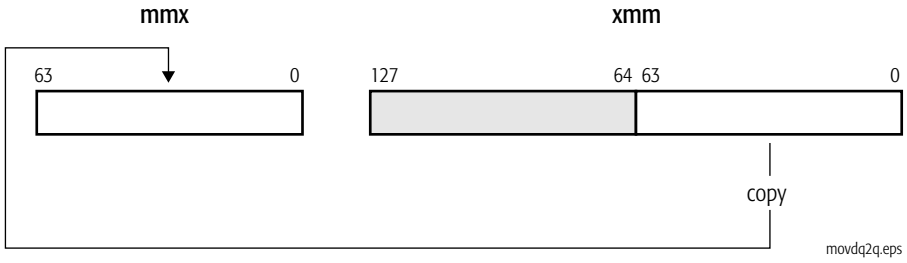
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode, #UD	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0 returned by CPUID function 0000_0001h or 8000_0001h.
	X	X	X	The SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The instruction used XMM registers while CR4.OSFXSR=0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MOVDQ2Q Move Quadword to Quadword

Moves the low-order 64-bit value in an XMM register to a 64-bit MMX register.

The MOVDQ2Q instruction is an SSE2 instruction. Support for this instruction subset is indicated by CPUID Fn0000_0001_EDX[SSE2] = 1. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
MOVDQ2Q *mmx*, *xmm*	F2 0F D6 /r	Moves low-order 64-bit value from an XMM register to the destination MMX register.



Related Instructions

MOVD, MOVDQA, MOVDQU, MOVQ, MOVQ2DQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.

Exception	Real	Virtual 8086	Protected	Cause of Exception
General protection, #GP	X	X	X	The destination operand was in non-writable segment.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

MOVNTQ Move Non-Temporal Quadword

Stores a 64-bit MMX register value into a 64-bit memory location. This instruction indicates to the processor that the data is non-temporal, and is unlikely to be used again soon. The processor treats the store as a write-combining (WC) memory write, which minimizes cache pollution. The exact method by which cache pollution is minimized depends on the hardware implementation of the instruction. For further information, see “Memory Optimization” in Volume 1.

MOVNTQ is weakly-ordered with respect to other instructions that operate on memory. Software should use an SFENCE instruction to force strong memory ordering of MOVNTQ with respect to other stores.

MOVNTQ implicitly uses weakly-ordered, write-combining buffering for the data, as described in “Buffering and Combining Memory Writes” in Volume 2. For data that is shared by multiple processors, this instruction should be used together with a fence instruction in order to ensure data coherency (refer to “Cache and TLB Management” in Volume 2).

The MOVNTQ instruction is a member of both the AMD MMX extensions and the SSE1 instruction sets. Support for the SSE1 instruction subset is indicated by CPUID Fn0000_0001_EDX[SSE] = 1. Support for AMD’s extensions to the MMX instruction subset is indicated by CPUID Fn8000_0001_EDX[MmxExt] = 1. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
MOVNTQ *mem64*, *mmx*	0F E7 */r*	Stores a 64-bit MMX register value into a 64-bit memory location, minimizing cache pollution.

Related Instructions

MOVNTDQ, MOVNTI, MOVNTPD, MOVNTPS

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0 and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

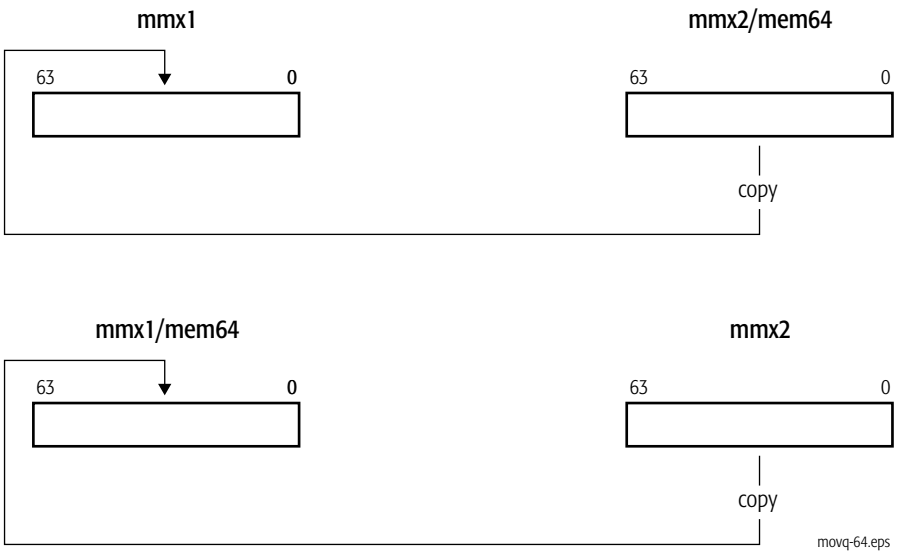
MOVQ Move Quadword

Moves a 64-bit value:

- from an MMX register or 64-bit memory location to another MMX register, or
- from an MMX register to another MMX register or 64-bit memory location.

The MOVQ instruction is an MMX™ instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
MOVQ *mmx1*, *mmx2*/*mem64*	0F 6F */r*	Moves 64-bit value from an MMX register or memory location to an MMX register.
MOVQ *mmx1*/*mem64*, *mmx2*	0F 7F */r*	Moves 64-bit value from an MMX register to an MMX register or memory location.



Related Instructions

MOVD, MOVDQA, MOVDQU, MOVDQ2Q, MOVQ2DQ

rFLAGS Affected

None

Exceptions

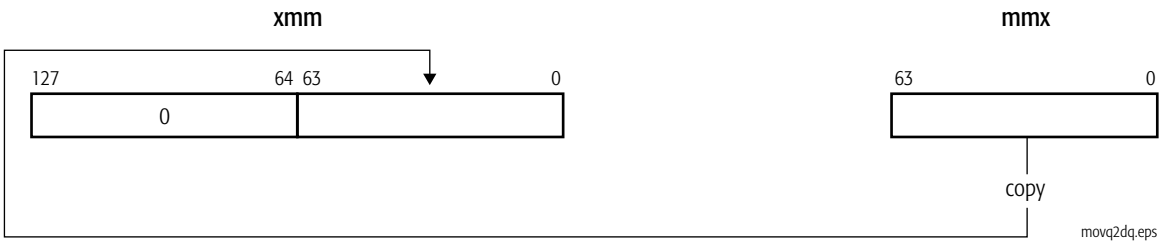
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeds the stack segment limit or is non-canonical.
General protection, #GP	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
			X	The destination operand was in a non-writable segment.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

MOVQ2DQ Move Quadword to Quadword

Moves a 64-bit value from an MMX register to the low-order 64 bits of an XMM register, with zero-extension to 128 bits.

The MOVQ2DQ instruction is an SSE2 instruction. Support for this instruction subset is indicated by CPUID Fn0000_0001_EDX[SSE2] = 1. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
MOVQ2DQ *xmm*, *mmx*	F3 0F D6 /r	Moves 64-bit value from an MMX register to an XMM register.



Related Instructions

MOVD, MOVDQA, MOVDQU, MOVDQ2Q, MOVQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0.
	X	X	X	The SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

PACKSSDW Pack with Saturation Signed Doubleword to Word

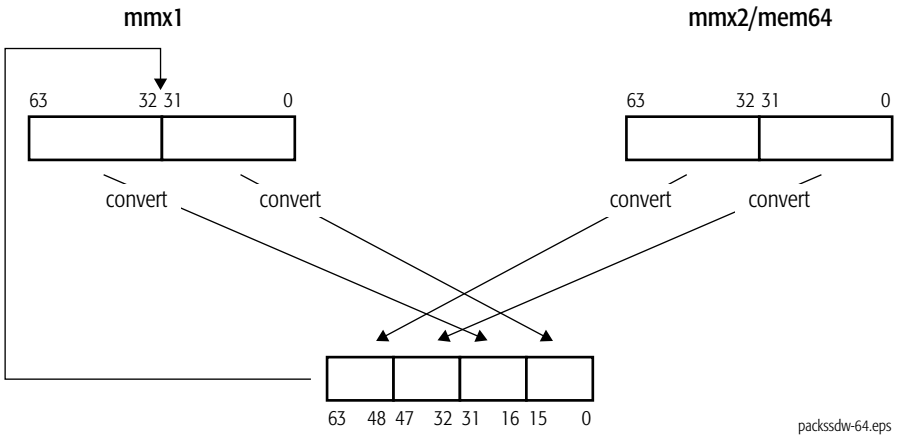
Converts each 32-bit signed integer in the first and second source operands to a 16-bit signed integer and packs the converted values into words in the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

Converted values from the first source operand are packed into the low-order words of the destination, and the converted values from the second source operand are packed into the high-order words of the destination.

For each packed value in the destination, if the value is larger than the largest signed 16-bit integer, it is saturated to 7FFFh, and if the value is smaller than the smallest signed 16-bit integer, it is saturated to 8000h.

The PACKSSDW instruction is an MMX™ instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PACKSSDW *mmx1*, *mmx2/mem64*	0F 6B */r*	Packs 32-bit signed integers in an MMX register and another MMX register or 64-bit memory location into 16-bit signed integers in an MMX register.



Related Instructions

PACKSSWB, PACKUSWB

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PACKSSWB Pack with Saturation Signed Word to Byte

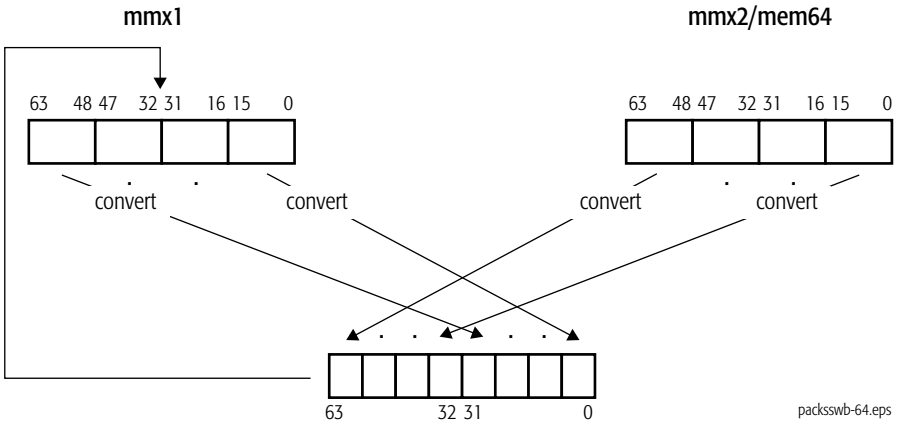
Converts each 16-bit signed integer in the first and second source operands to an 8-bit signed integer and packs the converted values into bytes in the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

Converted values from the first source operand are packed into the low-order bytes of the destination, and the converted values from the second source operand are packed into the high-order bytes of the destination.

For each packed value in the destination, if the value is larger than the largest signed 8-bit integer, it is saturated to 7Fh, and if the value is smaller than the smallest signed 8-bit integer, it is saturated to 80h.

The PACKSSWB instruction is an MMX instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PACKSSWB *mmx1*, *mmx2*/*mem64*	0F 63 */r*	Packs 16-bit signed integers in an MMX register and another MMX register or 64-bit memory location into 8-bit signed integers in an MMX register.



Related Instructions

PACKSSDW, PACKUSWB

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PACKUSWB Pack with Saturation Signed Word to Unsigned Byte

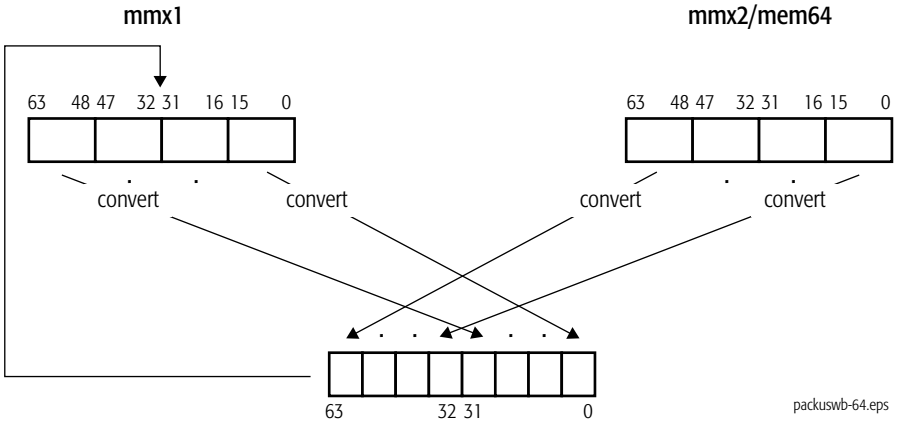
Converts each 16-bit signed integer in the first and second source operands to an 8-bit unsigned integer and packs the converted values into bytes in the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

Converted values from the first source operand are packed into the low-order bytes of the destination, and the converted values from the second source operand are packed into the high-order bytes of the destination.

For each packed value in the destination, if the value is larger than the largest unsigned 8-bit integer, it is saturated to FFh, and if the value is smaller than the smallest unsigned 8-bit integer, it is saturated to 00h.

The PACKUSWB instruction is an MMX™ instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PACKUSWB *mmx1*, *mmx2/mem64*	0F 67 */r*	Packs 16-bit signed integers in an MMX register and another MMX register or 64-bit memory location into 8-bit unsigned integers in an MMX register.



Related Instructions

PACKSSDW, PACKSSWB

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

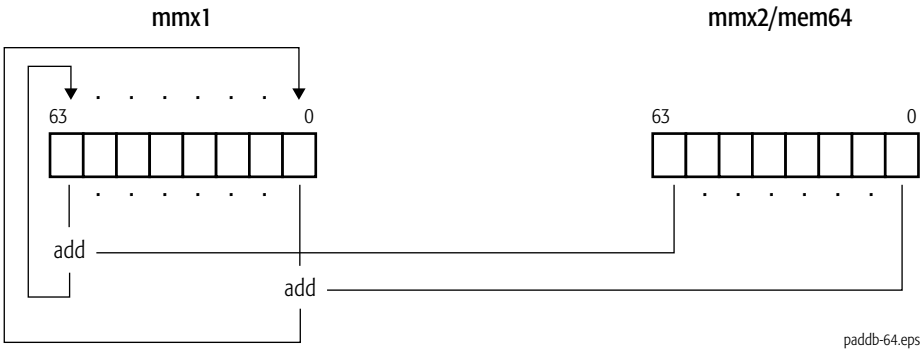
PADDB Packed Add Bytes

Adds each packed 8-bit integer value in the first source operand to the corresponding packed 8-bit integer in the second source operand and writes the integer result of each addition in the corresponding byte of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PADDB instruction operates on both signed and unsigned integers. If the result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 8 bits of each result are written in the destination.

The PADDB instruction is an MMX™ instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PADDB *mmx1*, *mmx2/mem64*	0F FC */r*	Adds packed byte integer values in an MMX register and another MMX register or 64-bit memory location and writes the result in the destination MMX register.



Related Instructions

PADDD, PADDQ, PADDSB, PADDSW, PADDUSB, PADDUSW, PADDW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

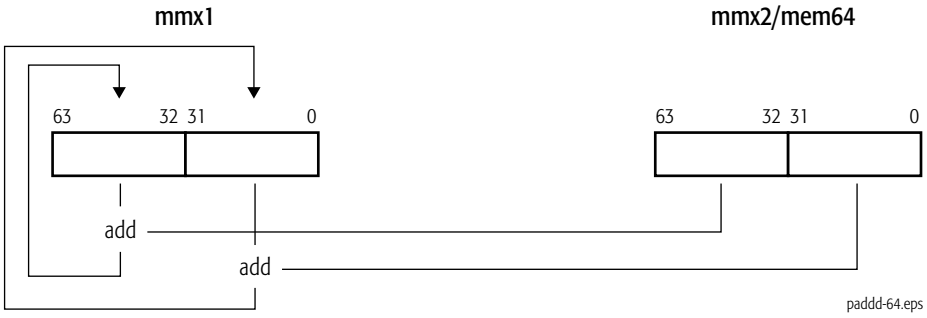
PADDD Packed Add Doublewords

Adds each packed 32-bit integer value in the first source operand to the corresponding packed 32-bit integer in the second source operand and writes the integer result of each addition in the corresponding doubleword of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PADDD instruction operates on both signed and unsigned integers. If the result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 32 bits of each result are written in the destination.

The PADDD instruction is an MMX™ instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PADDD *mmx1*, *mmx2*/*mem64*	0F FE */r*	Adds packed 32-bit integer values in an MMX register and another MMX register or 64-bit memory location and writes the result in the destination MMX register.



Related Instructions

PADDB, PADDQ, PADDSB, PADDSW, PADDUSB, PADDUSW, PADDW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PADDQ Packed Add Quadwords

Adds each packed 64-bit integer value in the first source operand to the corresponding packed 64-bit integer in the second source operand and writes the integer result of each addition in the corresponding quadword of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PADDQ instruction operates on both signed and unsigned integers. If the result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 64 bits of each result are written in the destination.

The PADDQ instruction is an SSE2 instruction. The presence of this instruction set is indicated by a CPUID Fn0000_0001_EDX[SSE2] = 1. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PADDQ *mmx1*, *mmx2*/*mem64*	0F D4 */r*	Adds 64-bit integer value in an MMX register and another MMX register or 64-bit memory location and writes the result in the destination MMX register.

Related Instructions

PADDB, PADDD, PADDSB, PADDSW, PADDUSB, PADDUSW, PADDW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

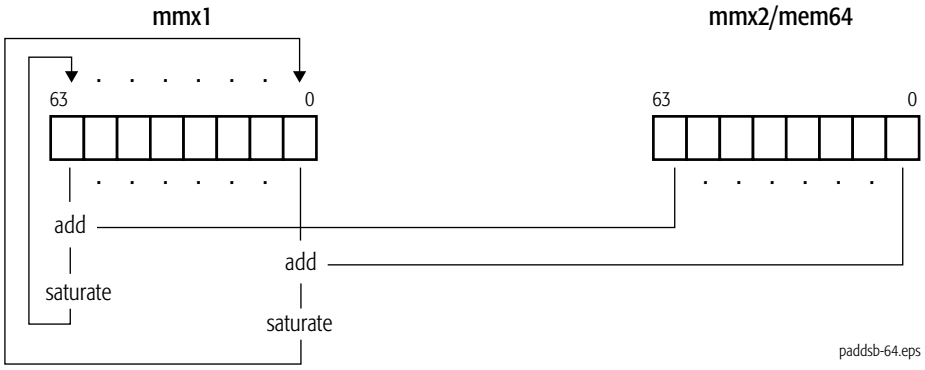
PADDSB Packed Add Signed with Saturation Bytes

Adds each packed 8-bit signed integer value in the first source operand to the corresponding packed 8-bit signed integer in the second source operand and writes the signed integer result of each addition in the corresponding byte of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

For each packed value in the destination, if the value is larger than the largest representable signed 8-bit integer, it is saturated to 7Fh, and if the value is smaller than the smallest signed 8-bit integer, it is saturated to 80h.

The PADDSB instruction is an MMX™ instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PADDSB *mmx1*, *mmx2*/*mem64*	0F EC */r*	Adds packed byte signed integer values in an MMX register and another MMX register or 64-bit memory location and writes the result in the destination MMX register.



Related Instructions

PADDB, PADDD, PADDQ, PADDSW, PADDUSB, PADDUSW, PADDW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

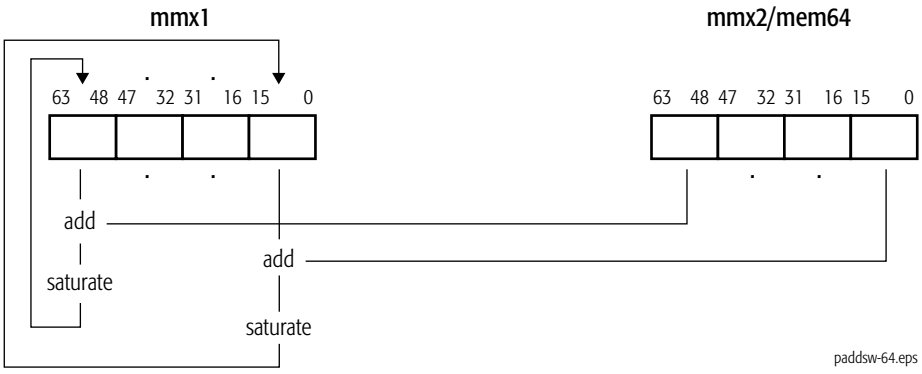
PADDSW Packed Add Signed with Saturation Words

Adds each packed 16-bit signed integer value in the first source operand to the corresponding packed 16-bit signed integer in the second source operand and writes the signed integer result of each addition in the corresponding word of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

For each packed value in the destination, if the value is larger than the largest representable signed 16-bit integer, it is saturated to 7FFFh, and if the value is smaller than the smallest signed 16-bit integer, it is saturated to 8000h.

The PADDSW instruction is an MMX™ instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PADDSW *mmx1*, *mmx2/mem64*	0F ED */r*	Adds packed 16-bit signed integer values in an MMX register and another MMX register or 64-bit memory location and writes the result in the destination MMX register.



Related Instructions

PADDB, PADDD, PADDQ, PADDSB, PADDUSB, PADDUSW, PADDW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

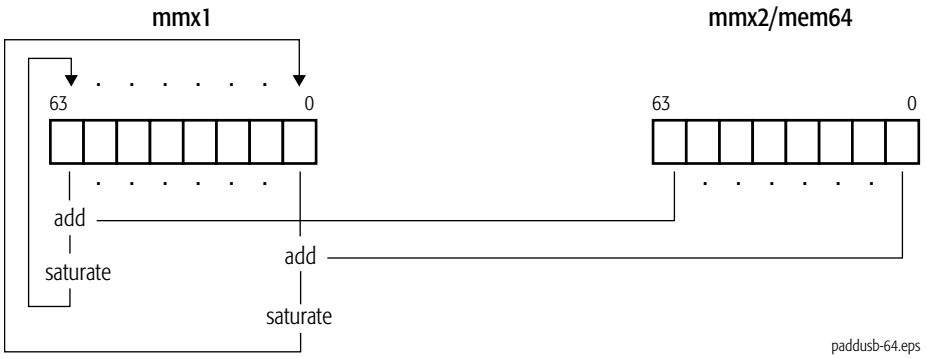
PADDUSB Packed Add Unsigned with Saturation Bytes

Adds each packed 8-bit unsigned integer value in the first source operand to the corresponding packed 8-bit unsigned integer in the second source operand and writes the unsigned integer result of each addition in the corresponding byte of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

For each packed value in the destination, if the value is larger than the largest unsigned 8-bit integer, it is saturated to FFh.

The PADDUSB instruction is an MMX™ instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PADDUSB *mmx1*, *mmx2/mem64*	0F DC */r*	Adds packed byte unsigned integer values in an MMX register and another MMX register or 64-bit memory location and writes the result in the destination MMX register.



Related Instructions

PADDB, PADDD, PADDQ, PADDSB, PADDSW, PADDUSW, PADDW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

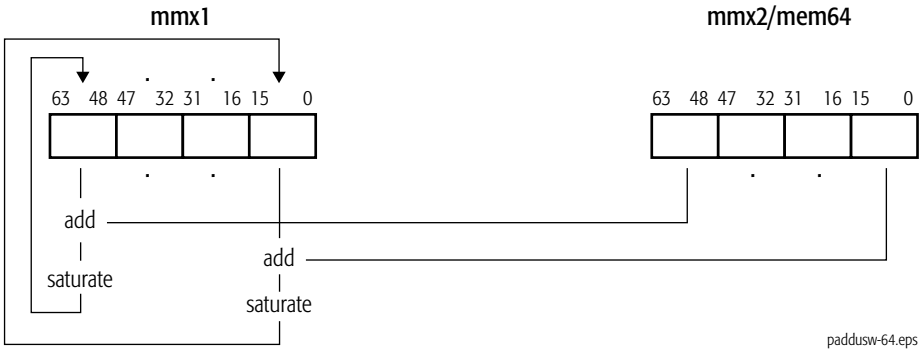
PADDUSW Packed Add Unsigned with Saturation Words

Adds each packed 16-bit unsigned integer value in the first source operand to the corresponding packed 16-bit unsigned integer in the second source operand and writes the unsigned integer result of each addition in the corresponding word of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

For each packed value in the destination, if the value is larger than the largest unsigned 16-bit integer, it is saturated to FFFFh.

The PADDUSW instruction is an MMX™ instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PADDUSW *mmx1*, *mmx2*/*mem64*	0F DD */r*	Adds packed 16-bit unsigned integer values in an MMX register and another MMX register or 64-bit memory location and writes result in the destination MMX register.



Related Instructions

PADDB, PADDD, PADDQ, PADDSB, PADDSW, PADDUSB, PADDW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

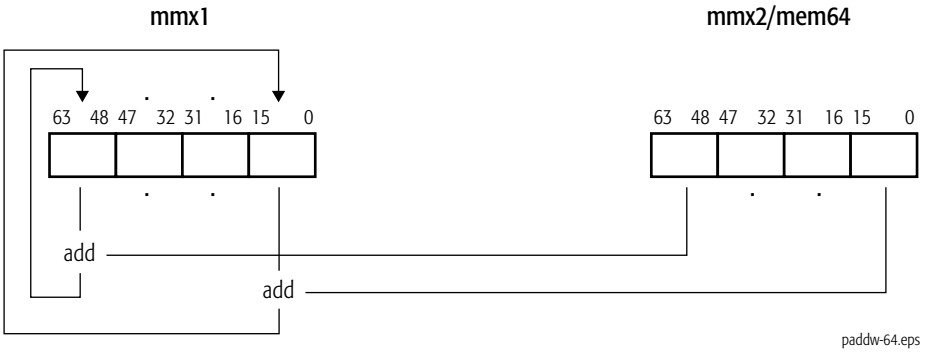
PADDW Packed Add Words

Adds each packed 16-bit integer value in the first source operand to the corresponding packed 16-bit integer in the second source operand and writes the integer result of each addition in the corresponding word of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

This instruction operates on both signed and unsigned integers. If the result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 16 bits of the result are written in the destination.

The PADDW instruction is an MMX™ instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PADDW *mmx1*, *mmx2*/*mem64*	0F FD */r*	Adds packed 16-bit integer values in an MMX register and another MMX register or 64-bit memory location and writes the result in the destination MMX register.



Related Instructions

PADDB, PADDD, PADDQ, PADDSB, PADDSW, PADDUSB, PADDUSW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PAND Packed Logical Bitwise AND

Performs a bitwise logical AND of the values in the first and second source operands and writes the result in the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PAND instruction is an MMX™ instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PAND *mmx1*, *mmx2/mem64*	0F DB */r*	Performs bitwise logical AND of values in an MMX register and in another MMX register or 64-bit memory location and writes the result in the destination MMX register.

Related Instructions

PANDN, POR, PXOR

rFLAGS Affected

None

Exceptions

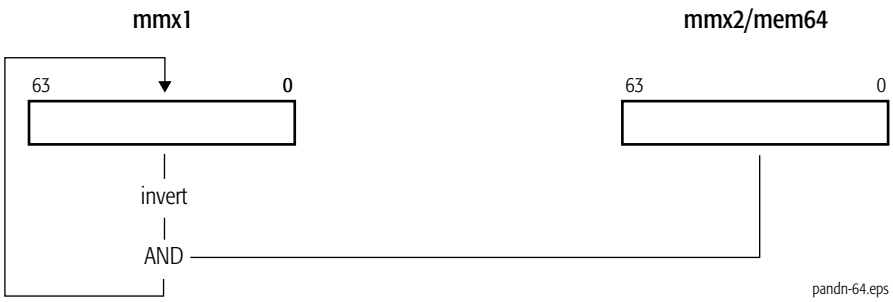
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PANDN Packed Logical Bitwise AND NOT

Performs a bitwise logical AND of the value in the second source operand and the one's complement of the value in the first source operand and writes the result in the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PANDN instruction is an MMX™ instruction. Support for this instruction subset is indicated by EDX[MMX] = 1, as returned by CPUID Fn0000_0001 or CPUID Fn8000_0001. See "CPUID" in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PANDN *mmx1*, *mmx2*/*mem64*	0F DF */r*	Performs bitwise logical AND NOT of values in an MMX register and in another MMX register or 64-bit memory location and writes the result in the destination MMX register.



Related Instructions

PAND, POR, PXOR

rFLAGS Affected

None

Exceptions

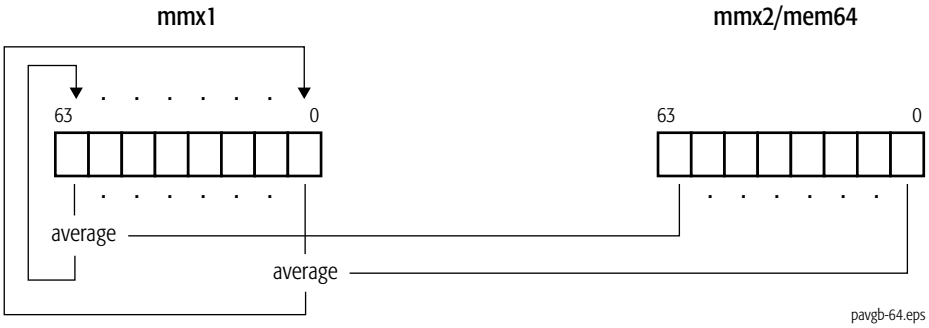
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PAVGB Packed Average Unsigned Bytes

Computes the rounded average of each packed unsigned 8-bit integer value in the first source operand and the corresponding packed 8-bit unsigned integer in the second source operand and writes each average in the corresponding byte of the destination (first source). The average is computed by adding each pair of operands, adding 1 to the 9-bit temporary sum, and then right-shifting the temporary sum by one bit position. The destination and source operands are an MMX register and another MMX register or 64-bit memory location.

The PAVGB instruction is a member of both the AMD MMX™ extensions and the SSE1 instruction sets. Support for the SSE1 instruction subset is indicated by CPUID Fn0000_0001_EDX[SSE] = 1. Support for AMD's extensions to the MMX instruction subset is indicated by CPUID Fn8000_0001_EDX[MmxExt] = 1. See "CPUID" in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PAVGB *mmx1*, *mmx2/mem64*	0F E0 */r*	Averages packed 8-bit unsigned integer values in an MMX register and another MMX register or 64-bit memory location and writes the result in the destination MMX register.



Related Instructions

PAVGW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0; and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PAVGUSB Packed Average Unsigned Bytes

Computes the rounded-up average of each packed unsigned 8-bit integer value in the first source operand and the corresponding packed 8-bit unsigned integer in the second source operand and writes each average in the corresponding byte of the destination (first source). The average is computed by adding each pair of operands, adding 1 to the 9-bit temporary sum, and then right-shifting the temporary sum by one bit position. The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location.

The PAVGUSB instruction performs a function identical to the 64-bit version of the PAVGB instruction, although the two instructions have different opcodes. PAVGUSB is a 3DNow! instruction. It is useful for pixel averaging in MPEG-2 motion compensation and video scaling operations.

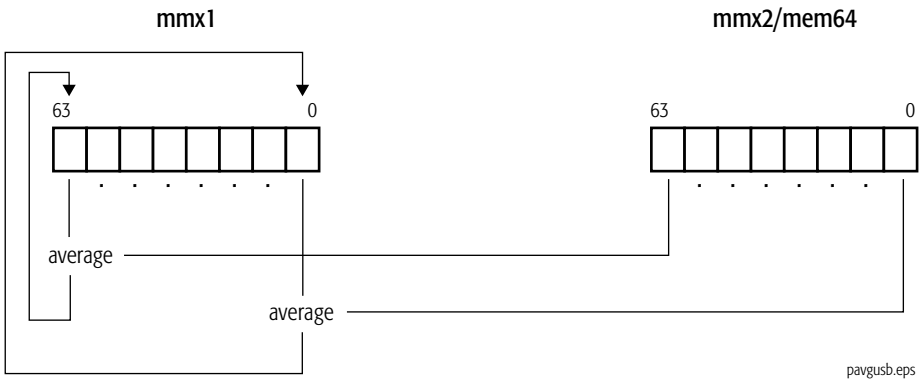
The PAVGUSB instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by a CPUID feature bit. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

PAVGB

Mnemonic	Opcode	Description
PAVGUSB *mmx1*, *mmx2*/*mem64*	0F 0F /r BF	Averages packed 8-bit unsigned integer values in an MMX register and another MMX register or 64-bit memory location and writes the result in the destination MMX register.



Related Instructions

None

rFLAGS Affected

None

Exceptions

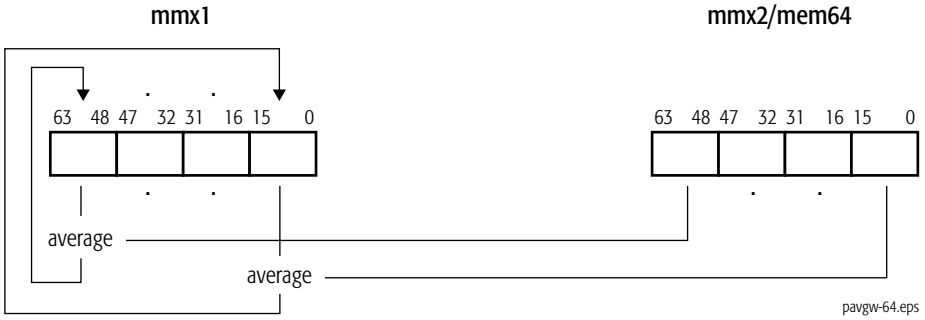
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PAVGW Packed Average Unsigned Words

Computes the rounded average of each packed unsigned 16-bit integer value in the first source operand and the corresponding packed 16-bit unsigned integer in the second source operand and writes each average in the corresponding word of the destination (first source). The average is computed by adding each pair of operands, adding 1 to the 17-bit temporary sum, and then right-shifting the temporary sum by one bit position. The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PAVGW instruction is a member of both the AMD MMX™ extensions and the SSE1 instruction sets. Support for the SSE1 instruction subset is indicated by CPUID Fn0000_0001_EDX[SSE] = 1. Support for AMD's extensions to the MMX instruction subset is indicated by CPUID Fn8000_0001_EDX[MmxExt] = 1. See "CPUID" in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PAVGW *mmx1*, *mmx2*/*mem64*	0F E3 */r*	Averages packed 16-bit unsigned integer values in an MMX register and another MMX register or 64-bit memory location and writes the result in the destination MMX register.



Related Instructions

PAVGB

rFLAGS Affected

None

Exceptions

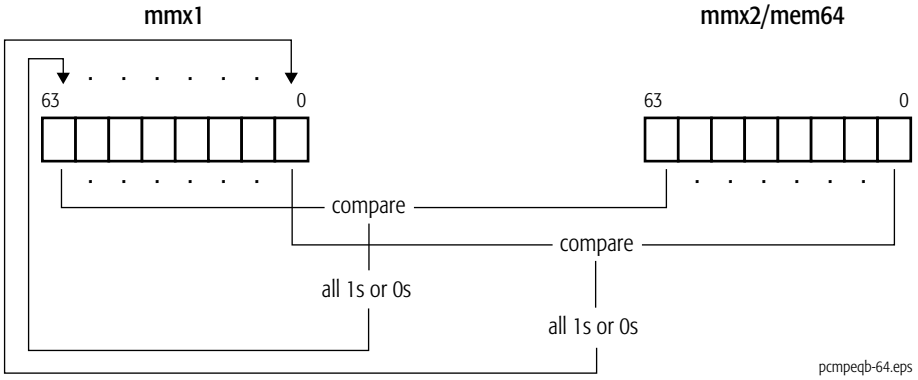
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0; and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PCMPEQB Packed Compare Equal Bytes

Compares corresponding packed bytes in the first and second source operands and writes the result of each compare in the corresponding byte of the destination (first source). For each pair of bytes, if the values are equal, the result is all 1s. If the values are not equal, the result is all 0s. The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PCMPEQB instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PCMPEQB *mmx1*, *mmx2/mem64*	0F 74 */r*	Compares packed bytes in an MMX register and an MMX register or 64-bit memory location.



Related Instructions

PCMPEQD, PCMPEQW, PCMPGTB, PCMPGTD, PCMPGTW

rFLAGS Affected

None

Exceptions

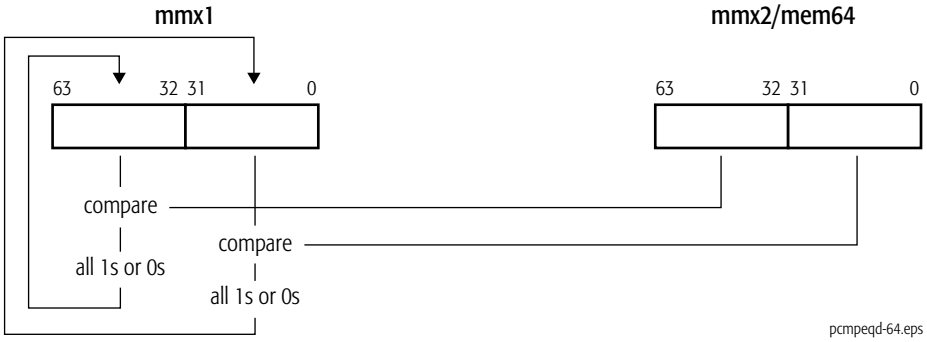
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PCMPEQD Packed Compare Equal Doublewords

Compares corresponding packed 32-bit values in the first and second source operands and writes the result of each compare in the corresponding 32 bits of the destination (first source). For each pair of doublewords, if the values are equal, the result is all 1s. If the values are not equal, the result is all 0s. The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PCMPEQD instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PCMPEQD *mmx1*, *mmx2/mem64*	0F 76 */r*	Compares packed doublewords in an MMX register and an MMX register or 64-bit memory location.



Related Instructions

PCMPEQB, PCMPEQW, PCMPGTB, PCMPGTD, PCMPGTW

rFLAGS Affected

None

Exceptions

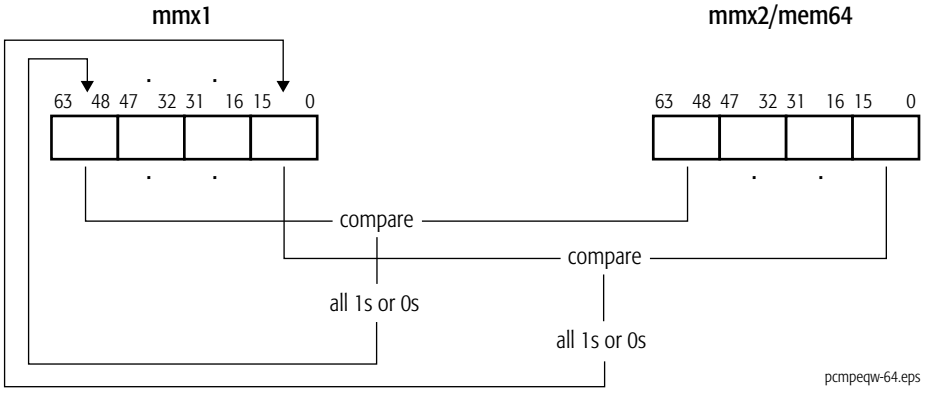
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PCMPEQW Packed Compare Equal Words

Compares corresponding packed 16-bit values in the first and second source operands and writes the result of each compare in the corresponding 16 bits of the destination (first source). For each pair of words, if the values are equal, the result is all 1s. If the values are not equal, the result is all 0s. The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PCMPEQW instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PCMPEQW *mmx1*, *mmx2/mem64*	0F 75 */r*	Compares packed 16-bit values in an MMX register and an MMX register or 64-bit memory location.



Related Instructions

PCMPEQB, PCMPEQD, PCMPGTB, PCMPGTD, PCMPGTW

rFLAGS Affected

None

Exceptions

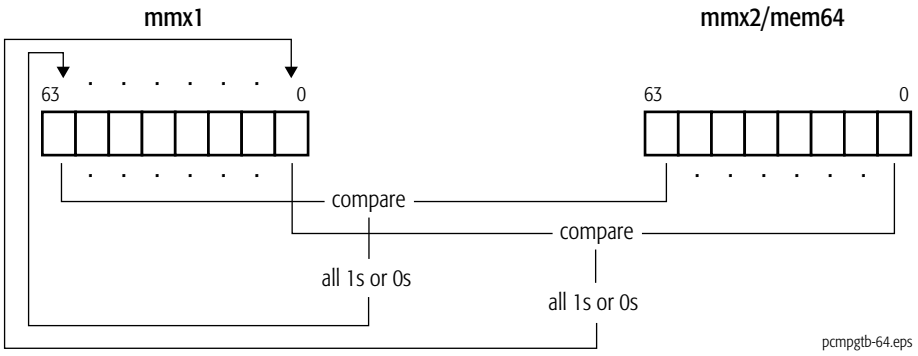
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PCMPGTB Packed Compare Greater Than Signed Bytes

Compares corresponding packed signed bytes in the first and second source operands and writes the result of each compare in the corresponding byte of the destination (first source). For each pair of bytes, if the value in the first source operand is greater than the value in the second source operand, the result is all 1s. If the value in the first source operand is less than or equal to the value in the second source operand, the result is all 0s. The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PCMPGTB instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PCMPGTB *mmx1*, *mmx2/mem64*	0F 64 */r*	Compares packed signed bytes in an MMX register and an MMX register or 64-bit memory location.



Related Instructions

PCMPEQB, PCMPEQD, PCMPEQW, PCMPGTD, PCMPGTW

rFLAGS Affected

None

Exceptions

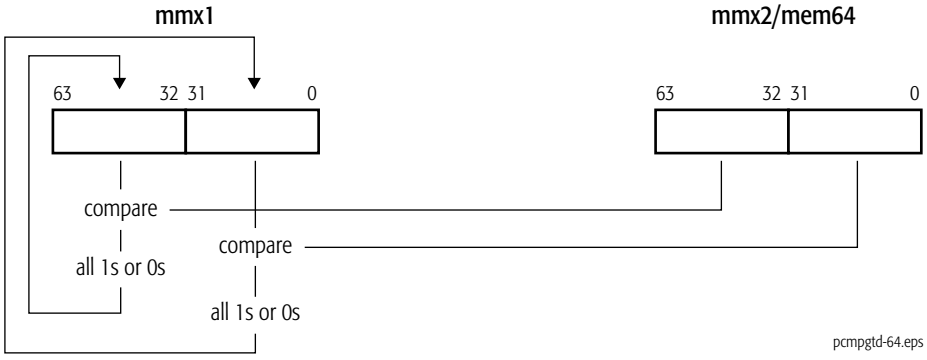
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PCMPGTD Packed Compare Greater Than Signed Doublewords

Compares corresponding packed signed 32-bit values in the first and second source operands and writes the result of each compare in the corresponding 32 bits of the destination (first source). For each pair of doublewords, if the value in the first source operand is greater than the value in the second source operand, the result is all 1s. If the value in the first source operand is less than or equal to the value in the second source operand, the result is all 0s. The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PCMPGTD instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PCMPGTD *mmx1*, *mmx2/mem64*	0F 66 */r*	Compares packed signed 32-bit values in an MMX register and an MMX register or 64-bit memory location.



Related Instructions

PCMPEQB, PCMPEQD, PCMPEQW, PCMPGTB, PCMPGTW

rFLAGS Affected

None

Exceptions

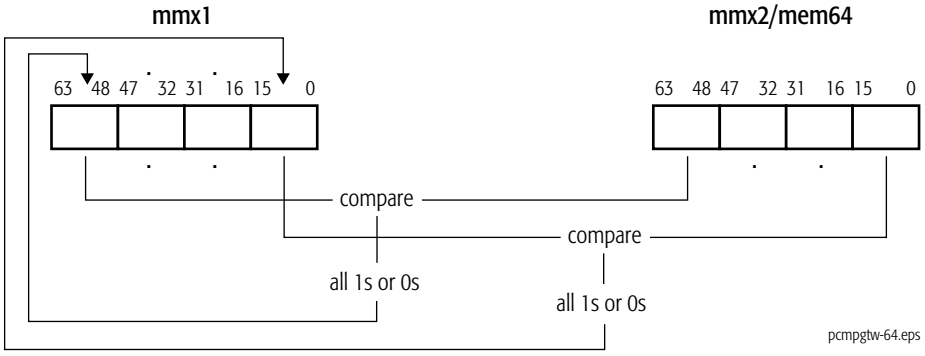
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PCMPGTW Packed Compare Greater Than Signed Words

Compares corresponding packed signed 16-bit values in the first and second source operands and writes the result of each compare in the corresponding 16 bits of the destination (first source). For each pair of words, if the value in the first source operand is greater than the value in the second source operand, the result is all 1s. If the value in the first source operand is less than or equal to the value in the second source operand, the result is all 0s. The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PCMPGTW instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PCMPGTW *mmx1*, *mmx2/mem64*	0F 65 */r*	Compares packed signed 16-bit values in an MMX register and an MMX register or 64-bit memory location.



Related Instructions

PCMPEQB, PCMPEQD, PCMPEQW, PCMPGTB, PCMPGTD

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PEXTRW Extract Packed Word

Extracts a 16-bit value from an MMX register, as selected by the immediate byte operand (as shown in Table 1-1) and writes it to the low-order word of a 32-bit general-purpose register, with zero-extension to 32 bits.

The PEXTRW instruction is a member of both the AMD MMX™ extensions and the SSE1 instruction set. Support for the SSE1 instruction subset is indicated by CPUID Fn0000_0001_EDX[SSE] = 1. Support for AMD's extensions to the MMX instruction subset is indicated by CPUID Fn8000_0001_EDX[MmxExt] = 1. See "CPUID" in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PEXTRW *reg32*, *mmx*, *imm8*	0F C5 */r ib*	Extracts a 16-bit value from an MMX register and writes it to low-order 16 bits of a general-purpose register.

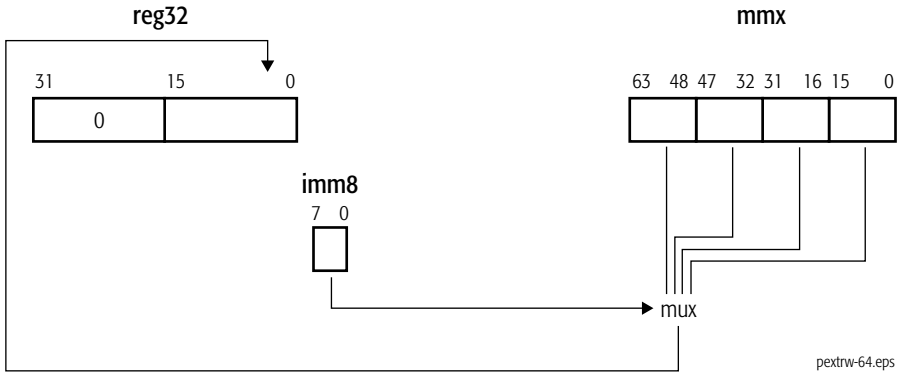


Table 1-1. Immediate-Byte Operand Encoding for 64-Bit PEXTRW

Immediate-Byte Bit Field	Value of Bit Field	Source Bits Extracted
1–0	0	15–0
	1	31–16
	2	47–32
	3	63–48

Related Instructions

PINSRW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0; and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

PF2ID Packed Floating-Point to Integer Doubleword Converson

Converts two packed single-precision floating-point values in an MMX register or a 64-bit memory location to two packed 32-bit signed integer values and writes the converted values in another MMX register. If the result of the conversion is an inexact value, the value is truncated (rounded toward zero). The numeric range for source and destination operands is shown in Table 1-2 on page 89.

The PF2ID instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

CVTTPS2DQ

Mnemonic	Opcode	Description
PF2ID *mmx1*, *mmx2/mem64*	0F 0F /r 1D	Converts packed single-precision floating-point values in an MMX register or memory location to a doubleword integer value in the destination MMX register.

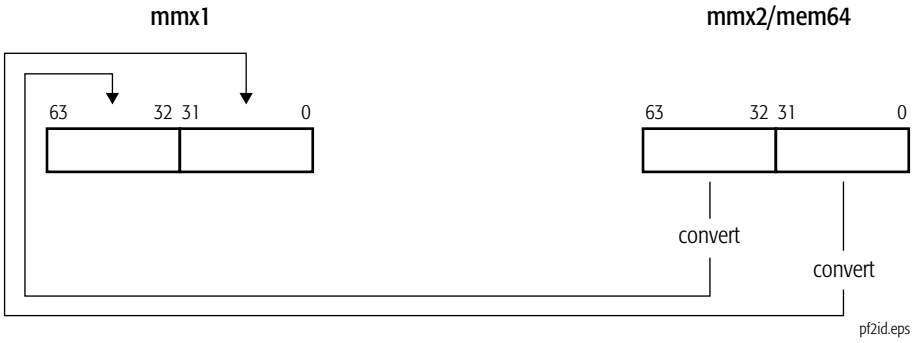


Table 1-2. Numeric Range for PF2ID Results

Source 2	Source 1 and Destination
0	0
Normal, abs(Source 2) < 1	0
Normal, -2^{31} < Source 2 <= –1	Round to zero (Source 2)
Normal, 1 <= Source 2 < 2^{31}	Round to zero (Source 2)
Normal, Source 2 >= 2^{31}	7FFF_FFFFh
Normal, Source 2 <= -2^{31}	8000_0000h
Unsupported	Undefined

Related Instructions

PF2IW, PI2FD, PI2FW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PF2IW Packed Floating-Point to Integer Word Conversion

Converts two packed single-precision floating-point values in an MMX register or a 64-bit memory location to two packed 16-bit signed integer values, sign-extended to 32 bits, and writes the converted values in another MMX register. If the result of the conversion is an inexact value, the value is truncated (rounded toward zero). The numeric range for source and destination operands is shown in Table 1-3 on page 91. Arguments outside the range representable by signed 16-bit integers are saturated to the largest and smallest 16-bit integer, depending on their sign.

The PF2IW instruction is an extension to the AMD 3DNow!™ instruction set. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

CVTTPS2DQ

Mnemonic	Opcode	Description
PF2IW *mmx1*, *mmx2/mem64*	0F 0F /r 1C	Converts packed single-precision floating-point values in an MMX register or memory location to word integer values in the destination MMX register.

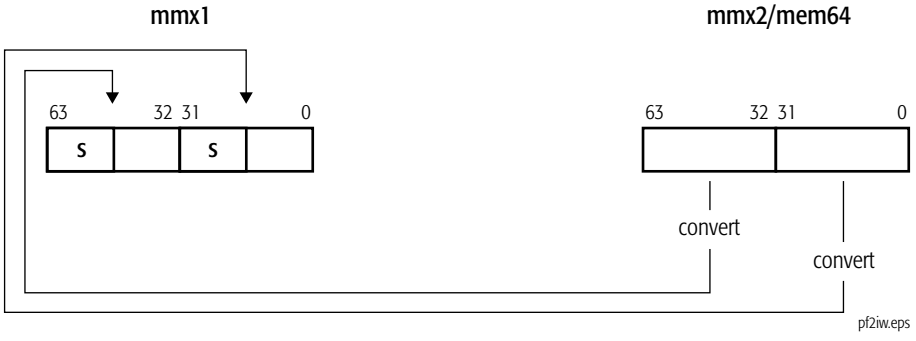


Table 1-3. Numeric Range for PF2IW Results

Source 2	Source 1 and Destination
0	0
Normal, abs(Source 2) < 1	0
Normal, -2^{15} < Source 2 <= –1	Round to zero (Source 2)
Normal, 1 <= Source 2 < 2^{15}	Round to zero (Source 2)
Normal, Source 2 >= 2^{15}	0000_7FFFh
Normal, Source 2 <= -2^{15}	FFFF_8000h
Unsupported	Undefined

Related Instructions

PF2ID, PI2FD, PI2FW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD extensions to 3DNow!™ are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNowExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFACC Packed Floating-Point Accumulate

Adds the two single-precision floating-point values in the first source operand and adds the two single-precision values in the second source operand and writes the two results to the low-order and high-order doubleword, respectively, of the destination (first source). The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location.

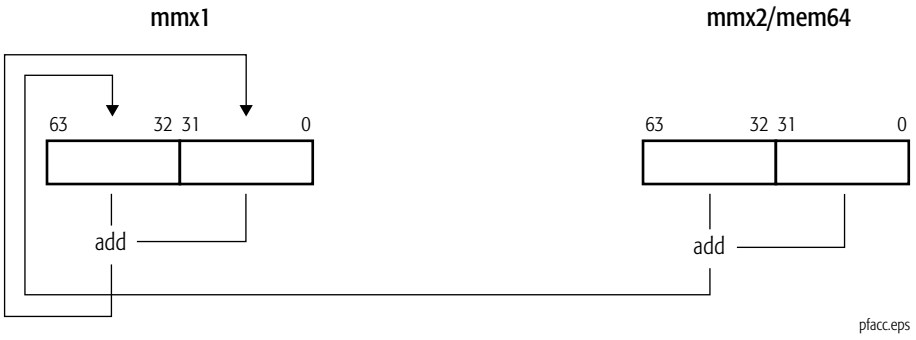
The PFACC instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

HADDPS

Mnemonic	Opcode	Description
PFACC *mmx1*, *mmx2/mem64*	0F 0F /r AE	Accumulates packed single-precision floating-point values in an MMX register or 64-bit memory location and another MMX register and writes each result in the destination MMX register.



The numeric range for operands is shown in Table 1-4 on page 93.

Table 1-4. Numeric Range for PFACC Results

Source Operand		High Operand[2]		
		0	Normal	Unsupported
Low Operand[1]	0	+/– 0[3]	High Operand	High Operand
	Normal	Low Operand	Normal, +/– 0[4]	Undefined
	Unsupported[5]	Low Operand	Undefined	Undefined

Note:

1. *Least-significant floating-point value in first or second source operand.*
2. *Most-significant floating-point value in first or second source operand.*
3. *The sign of the result is the logical AND of the signs of the low and high operands.*
4. *If the absolute value of the infinitely precise result is less than 2^{-126} (but not zero), the result is a zero with the sign of the operand (low or high) that is larger in magnitude. If the infinitely precise result is exactly zero, the result is zero with the sign of the low operand. If the absolute value of the infinitely precise result is greater than or equal to 2^{128}, the result is the largest normal number with the sign of the low operand.*
5. *"Unsupported" means that the exponent is all ones (1s).*

Related Instructions

PFADD, PFNACC, PFPNACC

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFADD Packed Floating-Point Add

Adds each packed single-precision floating-point value in the first source operand to the corresponding packed single-precision floating-point value in the second operand and writes the result of each addition in the corresponding doubleword of the destination (first source). The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location. The numeric range for operands is shown in Table 1-5 on page 95.

The PFADD instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

ADDPS

Mnemonic	Opcode	Description
PFADD *mmx1*, *mmx2/mem64*	0F 0F /r 9E	Adds two packed single-precision floating-point values in an MMX register or 64-bit memory location and another MMX register and writes each result in the destination MMX register.

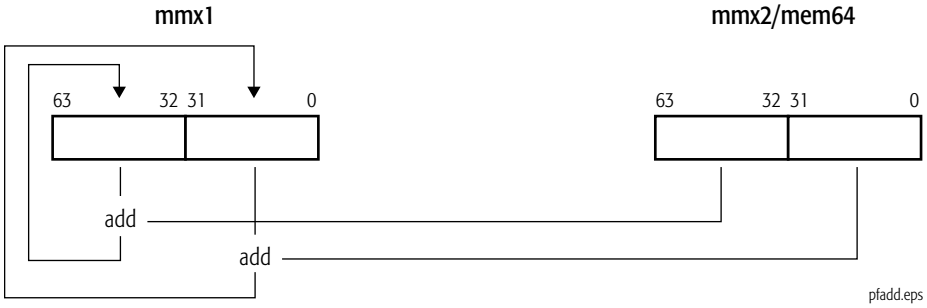


Table 1-5. Numeric Range for the PFADD Results

Source Operand		Most-Significant Doubleword		
		0	Normal	Unsupported
Source 1 and Destination	0	+/– 0[1]	Source 2	Source 2
	Normal	Source 1	Normal, +/– 0[2]	Undefined
	Unsupported[3]	Source 1	Undefined	Undefined

Note:

1. *The sign of the result is the logical AND of the signs of the source operands.*
2. *If the absolute value of the infinitely precise result is less than 2^{-126} (but not zero), the result is a zero with the sign of the source operand that is larger in magnitude. If the infinitely precise result is exactly zero, the result is zero with the sign of source 1. If the absolute value of the infinitely precise result is greater than or equal to 2^{128}, the result is the largest normal number with the sign of source 1.*
3. *"Unsupported" means that the exponent is all ones (1s).*

Related Instructions

PFACC, PFNACC, PFPNACC

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFCMPEQ Packed Floating-Point Compare Equal

Compares each of the two packed single-precision floating-point values in the first source operand with the corresponding packed single-precision floating-point value in the second source operand and writes the result of each comparison in the corresponding doubleword of the destination (first source). For each pair of floating-point values, if the values are equal, the result is all 1s. If the values are not equal, the result is all 0s. The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location. The numeric range for operands is shown in Table 1-6 on page 97.

The PFCMPEQ instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

CMPSS

Mnemonic	Opcode	Description
PFCMPEQ *mmx1*, *mmx2*/*mem64*	0F 0F /r B0	Compares two pairs of packed single-precision floating-point values in an MMX register and an MMX register or 64-bit memory location.

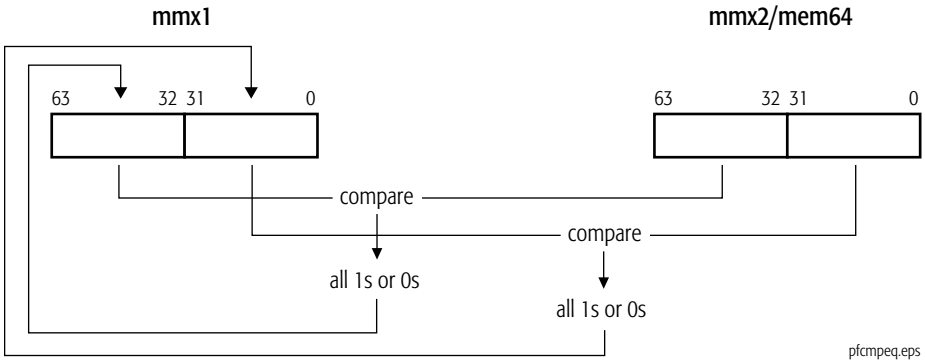


Table 1-6. Numeric Range for the PFCMPEQ Instruction

Operand	Value	Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	FFFF_FFFFh[1]	0000_0000h	0000_0000h
	Normal	0000_0000h	0000_0000h or FFFF_FFFFh[2]	0000_0000h
	Unsupported[3]	0000_0000h	0000_0000h	Undefined

Note:

1. *Positive zero is equal to negative zero.*
2. *The result is FFFF_FFFFh if source 1 and source 2 have identical signs, exponents, and mantissas. Otherwise, the result is 0000_0000h.*
3. *"Unsupported" means that the exponent is all ones (1s).*

Related Instructions

PFCMPGE, PFCMPGT

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFCMPGE Packed Floating-Point Compare Greater or Equal

Compares each of the two packed single-precision floating-point values in the first source operand with the corresponding packed single-precision floating-point value in the second source operand and writes the result of each comparison in the corresponding doubleword of the destination (first source). For each pair of floating-point values, if the value in the first source operand is greater than or equal to the value in the second source operand, the result is all 1s. If the value in the first source operand is less than the value in the second source operand, the result is all 0s. The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location. The numeric range for operands is shown in Table 1-7 on page 99.

The PFCMPGE instruction is a 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

CMPPS

Mnemonic	Opcode	Description
PFCMPGE *mmx1*, *mmx2/mem64*	0F 0F /r 90	Compares two pairs of packed single-precision floating-point values in an MMX register and an MMX register or 64-bit memory location.

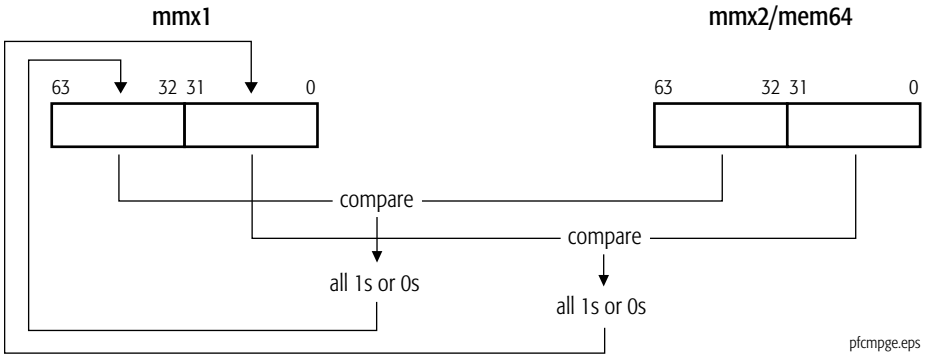


Table 1-7. Numeric Range for the PFCMPGE Instruction

Operand	Value	Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	FFFF_FFFFh[1]	0000_0000h, FFFF_FFFFh[2]	Undefined
	Normal	0000_0000h, FFFF_FFFFh[3]	0000_0000h, FFFF_FFFFh[4]	Undefined
	Unsupported[5]	Undefined	Undefined	Undefined

Note:

1. *Positive zero is equal to negative zero.*
2. *The result is FFFF_FFFFh, if source 2 is negative. Otherwise, the result is 0000_0000h.*
3. *The result is FFFF_FFFFh, if source 1 is positive. Otherwise, the result is 0000_0000h.*
4. *The result is FFFF_FFFFh, if source 1 is positive and source 2 is negative, or if they are both negative and source 1 is smaller than or equal in magnitude to source 2, or if source 1 and source 2 are both positive and source 1 is greater than or equal in magnitude to source 2. The result is 0000_0000h in all other cases.*
5. *"Unsupported" means that the exponent is all ones (1s).*

Related Instructions

PFCMPEQ, PFCMPGT

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.

Exception	Real	Virtual 8086	Protected	Cause of Exception
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFCMPGT Packed Floating-Point Compare Greater Than

Compares each of the two packed single-precision floating-point values in the first source operand with the corresponding packed single-precision floating-point value in the second source operand and writes the result of each comparison in the corresponding doubleword of the destination (first source). For each pair of floating-point values, if the value in the first source operand is greater than the value in the second source operand, the result is all 1s. If the value in the first source operand is less than or equal to the value in the second source operand, the result is all 0s. The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location. The numeric range for operands is shown in Table 1-8 on page 102.

The PFCMPGT instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

CMPPS

Mnemonic	Opcode	Description
PFCMPGT *mmx1*, *mmx2/mem64*	0F 0F /r A0	Compares two pairs of packed single-precision floating-point values in an MMX register and an MMX register or 64-bit memory location.

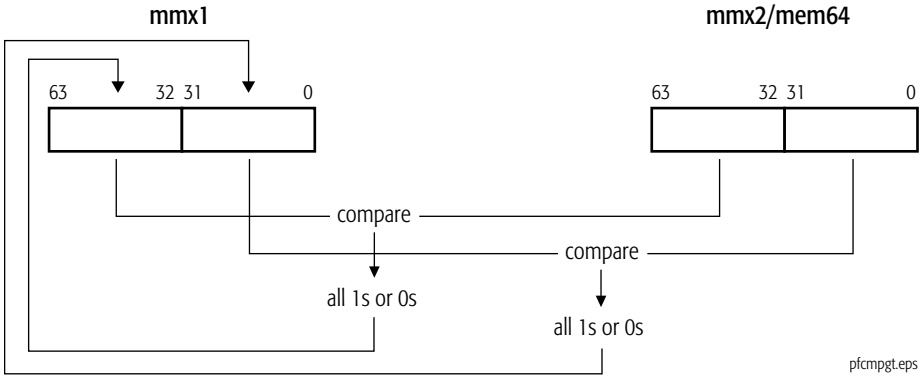


Table 1-8. Numeric Range for the PFCMPGT Instruction

Operand	Value	Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	0000_0000h	0000_0000h, FFFF_FFFFh[1]	Undefined
	Normal	0000_0000h, FFFF_FFFFh[2]	0000_0000h, FFFF_FFFFh[3]	Undefined
	Unsupported[4]	Undefined	Undefined	Undefined

Note:

1. *The result is FFFF_FFFFh, if source 2 is negative. Otherwise, the result is 0000_0000h.*
2. *The result is FFFF_FFFFh, if source 1 is positive. Otherwise, the result is 0000_0000h.*
3. *The result is FFFF_FFFFh, if source 1 is positive and source 2 is negative, or if they are both negative and source 1 is smaller in magnitude than source 2, or if source 1 and source 2 are positive and source 1 is greater in magnitude than source 2. The result is 0000_0000h in all other cases.*
4. *"Unsupported" means that the exponent is all ones (1s).*

Related Instructions

PFCMPEQ, PFCMPGE

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by ECPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFMAX Packed Single-Precision Floating-Point Maximum

Compares each of the two packed single-precision floating-point values in the first source operand with the corresponding packed single-precision floating-point value in the second source operand and writes the maximum of the two values for each comparison in the corresponding doubleword of the destination (first source). The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location.

Any operation with a zero and a negative number returns positive zero. An operation consisting of two zeros returns positive zero. If either source operand is an undefined value, the result is undefined. The numeric range for source and destination operands is shown in Table 1-9 on page 104.

The PFMAX instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

MAXPS

Mnemonic	Opcode	Description
PFMAX *mmx1*, *mmx2/mem64*	0F 0F /r A4	Compares two pairs of packed single-precision values in an MMX register and another MMX register or 64-bit memory location and writes the maximum value of each comparison in the destination MMX register.

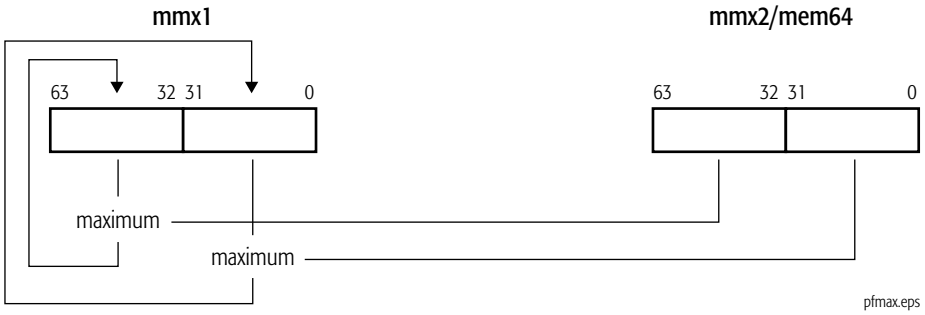


Table 1-9. Numeric Range for the PFMAX Instruction

Operand	Value	Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+0	Source 2, +0[1]	Undefined
	Normal	Source 1, +0[2]	Source 1/Source 2[3]	Undefined
	Unsupported[4]	Undefined	Undefined	Undefined

Note:

1. *The result is source 2, if source 2 is positive. Otherwise, the result is positive zero.*
2. *The result is source 1, if source 1 is positive. Otherwise, the result is positive zero.*
3. *The result is source 1, if source 1 is positive and source 2 is negative. The result is source 1, if both are positive and source 1 is greater in magnitude than source 2. The result is source 1, if both are negative and source 1 is lesser in magnitude than source 2. The result is source 2 in all other cases.*
4. *"Unsupported" means that the exponent is all ones (1s).*

Related Instructions

PFMIN

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFMIN Packed Single-Precision Floating-Point Minimum

Compares each of the two packed single-precision floating-point values in the first source operand with the corresponding packed single-precision floating-point value in the second source operand and writes the minimum of the two values for each comparison in the corresponding doubleword of the destination (first source). The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location.

Any operation with a zero and a positive number returns positive zero. An operation consisting of two zeros returns positive zero. If either source operand is an undefined value, the result is undefined. The numeric range for source and destination operands is shown in Table 1-10 on page 106.

The PFMIN instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

MINPS

Mnemonic	Opcode	Description
PFMIN *mmx1*, *mmx2/mem64*	0F 0F /r 94	Compares two pairs of packed single-precision values in an MMX register and another MMX register or 64-bit memory location and writes the minimum value of each comparison in the destination MMX register.

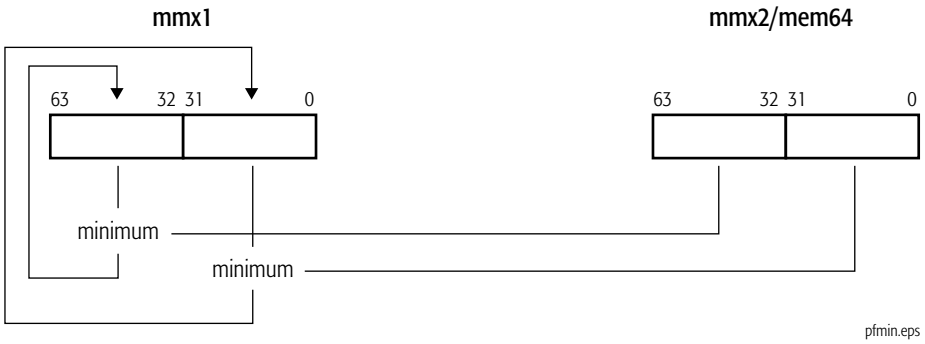


Table 1-10. Numeric Range for the PFMIN Instruction

Operand	Value	Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+0	Source 2, +0[1]	Undefined
	Normal	Source 1, +0[2]	Source 1/Source 2[3]	Undefined
	Unsupported[4]	Undefined	Undefined	Undefined

Note:

1. *The result is source 2, if source 2 is negative. Otherwise, the result is positive zero.*
2. *The result is source 1, if source 1 is negative. Otherwise, the result is positive zero.*
3. *The result is source 1, if source 1 is negative and source 2 is positive. The result is source 1, if both are negative and source 1 is greater in magnitude than source 2. The result is source 1, if both are positive and source 1 is lesser in magnitude than source 2. The result is source 2 in all other cases.*
4. *“Unsupported” means that the exponent is all ones (1s).*

Related Instructions

PFMAX

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFMUL Packed Floating-Point Multiply

Multiplies each of the two packed single-precision floating-point values in the first source operand by the corresponding packed single-precision floating-point value in the second source operand and writes the result of each multiplication in the corresponding doubleword of the destination (first source). The numeric range for source and destination operands is shown in Table 1-11 on page 108. The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location.

The PFMUL instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

MULPS

Mnemonic	Opcode	Description
PFMUL *mmx1*, *mmx2/mem64*	0F 0F /r B4	Multiplies packed single-precision floating-point values in an MMX register and another MMX register or 64-bit memory location and writes the result in the destination MMX register.

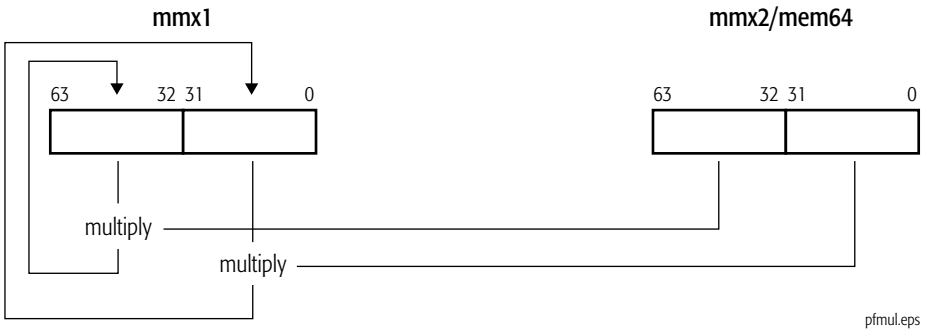


Table 1-11. Numeric Range for the PFMUL Instruction

Operand	Value	Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+/– 0[1]	+/– 0[1]	+/– 0[1]
	Normal	+/– 0[1]	Normal, +/– 0[2]	Undefined
	Unsupported[3]	+/– 0[1]	Undefined	Undefined

Note:

1. *The sign of the result is the exclusive-OR of the signs of the source operands.*
2. *If the absolute value of the result is less than 2^{-126}, the result is zero with the sign being the exclusive-OR of the signs of the source operands. If the absolute value of the product is greater than or equal to 2^{128}, the result is the largest normal number with the sign being the exclusive-OR of the signs of the source operands.*
3. *"Unsupported" means that the exponent is all ones (1s).*

Related Instructions

None

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFNACC Packed Floating-Point Negative Accumulate

Subtracts the first source operand's high-order single-precision floating-point value from its low-order single-precision floating-point value, subtracts the second source operand's high-order single-precision floating-point value from its low-order single-precision floating-point value, and writes each result to the low-order or high-order doubleword, respectively, of the destination (first source). The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location.

The numeric range for operands is shown in Table 1-12 on page 110.

The PFNACC instruction is an extension to the AMD 3DNow!™ instruction set. The presence of this instruction set is indicated by CPUID Fn8000_0001_EDX[3DNowExt] =1. See "CPUID" in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, "Recommended Substitutions for 3DNow!™ Instructions" on page 333.

Recommended Instruction Substitution

HSUBPS

Mnemonic	Opcode	Description
PFNACC *mmx1*, *mmx2/mem64*	0F 0F /r 8A	Subtracts the packed single-precision floating-point values in an MMX register or 64-bit memory location and another MMX register and writes each value in the destination MMX register.

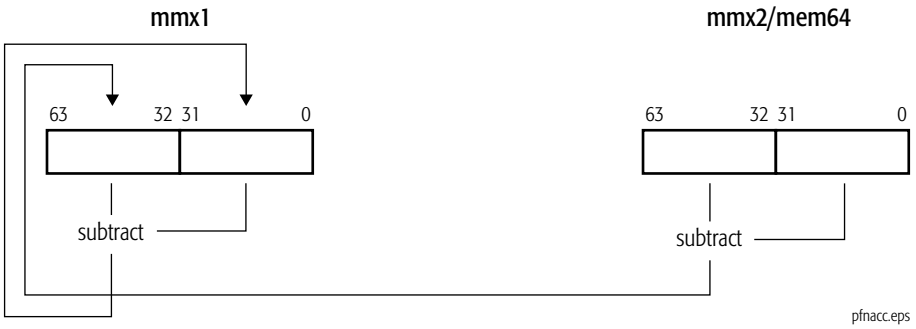


Table 1-12. Numeric Range of PFNACC Results

Source Operand		High Operand[2]		
		0	Normal	Unsupported
Low Operand[1]	0	+/– 0[3]	- High Operand	- High Operand
	Normal	Low Operand	Normal, +/– 0[4]	Undefined
	Unsupported[5]	Low Operand	Undefined	Undefined

Note:

1. *Least-significant floating-point value in first or second source operand.*
2. *Most-significant floating-point value in first or second source operand.*
3. *The sign is the logical AND of the sign of the low operand and the inverse of the sign of the high operand.*
4. *If the absolute value of the infinitely precise result is less than 2^{-126} (but not zero), the result is a zero. If the low operand is larger in magnitude than the high operand, the sign of this zero is the same as the sign of the low operand, else it is the inverse of the sign of the high operand. If the infinitely precise result is exactly zero, the result is zero with the sign of the low operand. If the absolute value of the infinitely precise result is greater than or equal to 2^{128}, the result is the largest normal number with the sign of the low operand.*
5. *"Unsupported" means that the exponent is all ones (1s).*

Related Instructions

PFSUB, PFACC, PFPNACC

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD extensions to 3DNow!™ are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNowExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.

Exception	Real	Virtual 8086	Protected	Cause of Exception
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFPNACC Packed Floating-Point Positive-Negative Accumulate

Subtracts the first source operand's high-order single-precision floating-point value from its low-order single-precision floating-point value, adds the two single-precision values in the second source operand, and writes each result to the low-order or high-order doubleword, respectively, of the destination (first source). The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location.

The numeric range for operands is shown in Table 1-13 (for the low result) and Table 1-14 (for the high result), both on page 113.

The PFPNACC instruction is an extension to the AMD 3DNow!™ instruction set. The presence of this instruction set is indicated by CPUID feature bits. See "CPUID" in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, "Recommended Substitutions for 3DNow!™ Instructions" on page 333.

Recommended Instruction Substitution

ADDSUBPS

Mnemonic	Opcode	Description
PFPNACC *mmx1*, *mmx2/mem64*	0F 0F /r 8E	Subtracts the packed single-precision floating-point values in an MMX register, adds the packed single-precision floating-point values in another MMX register or 64-bit memory location, and writes each value in the destination MMX register.

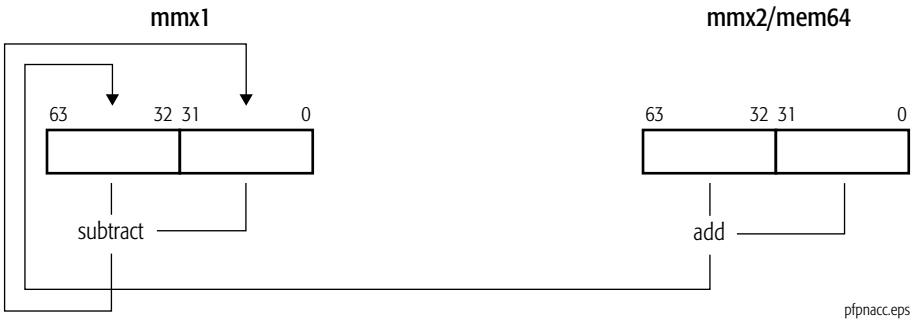


Table 1-13. Numeric Range of PFPNACC Result (Low Result)

Source Operand		High Operand[2]		
		0	**Normal**	**Unsupported**
Low Operand[1]	**0**	+/– 0[3]	- High Operand	- High Operand
	Normal	Low Operand	Normal, +/– 0[4]	Undefined
	Unsupported[5]	Low Operand	Undefined	Undefined

Note:

1. *Least-significant floating-point value in first or second source operand.*
2. *Most-significant floating-point value in first or second source operand.*
3. *The sign is the logical AND of the sign of the low operand and the inverse of the sign of the high operand.*
4. *If the absolute value of the infinitely precise result is less than 2^{-126} (but not zero), the result is a zero. If the low operand is larger in magnitude than the high operand, the sign of this zero is the same as the sign of the low operand, else it is the inverse of the sign of the high operand. If the infinitely precise result is exactly zero, the result is zero with the sign of the low operand. If the absolute value of the infinitely precise result is greater than or equal to 2^{128}, the result is the largest normal number with the sign of the low operand.*
5. *"Unsupported" means that the exponent is all ones (1s).*

Table 1-14. Numeric Range of PFPNACC Result (High Result)

Source Operand		High Operand[2]		
		0	**Normal**	**Unsupported**
Low Operand[1]	**0**	+/– 0[3]	High Operand	High Operand
	Normal	Low Operand	Normal, +/– 0[4]	Undefined
	Unsupported[5]	Low Operand	Undefined	Undefined

Note:

1. *Least-significant floating-point value in first or second source operand.*
2. *Most-significant floating-point value in first or second source operand.*
3. *The sign is the logical AND of the signs of the low and high operands.*
4. *If the absolute value of the infinitely precise result is less than 2^{-126} (but not zero), the result is zero with the sign of the operand (low or high) that is larger in magnitude. If the infinitely precise result is exactly zero, the result is zero with the sign of the low operand. If the absolute value of the infinitely precise result is greater than or equal to 2^{128}, the result is the largest normal number with the sign of the low operand.*
5. *"Unsupported" means that the exponent is all ones (1s).*

Related Instructions

PFADD, PFSUB, PFACC, PFNACC

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD extensions to 3DNow!™ are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNowExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFRCP Floating-Point Reciprocal Approximation

Computes the approximate reciprocal of the single-precision floating-point value in the low-order 32 bits of an MMX register or 64-bit memory location and writes the result in both doublewords of another MMX register. The result is accurate to 14 bits.

The PFRCP result can be forwarded to the Newton-Raphson iteration step 1 (PFRCPIT1) and Newton-Raphson iteration step 2 (PFRCPIT2) instructions to increase the accuracy of the reciprocal. The first stage of this refinement in accuracy (PFRCPIT1) requires that the input and output of the previously executed PFRCP instruction be used as input to the PFRCPIT1 instruction.

The estimate contains the correct round-to-nearest value for approximately 99% of all arguments. The remaining arguments differ from the correct round-to-nearest value for the reciprocal by 1 unit-in-the-last-place (ulp). For details, see the data sheet or other software-optimization documentation relating to particular hardware implementations.

PFRCP(x) returns 0 for x >= 2^{-126}. The numeric range for operands is shown in Table 1-15 on page 116.

The PFRCP instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

RCPSS

Mnemonic	Opcode	Description
PFRCP *mmx1*, *mmx2/mem64*	0F 0F /r 96	Computes approximate reciprocal of single-precision floating-point value in an MMX register or 64-bit memory location and writes the result in both doublewords of the destination MMX register.

Table 1-15. Numeric Range for the PFRCP Result

Operand		Source 1 and Destination
Source 2	0	+/– Maximum Normal[1]
	Normal	Normal, +/– 0[2]
	Unsupported[3]	Undefined

Note:

1. *The result has the same sign as the source operand.*
2. *If the absolute value of the result is less than 2^{-126}, the result is zero with the sign being the sign of the source operand. Otherwise, the result is a normal with the sign being the same sign as the source operand.*
3. *"Unsupported" means that the exponent is all ones (1s).*

Examples

The general Newton-Raphson recurrence for the reciprocal 1/b is:

$$Z_{i+1} \leftarrow Z_i \cdot (2 - b \cdot Z_i)$$

The following code sequence shows the computation of a/b:

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

The 24-bit final reciprocal value is X_2. The quotient is formed in the last step by multiplying the reciprocal by the dividend a.

Related Instructions

PFRCPIT1, PFRCPIT2

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFRCPIT1 Packed Floating-Point Reciprocal Iteration 1

Performs the first step in the Newton-Raphson iteration to refine the reciprocal approximation produced by the PFRCP instruction. The first source/destination operand is an MMX register containing the results of two previous PFRCP instructions, and the second source operand is another MMX register or 64-bit memory location containing the source operands from the same PFRCP instructions.

This instruction is only defined for those combinations of operands such that the first source operand (mmx1) is the approximate reciprocal of the second source operand (mmx2/mem64), and thus the range of the product, mmx1 * mmx2/mem64, is (0.5, 2). The initial approximation of an operand is accurate to about 12 bits, and the length of the operand itself is 24 bits, so the product of these two operands is greater than 24 bits. PFRCPIT1 applies the one's complement of the product and rounds the result to 32 bits. It then compresses the result to fit into 24 bits by removing the 8 redundant most-significant bits after the hidden integer bit.

The estimate contains the correct round-to-nearest value for approximately 99% of all arguments. The remaining arguments differ from the correct round-to-nearest value for the reciprocal by 1 unit-in-the-last-place (ulp). For details, see the data sheet or other software-optimization documentation relating to particular hardware implementations.

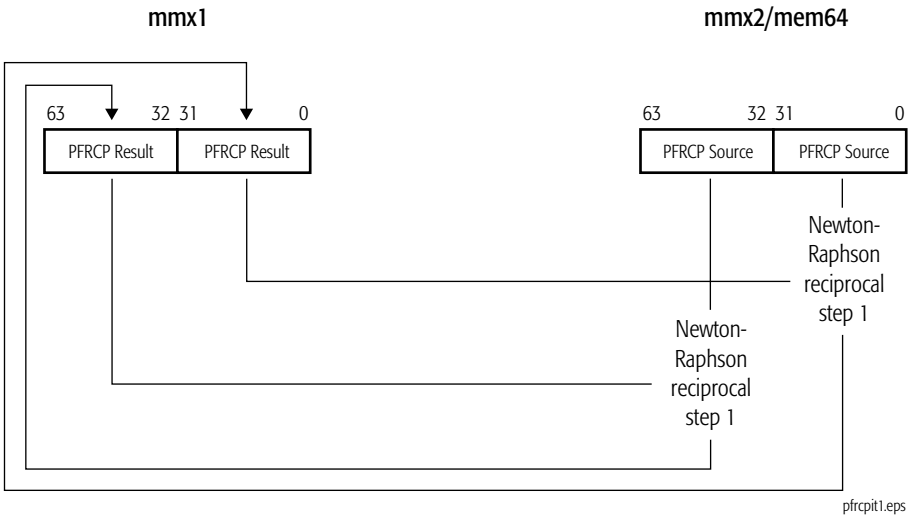
The PFRCPIT1 instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

PFRCP

Mnemonic	Opcode	Description
PFRCPIT1 *mmx1*, *mmx2/mem64*	0F 0F /r A6	Refine approximate reciprocal of result from previous PFRCP instruction.



Operation

```
mmx1[31:0] = Compress (2 - mmx1[31:0] * (mmx2/mem64[31:0]) - 2^31);
mmx1[63:32] = Compress (2 - mmx1[63:32] * (mmx2/mem64[63:32]) - 2^31);
```

where:

“Compress” means discard the 8 redundant most-significant bits after the hidden integer bit.

Examples

The general Newton-Raphson recurrence for the reciprocal 1/b is:

$Z_{i+1} \leftarrow Z_i \cdot (2 - b \cdot Z_i)$

The following code sequence computes a 24-bit approximation to a/b with one Newton-Raphson iteration:

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

a/b is formed in the last step by multiplying the reciprocal approximation by a.

Related Instructions

PFRCP, PFRCPIT2

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFRCPIT2 Packed Floating-Point Reciprocal or Reciprocal Square Root Iteration 2

Performs the second and final step in the Newton-Raphson iteration to refine the reciprocal approximation produced by the PFRCP instruction or the reciprocal square-root approximation produced by the PFSQRT instruction. PFRCPIT2 takes two paired elements in each source operand. These paired elements are the results of a PFRCP and PFRCPIT1 instruction sequence or of a PFRSQRT and PFRSQIT1 instruction sequence. The first source/destination operand is an MMX register that contains the PFRCPIT1 or PFRSQIT1 results and the second source operand is another MMX register or 64-bit memory location that contains the PFRCP or PFRSQRT results.

The PFRCPIT2 instruction expands the compressed PFRCPIT1 or PFRSQIT1 results from 24 to 32 bits and multiplies them by their respective source operands. An optimal correction factor is added to the product, which is then rounded to 24 bits.

The estimate contains the correct round-to-nearest value for approximately 99% of all arguments. The remaining arguments differ from the correct round-to-nearest value for the reciprocal by 1 unit-in-the-last-place (ulp). For details, see the data sheet or other software-optimization documentation relating to particular hardware implementations.

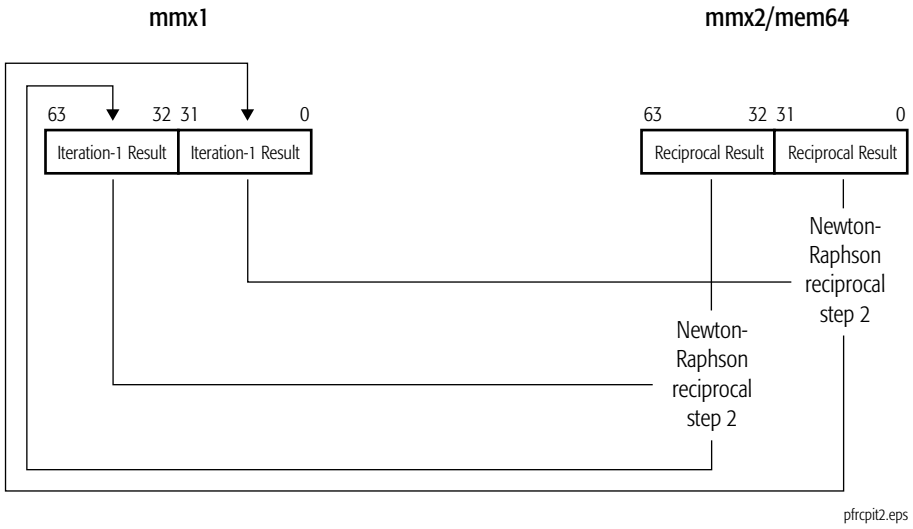
The PFRCPIT2 instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

PFRCP

Mnemonic	Opcode	Description
PFRCPIT2 *mmx1*, *mmx2/mem64*	0F 0F /r B6	Refines approximate reciprocal result from previous PFRCP and PFRCPIT1 instructions or from previous PFRSQRT and PFRSQIT1 instructions.



Operation

```
mmx1[31:0] = Expand(mmx1[31:0]) * mmx2/mem64[31:0];
mmx1[63:32] = Expand(mmx1[63:32]) * mmx2/mem64[63:32];
```

where:

“Expand” means convert a 24-bit significand to a 32-bit significand according to the following rule:

```
temp[31:0] = {1'b1, 8{mmx1[22]}, mmx1[22:0]};
```

Examples

The general Newton-Raphson recurrence for the reciprocal 1/b is:

$Z_{i+1} \leftarrow Z_i \cdot (2 - b \cdot Z_i)$

The following code sequence computes a 24-bit approximation to a/b with one Newton-Raphson iteration:

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

a/b is formed in the last step by multiplying the reciprocal approximation by a.

Related Instructions

PFRCP, PFRCPIT1, PFRSQRT, PFRSQIT1

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFRSQIT1 Packed Floating-Point Reciprocal Square Root Iteration 1

Performs the first step in the Newton-Raphson iteration to refine the reciprocal square-root approximation produced by the PFSQRT instruction. The first source/destination operand is an MMX register containing the result from a previous PFRSQRT instruction, and the second source operand is another MMX register or 64-bit memory location containing the source operand from the same PFRSQRT instruction.

This instruction is only defined for those combinations of operands such that the first source operand (mmx1) is the approximate reciprocal of the second source operand (mmx2/mem64), and thus the range of the product, mmx1 * mmx2/mem64, is (0.5, 2). The length of both operands is 24 bits, so the product of these two operands is greater than 24 bits. The product is normalized and then rounded to 32 bits. The one's complement of the result is applied, a 1 is added as the most-significant bit, and the result re-normalized. The result is then compressed to fit into 24 bits by removing 8 redundant most-significant bits after the hidden integer bit, and the exponent is reduced by 1 to account for the division by 2.

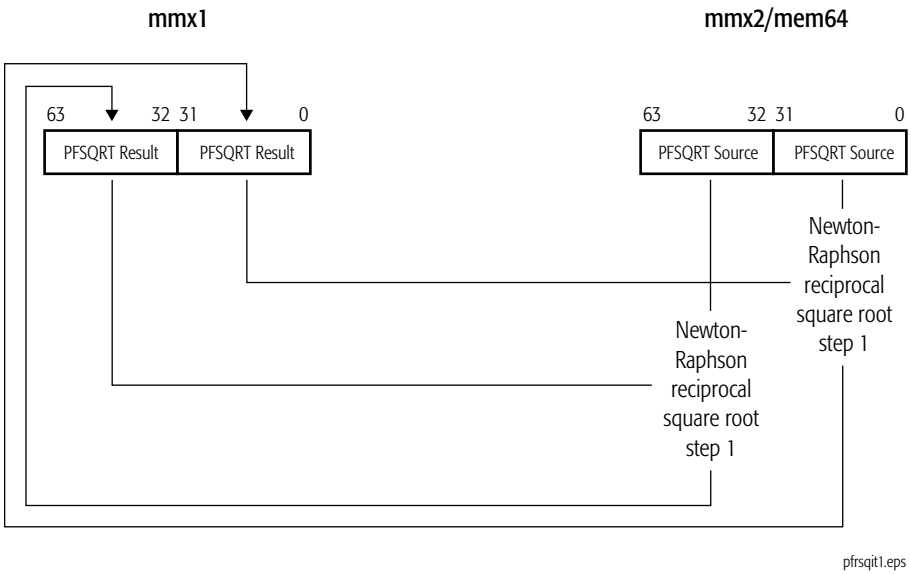
The PFRSQIT1 instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

PFRSQRT

Mnemonic	Opcode	Description
PFRSQIT1 *mmx1*, *mmx2/mem64*	0F 0F /r A7	Refines reciprocal square root approximation of previous PFRSQRT instruction.



Operation

```
mmx1[31:0] = Compress ((3 - mmx1[31:0] * (mmx2/mem64[31:0]) - 2^31)/2);
mmx1[63:32] = Compress ((3 - mmx1[63:32] * (mmx2/mem64[63:32]) - 2^31)/2);
```

where:

“Compress” means discard the 8 redundant most-significant bits after the hidden integer bit.

Examples

The following code sequence shows how the PFRSQRT and PFMUL instructions can be used to compute a = 1/sqrt (b):

```
X0 = PFRSQRT(b)
X1 = PFMUL(X0,X0)
X2 = PFRSQIT1(b,X1)
a = PFRCPIT2(X2,X0)
```

Related Instructions

PFRCPIT2, PFRSQRT

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFRSQRT Packed Floating-Point Reciprocal Square Root Approximation

Computes the approximate reciprocal square root of the single-precision floating-point value in the low-order 32 bits of an MMX register or 64-bit memory location and writes the result in each doubleword of another MMX register. The source operand is single-precision with a 24-bit significand, and the result is accurate to 15 bits. Negative operands are treated as positive operands for purposes of reciprocal square-root computation, with the sign of the result the same as the sign of the source operand.

This instruction can be used together with the PFRSQIT1 and PFRCPIT2 instructions to increase accuracy. The first stage of this refinement in accuracy (PFRSQIT1) requires that the input and output of the previously executed PFRSQRT instruction be used as input to the PFRSQIT1 instruction.

The estimate contains the correct round-to-nearest value for approximately 99% of all arguments. The remaining arguments differ from the correct round-to-nearest value for the reciprocal by 1 unit-in-the-last-place (ulp). For details, see the data sheet or other software-optimization documentation relating to particular hardware implementations.

The PFRSQRT instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

The numeric range for operands is shown in Table 1-16 on page 128.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

RSQRTSS

Mnemonic	Opcode	Description
PFRSQRT *mmx1*, *mmx2/mem64*	0F 0F /r 97	Computes approximate reciprocal square root of a packed single-precision floating-point value.

Table 1-16. Numeric Range for the PFRCP Result

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

Note:
1. *The result has the same sign as the source operand.*
2. *"Unsupported" means that the exponent is all ones (1s).*

Examples

The following code sequence shows how the PFRSQRT and PFMUL instructions can be used to compute a = 1/sqrt (b):

```
X0 = PFRSQRT(b)
X1 = PFMUL(X0,X0)
X2 = PFRSQIT1(b,X1)
a = PFRCPIT2(X2,X0)
```

Related Instructions

PFRCPIT2, PFRSQIT1

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFSUB Packed Floating-Point Subtract

Subtracts each packed single-precision floating-point value in the second source operand from the corresponding packed single-precision floating-point value in the first source operand and writes the result of each subtraction in the corresponding doubleword of the destination (first source). The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location. The numeric range for operands is shown in Table 1-17 on page 131.

The PFSUB instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

SUBPS

Mnemonic	Opcode	Description
PFSUB *mmx1*, *mmx2/mem64*	0F 0F /r 9A	Subtracts packed single-precision floating-point values in an MMX register or 64-bit memory location from packed single-precision floating-point values in another MMX register and writes the result in the destination MMX register.

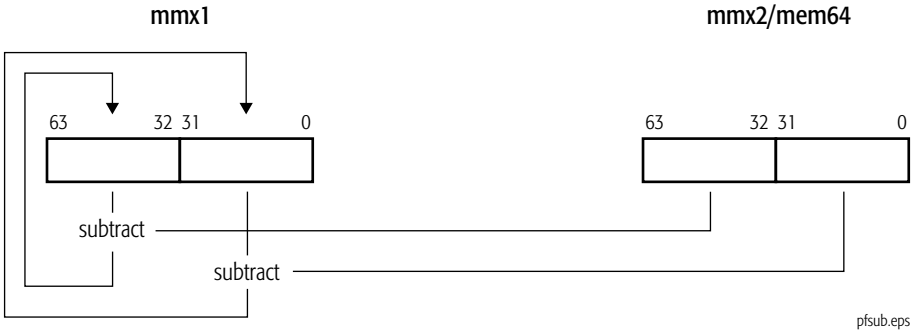


Table 1-17. Numeric Range for the PFSUB Results

Source Operand		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+/– 0[1]	- Source 2	- Source 2
	Normal	Source 1	Normal, +/– 0[2]	Undefined
	Unsupported[3]	Source 1	Undefined	Undefined

Note:

1. *The sign of the result is the logical AND of the sign of source 1 and the inverse of the sign of source 2.*
2. *If the absolute value of the infinitely precise result is less than 2^{-126} (but not zero), the result is a zero. If the source operand that is larger in magnitude is source 1, the sign of this zero is the same as the sign of source 1, else it is the inverse of the sign of source 2. If the infinitely precise result is exactly zero, the result is zero with the sign of source 1. If the absolute value of the infinitely precise result is greater than or equal to 2^{128}, the result is the largest normal number with the sign of source 1.*
3. *"Unsupported" means that the exponent is all ones (1s).*

Related Instructions

PFSUBR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PFSUBR Packed Floating-Point Subtract Reverse

Subtracts each packed single-precision floating-point value in the first source operand from the corresponding packed single-precision floating-point value in the second source operand and writes the result of each subtraction in the corresponding dword of the destination (first source). The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location. The numeric range for operands is shown in Table 1-18 on page 133.

The PFSUBR instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

SUBPS

Mnemonic	Opcode	Description
PFSUBR *mmx1*, *mmx2/mem64*	0F 0F /r AA	Subtracts packed single-precision floating-point values in an MMX register from packed single-precision floating-point values in another MMX register or 64-bit memory location and writes the result in the destination MMX register.

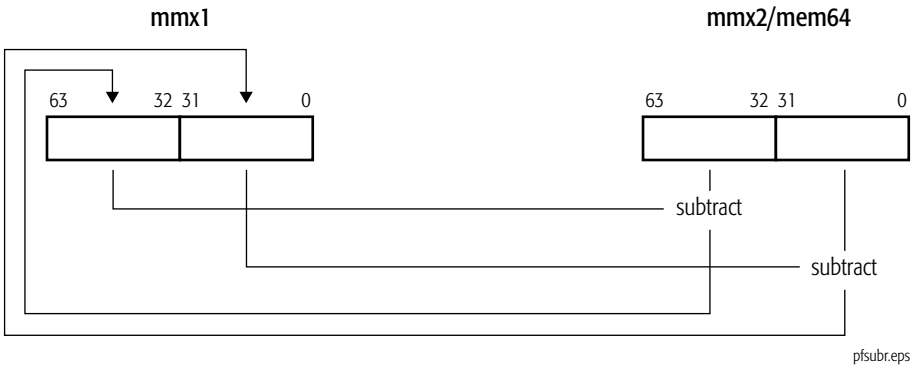


Table 1-18. Numeric Range for the PFSUBR Results

Source Operand		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+/– 0[1]	Source 2	Source 2
	Normal	- Source 1	Normal, +/– 0[2]	Undefined
	Unsupported[3]	- Source 1	Undefined	Undefined

Note:

1. *The sign is the logical AND of the sign of source 2 and the inverse of the sign of source 1.*
2. *If the absolute value of the infinitely precise result is less than 2^{-126} (but not zero), the result is a zero. If the source operand that is larger in magnitude is source 2, the sign of this zero is the same as the sign of source 2, else it is the inverse of the sign of source 1. If the infinitely precise result is exactly zero, the result is zero with the sign of source 2. If the absolute value of the infinitely precise result is greater than or equal to 2^{128}, the result is the largest normal number with the sign of source 2.*
3. *"Unsupported" means that the exponent is all ones (1s).*

Related Instructions

PFSUB

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by ECPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PI2FD Packed Integer to Floating-Point Doubleword Conversion

Converts two packed 32-bit signed integer values in an MMX register or a 64-bit memory location to two packed single-precision floating-point values and writes the converted values in another MMX register. If the result of the conversion is an inexact value, the value is truncated (rounded toward zero).

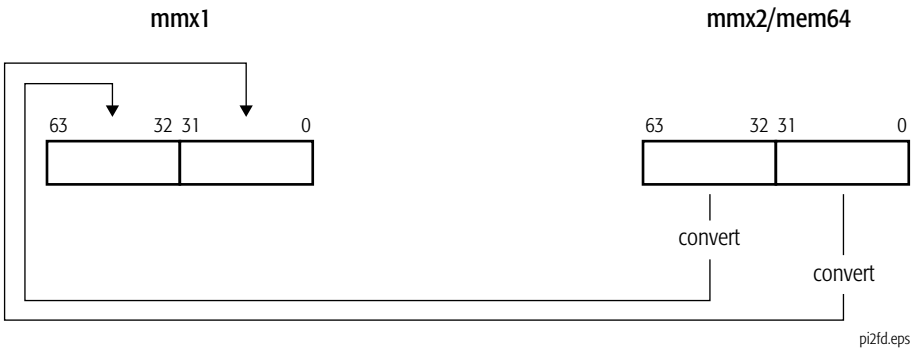
The PI2FD instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

AMD no longer recommends the use of 3DNow! instructions, which have been superceded by their more efficient 128-bit media counterparts. For a complete list of recommended instruction substitutions, see Appendix A, “Recommended Substitutions for 3DNow!™ Instructions” on page 333.

Recommended Instruction Substitution

CVTDQ2PS

Mnemonic	Opcode	Description
PI2FD *mmx1*, *mmx2/mem64*	0F 0F /r 0D	Converts packed doubleword integers in an MMX register or 64-bit memory location to single-precision floating-point values in the destination MMX register. Inexact results are truncated.



Related Instructions

PF2ID, PF2IW, PI2FW

rFLAGS Affected

None

Exceptions

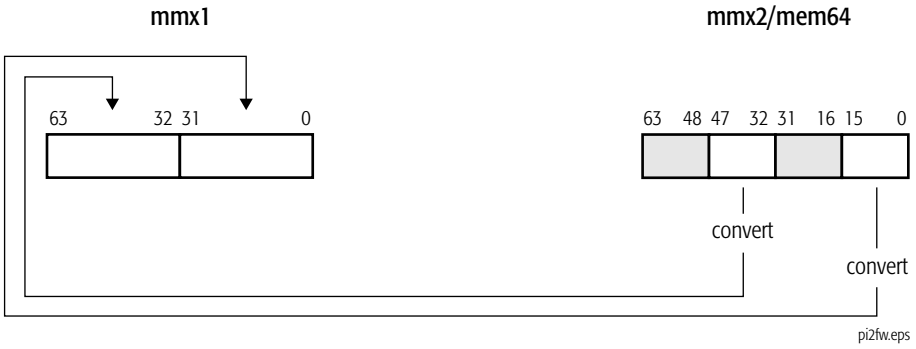
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PI2FW Packed Integer to Floating-Point Word Conversion

Converts two packed 16-bit signed integer values in an MMX register or a 64-bit memory location to two packed single-precision floating-point values and writes the converted values in another MMX register.

The PI2FW instruction is an extension to the AMD 3DNow!™ instruction set. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PI2FW *mmx1*, *mmx2/mem64*	0F 0F /r 0C	Converts packed 16-bit integers in an MMX register or 64-bit memory location to packed single-precision floating-point values in the destination MMX register.



Related Instructions

PF2ID, PF2IW, PI2FD

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD extensions to 3DNow!™ are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNowExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PINSRW Packed Insert Word

Inserts a 16-bit value from the low-order word of a 32-bit general purpose register or a 16-bit memory location into an MMX register. The location in the destination register is selected by the immediate byte operand, a shown in Table 1-19. The other words in the destination register operand are not modified.

The PINSRW instruction is an AMD extension to MMX™ instruction set and is an SSE1 instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PINSRW *mmx, reg32/mem16, imm8*	0F C4 */r ib*	Inserts a 16-bit value from a general-purpose register or memory location into an MMX register.

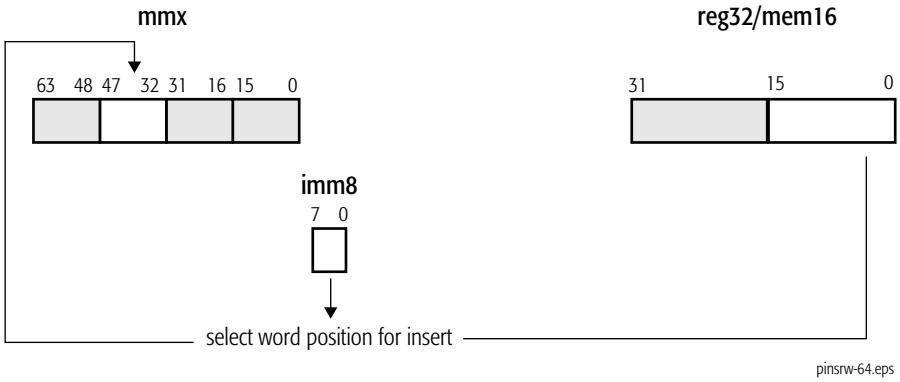


Table 1-19. Immediate-Byte Operand Encoding for 64-Bit PINSRW

Immediate-Byte Bit Field	Value of Bit Field	Destination Bits Filled
1–0	0	15–0
	1	31–16
	2	47–32
	3	63–48

Related Instructions

PEXTRW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0 and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

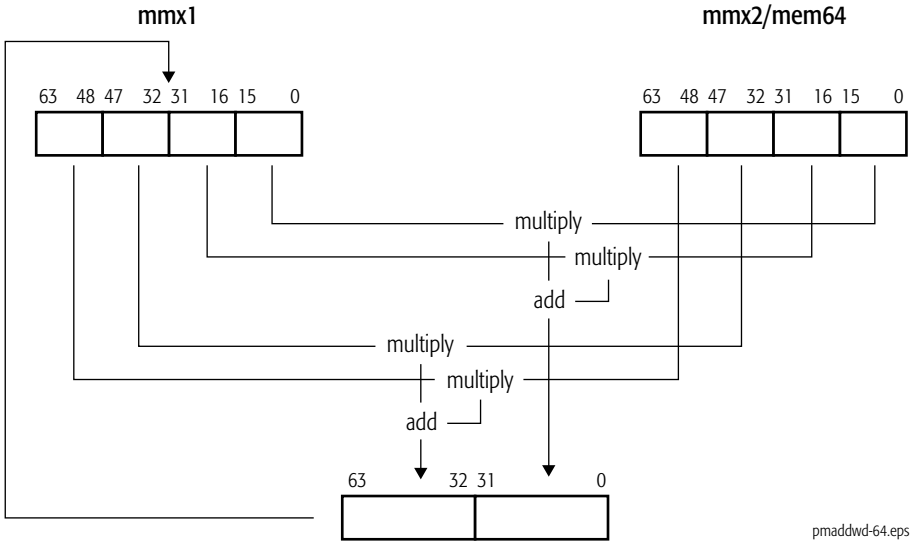
PMADDWD Packed Multiply Words and Add Doublewords

Multiplies each packed 16-bit signed value in the first source operand by the corresponding packed 16-bit signed value in the second source operand, adds the adjacent intermediate 32-bit results of each multiplication (for example, the multiplication results for the adjacent bit fields 63–48 and 47–32, and 31–16 and 15–0), and writes the 32-bit result of each addition in the corresponding doubleword of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

If all four of the 16-bit source operands used to produce a 32-bit multiply-add result have the value 8000h, the 32-bit result is 8000_0000h, which is not the correct 32-bit signed result.

The PMADDWD instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PMADDWD *mmx1*, *mmx2/mem64*	0F F5 */r*	Multiplies four packed 16-bit signed values in an MMX register and another MMX register or 64-bit memory location, adds intermediate results, and writes the result in the destination MMX register.



Related Instructions

PMULHUW, PMULHW, PMULLW, PMULUDQ

rFLAGS Affected

None

Exceptions

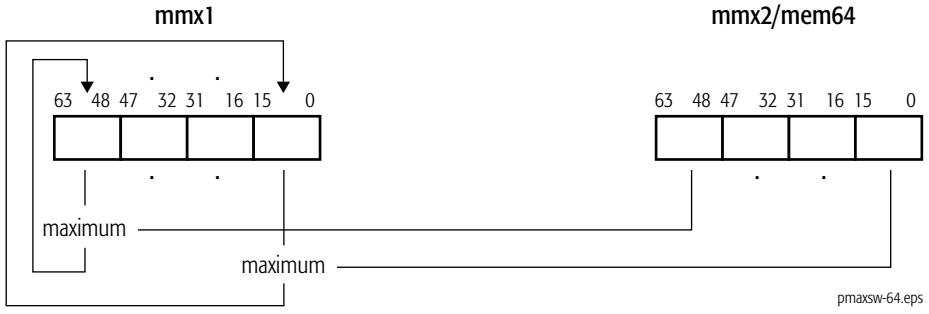
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PMAXSW Packed Maximum Signed Words

Compares each of the packed 16-bit signed integer values in the first source operand with the corresponding packed 16-bit signed integer value in the second source operand and writes the maximum of the two values for each comparison in the corresponding word of the destination (first source). The first source/destination and second source operands are an MMX register and an MMX register or 64-bit memory location.

The PMAXSW instruction is an AMD extension to MMX™ instruction set and is an SSE1 instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PMAXSW *mmx1*, *mmx2/mem64*	0F EE */r*	Compares packed signed 16-bit integer values in an MMX register and another MMX register or 64-bit memory location and writes the maximum value of each compare in destination MMX register.



Related Instructions

PMAXUB, PMINSW, PMINUB

rFLAGS Affected

None

Exceptions

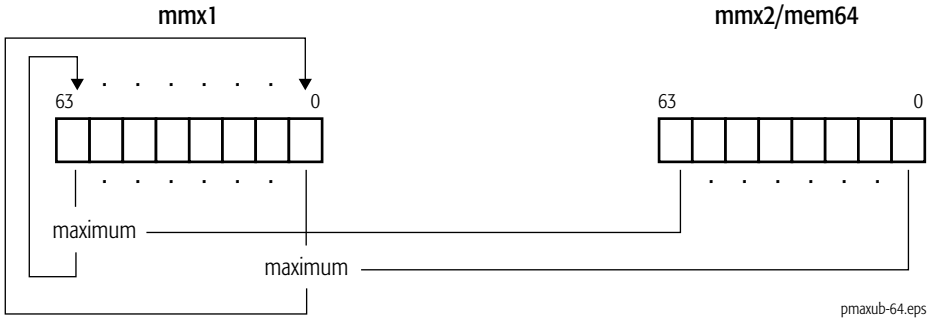
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0 and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PMAXUB Packed Maximum Unsigned Bytes

Compares each of the packed 8-bit unsigned integer values in the first source operand with the corresponding packed 8-bit unsigned integer value in the second source operand and writes the maximum of the two values for each comparison in the corresponding byte of the destination (first source). The first source/destination and second source operands are an MMX register and an MMX register or 64-bit memory location.

The PMAXUB instruction is an AMD extension to MMX™ instruction set and is an SSE1 instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PMAXUB *mmx1*, *mmx2/mem64*	0F DE */r*	Compares packed unsigned 8-bit integer values in an MMX register and another MMX register or 64-bit memory location and writes the maximum value of each compare in the destination MMX register.



Related Instructions

PMAXSW, PMINSW, PMINUB

rFLAGS Affected

None

Exceptions

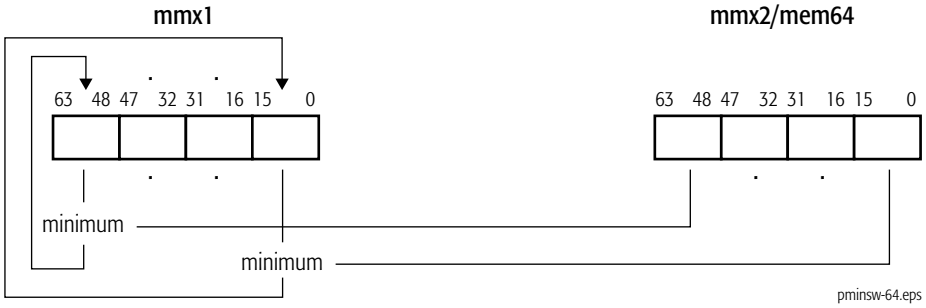
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0 and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PMINSW Packed Minimum Signed Words

Compares each of the packed 16-bit signed integer values in the first source operand with the corresponding packed 16-bit signed integer value in the second source operand and writes the minimum of the two values for each comparison in the corresponding word of the destination (first source). The first source/destination and second source operands are an MMX register and an MMX register or 64-bit memory location.

The PMINSW instruction is an AMD extension to MMX™ instruction set and is an SSE1 instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PMINSW *mmx1*, *mmx2*/*mem64*	0F EA */r*	Compares packed signed 16-bit integer values in an MMX register and another MMX register or 64-bit memory location and writes the minimum value of each compare in the destination MMX register.



Related Instructions

PMAXSW, PMAXUB, PMINUB

rFLAGS Affected

None

Exceptions

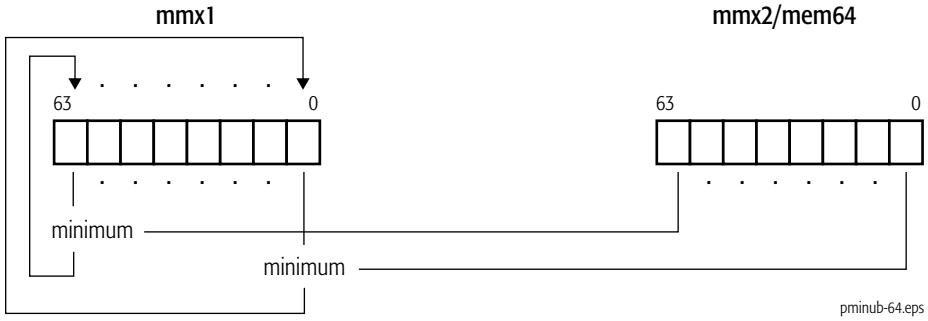
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0 and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PMINUB Packed Minimum Unsigned Bytes

Compares each of the packed 8-bit unsigned integer values in the first source operand with the corresponding packed 8-bit unsigned integer value in the second source operand and writes the minimum of the two values for each comparison in the corresponding byte of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PMINUB instruction is an AMD extension to MMX™ instruction set and is an SSE1 instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PMINUB *mmx1*, *mmx2*/*mem64*	0F DA */r*	Compares packed unsigned 8-bit integer values in an MMX register and another MMX register or 64-bit memory location and writes the minimum value of each comparison in the destination MMX register.



Related Instructions

PMAXSW, PMAXUB, PMINSW

rFLAGS Affected

None

Exceptions

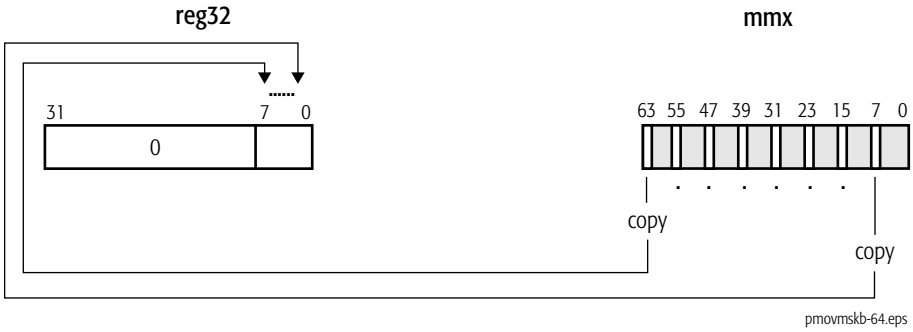
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0 and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PMOVMSKB Packed Move Mask Byte

Moves the most-significant bit of each byte in the source operand in bitwise order to the low order byte of the destination operand. The upper 24 bits of the destination operand are cleared to zeros. The destination operand is a 32-bit general-purpose register and the source operand is an MMX register.

The PMOVMSKB instruction is an AMD extension to MMX™ instruction set and is an SSE1 instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PMOVMSKB *reg32*, *mmx*	0F D7 */r*	Moves most-significant bit of each byte in an MMX register to the low-order byte of a 32-bit general-purpose register.



Related Instructions

MOVMSKPD, MOVMSKPS

rFLAGS Affected

None

Exceptions

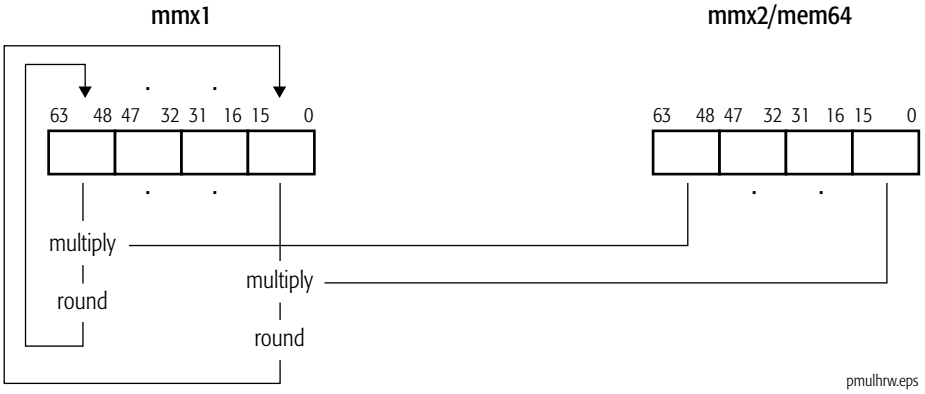
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0 and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

PMULHRW Packed Multiply High Rounded Word

Multiplies each of the four packed 16-bit signed integer values in the first source operand by the corresponding packed 16-bit integer value in the second source operand, adds 8000h to the lower 16 bits of the intermediate 32-bit result of each multiplication, and writes the high-order 16 bits of each result in the corresponding word of the destination (first source). The addition of 8000h results in the rounding of the result, providing a numerically more accurate result than the PMULHW instruction, which truncates the result. The first source/destination operand is an MMX register. The second source operand is another MMX register or 64-bit memory location.

The PMULHRW instruction is an AMD 3DNow!™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PMULHRW *mmx1*, *mmx2*/*mem64*	0F 0F /r B7	Multiply 16-bit signed integer values in an MMX register and another MMX register or 64-bit memory location and write rounded result in the destination MMX register.



Related Instructions

None

rFLAGS Affected

None

Exceptions

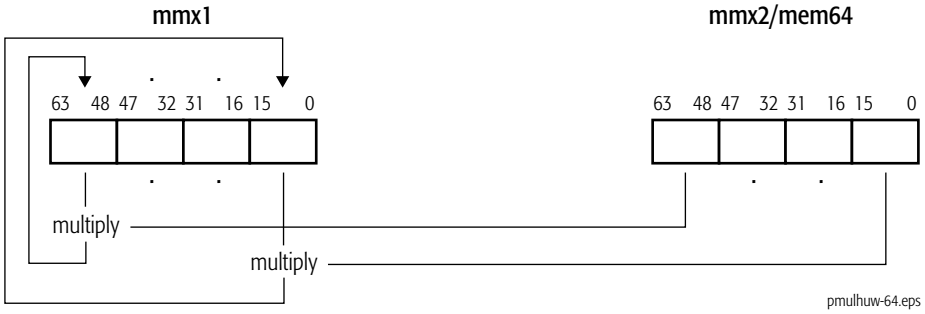
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD 3DNow!™ instructions are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNow] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PMULHUW Packed Multiply High Unsigned Word

Multiplies each packed unsigned 16-bit values in the first source operand by the corresponding packed unsigned word in the second source operand and writes the high-order 16 bits of each intermediate 32-bit result in the corresponding word of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PMULHUW instruction is an AMD extension to MMX™ instruction set and is an SSE1 instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PMULHUW *mmx1*, *mmx2*/*mem64*	0F E4 */r*	Multiplies packed 16-bit values in an MMX register by the packed 16-bit values in another MMX register or 64-bit memory location and writes the high-order 16 bits of each result in the destination MMX register.



Related Instructions

PMADDWD, PMULHW, PMULLW, PMULUDQ

rFLAGS Affected

None

Exceptions

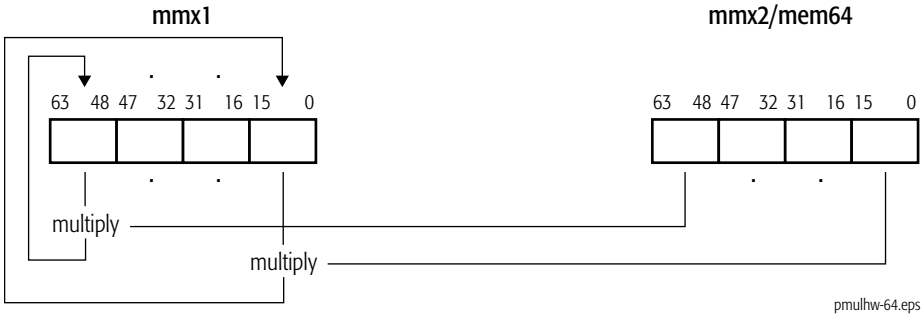
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0 and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PMULHW Packed Multiply High Signed Word

Multiplies each packed 16-bit signed integer value in the first source operand by the corresponding packed 16-bit signed integer in the second source operand and writes the high-order 16 bits of the intermediate 32-bit result of each multiplication in the corresponding word of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PMULHW instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PMULHW *mmx1*, *mmx2/mem64*	0F E5 */r*	Multiplies packed 16-bit signed integer values in an MMX register and another MMX register or 64-bit memory location and writes the high-order 16 bits of each result in the destination MMX register.



Related Instructions

PMADDWD, PMULHUW, PMULLW, PMULUDQ

rFLAGS Affected

None

Exceptions

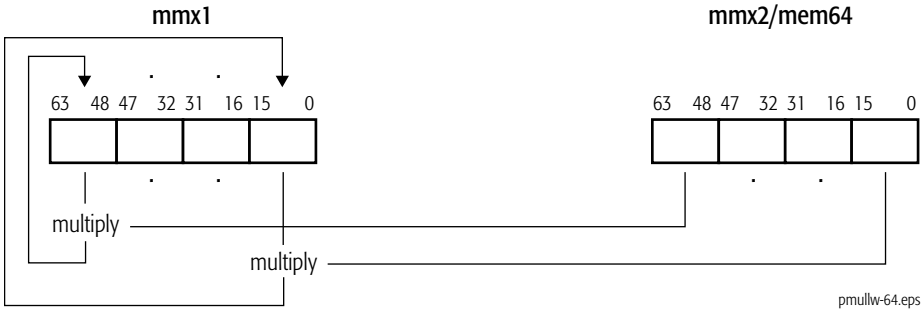
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PMULLW Packed Multiply Low Signed Word

Multiplies each packed 16-bit signed integer value in the first source operand by the corresponding packed 16-bit signed integer in the second source operand and writes the low-order 16 bits of the intermediate 32-bit result of each multiplication in the corresponding word of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PMULLW instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PMULLW *mmx1*, *mmx2/mem64*	0F D5 */r*	Multiplies packed 16-bit signed integer values in an MMX register and another MMX register or 64-bit memory location and writes the low-order 16 bits of each result in the destination MMX register.



Related Instructions

PMADDWD, PMULHUW, PMULHW, PMULUDQ

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PMULUDQ Packed Multiply Unsigned Doubleword and Store Quadword

Multiplies two 32-bit unsigned integer values in the low-order doubleword of the first and second source operands and writes the 64-bit result in the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PMULUDQ instruction is an SSE2 instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PMULUDQ *mmx1*, *mmx2/mem64*	0F F4 */r*	Multiplies low-order 32-bit unsigned integer value in an MMX register and another MMX register or 64-bit memory location and writes the 64-bit result in the destination MMX register.

Related Instructions

PMADDWD, PMULHUW, PMULHW, PMULLW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

POR Packed Logical Bitwise OR

Performs a bitwise logical OR of the values in the first and second source operands and writes the result in the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The POR instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
POR *mmx1*, *mmx2/mem64*	0F EB */r*	Performs bitwise logical OR of values in an MMX register and in another MMX register or 64-bit memory location and writes the result in the destination MMX register.

Related Instructions

PAND, PANDN, PXOR

rFLAGS Affected

None

Exceptions

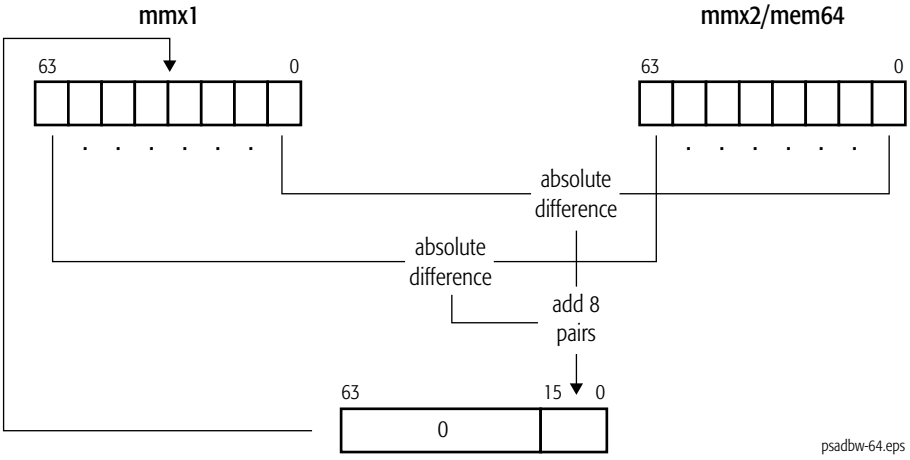
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PSADBW Packed Sum of Absolute Differences of Bytes Into a Word

Computes the absolute differences of eight corresponding packed 8-bit unsigned integers in the first and second source operands and writes the unsigned 16-bit integer result of the sum of the eight differences in a word in the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location. The result is stored in the low-order word of the destination operand, and the remaining bytes in the destination are cleared to all 0s.

The PSADBW instruction is an AMD extension to MMX™ instruction set and is an SSE1 instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSADBW *mmx1*, *mmx2/mem64*	0F F6 */r*	Compute the sum of the absolute differences of packed 8-bit unsigned integer values in an MMX register and another MMX register or 64-bit memory location and writes the 16-bit unsigned integer result in the destination MMX register.



rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0 and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PSHUFW Packed Shuffle Words

Moves any one of the four packed words in an MMX register or 64-bit memory location to a specified word location in another MMX register. In each case, the selection of the value of the destination word is determined by a two-bit field in the immediate-byte operand, with bits 0 and 1 selecting the contents of the low-order word, bits 2 and 3 selecting the second word, bits 4 and 5 selecting the third word, and bits 6 and 7 selecting the high-order word. Refer to Table 1-20 on page 167. A word in the source operand may be copied to more than one word in the destination.

The PSHUFW instruction is an AMD extension to MMX™ instruction set and is an SSE1 instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSHUFW *mmx1*, *mmx2/mem64*, *imm8*	0F 70 */r ib*	Shuffles packed 16-bit values in an MMX register or 64-bit memory location and puts the result in another MMX register.

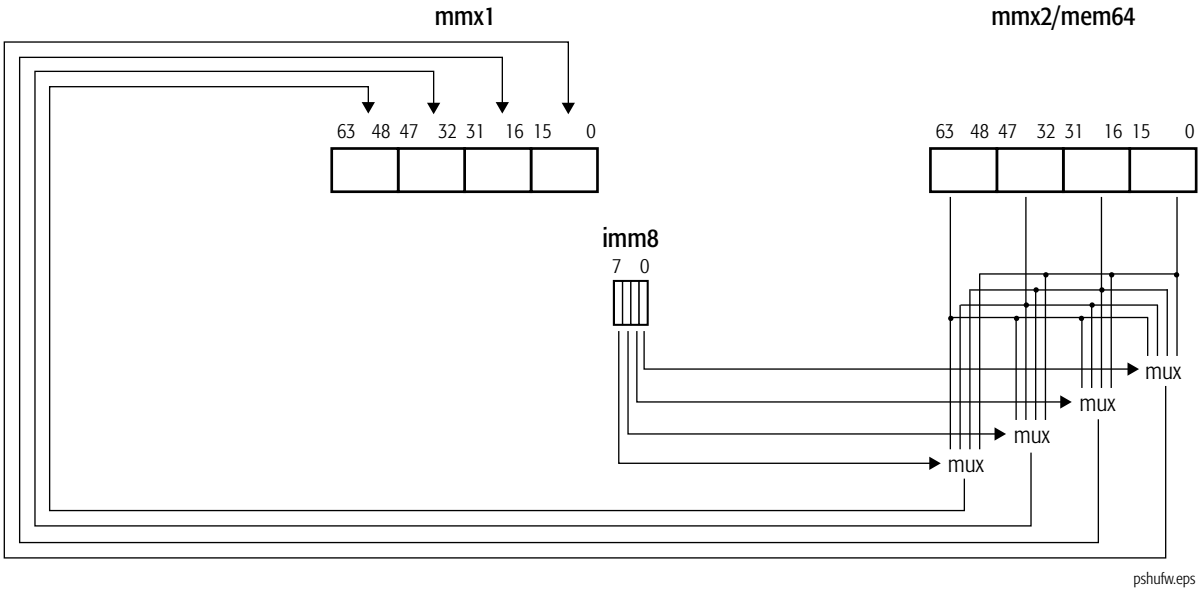


Table 1-20. Immediate-Byte Operand Encoding for PSHUFW

Destination Bits Filled	Immediate-Byte Bit Field	Value of Bit Field	Source Bits Moved
15–0	1–0	0	15–0
		1	31–16
		2	47–32
		3	63–48
31–16	3–2	0	15–0
		1	31–16
		2	47–32
		3	63–48
47–32	5–4	0	15–0
		1	31–16
		2	47–32
		3	63–48
63–48	7–6	0	15–0
		1	31–16
		2	47–32
		3	63–48

Related Instructions

PSHUFD, PSHUFHW, PSHUFLW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE1 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE] = 0 and the AMD extensions to the MMX™ instruction set are not supported, as indicated by CPUID Fn8000_0001_EDX[MmxExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PSLLD Packed Shift Left Logical Doublewords

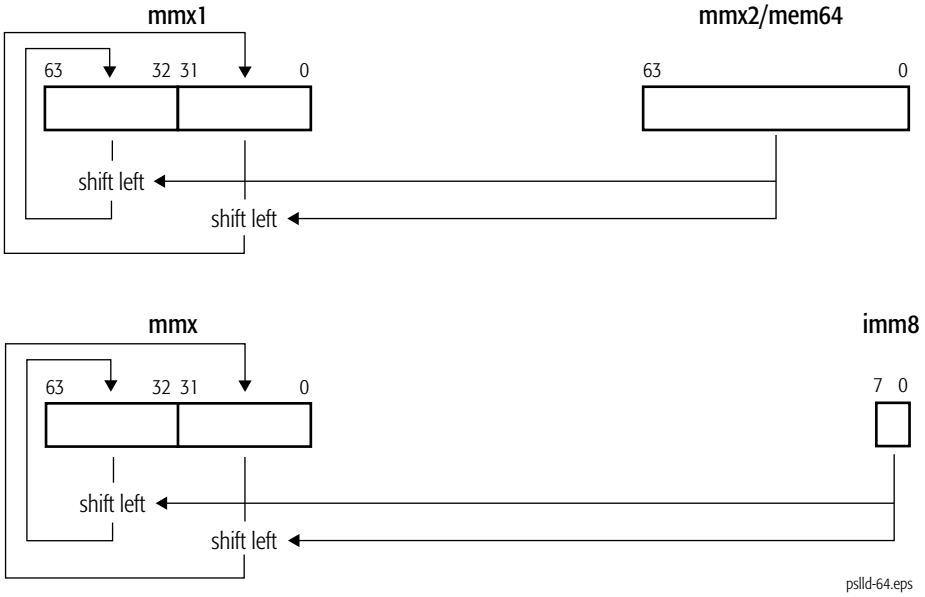
Left-shifts each of the packed 32-bit values in the first source operand by the number of bits specified in the second source operand and writes each shifted value in the corresponding doubleword of the destination (first source). The first source/destination and second source operands are:

- an MMX register and another MMX register or 64-bit memory location, or
- an MMX register and an immediate byte value.

The low-order bits that are emptied by the shift operation are cleared to 0. If the shift value is greater than 31, the destination is cleared to all 0s.

The PSLLD instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSLLD *mmx1*, *mmx2/mem64*	0F F2 */r*	Left-shifts packed doublewords in an MMX register by the amount specified in an MMX register or 64-bit memory location.
PSLLD *mmx*, *imm8*	0F 72 /6 *ib*	Left-shifts packed doublewords in an MMX register by the amount specified in an immediate byte value.



Related Instructions

PSLLDQ, PSLLQ, PSLLW, PSRAD, PSRAW, PSRLD, PSRLDQ, PSRLQ, PSRLW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PSLLQ Packed Shift Left Logical Quadwords

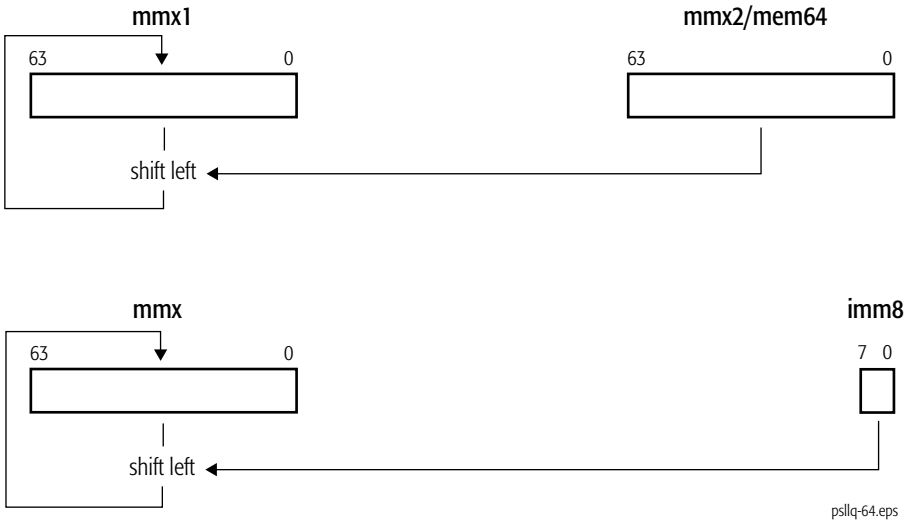
Left-shifts each 64-bit value in the first source operand by the number of bits specified in the second source operand and writes each shifted value in the corresponding quadword of the destination (first source). The first source/destination and second source operands are:

- an MMX register and another MMX register or 64-bit memory location, or
- an MMX register and an immediate byte value.

The low-order bits that are emptied by the shift operation are cleared to 0. If the shift value is greater than 63, the destination is cleared to all 0s.

The PSLLQ instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSLLQ *mmx1*, *mmx2/mem64*	0F F3 */r*	Left-shifts quadword in an MMX register by the amount specified in an MMX register or 64-bit memory location.
PSLLQ *mmx, imm8*	0F 73 /6 *ib*	Left-shifts quadword in an MMX register by the amount specified in an immediate byte value.



Related Instructions

PSLLD, PSLLDQ, PSLLW, PSRAD, PSRAW, PSRLD, PSRLDQ, PSRLQ, PSRLW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PSLLW Packed Shift Left Logical Words

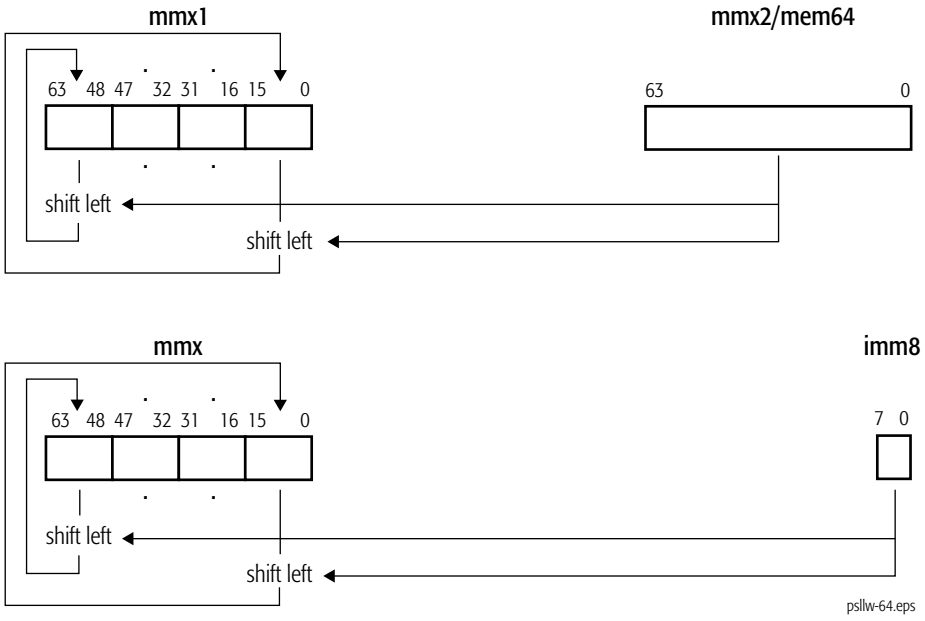
Left-shifts each of the packed 16-bit values in the first source operand by the number of bits specified in the second source operand and writes each shifted value in the corresponding word of the destination (first source). The first source/destination and second source operands are:

- an MMX register and another MMX register or 64-bit memory location, or
- an MMX register and an immediate byte value.

The low-order bits that are emptied by the shift operation are cleared to 0. If the shift value is greater than 15, the destination is cleared to all 0s.

The PSLLW instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSLLW *mmx1*, *mmx2/mem64*	0F F1 */r*	Left-shifts packed words in an MMX register by the amount specified in an MMX register or 64-bit memory location.
PSLLW *mmx, imm8*	0F 71 /6 *ib*	Left-shifts packed words in an MMX register by the amount specified in an immediate byte value.



Related Instructions

PSLLD, PSLLDQ, PSLLQ, PSRAD, PSRAW, PSRLD, PSRLDQ, PSRLQ, PSRLW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PSRAD Packed Shift Right Arithmetic Doublewords

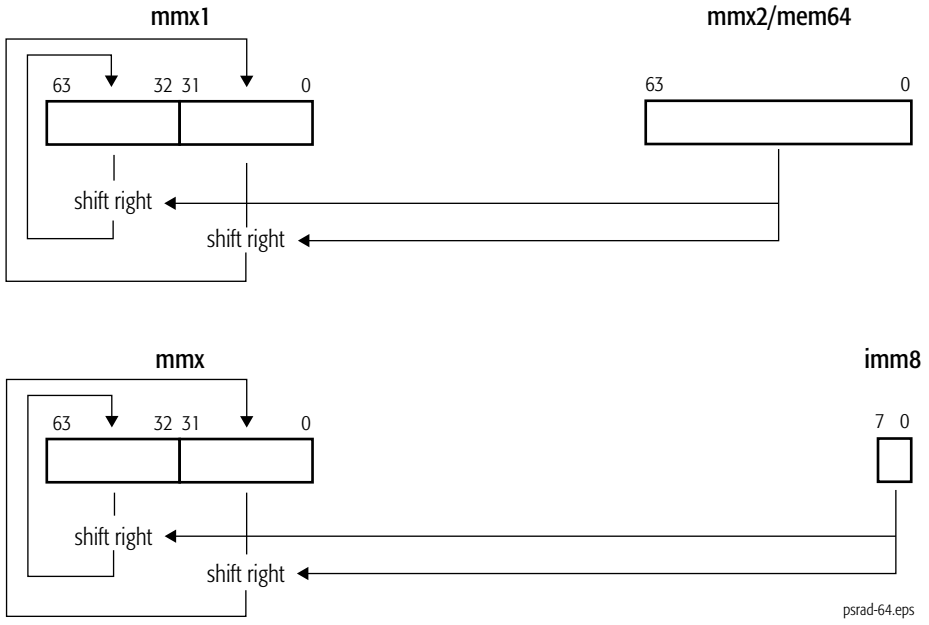
Right-shifts each of the packed 32-bit values in the first source operand by the number of bits specified in the second source operand and writes each shifted value in the corresponding doubleword of the destination (first source). The first source/destination and second source operands are:

- an MMX register and another MMX register or 64-bit memory location, or
- an MMX register and an immediate byte value.

The high-order bits that are emptied by the shift operation are filled with the sign bit of the doubleword's initial value. If the shift value is greater than 31, each doubleword in the destination is filled with the sign bit of the doubleword's initial value.

The PSRAD instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See "CPUID" in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSRAD *mmx1*, *mmx2/mem64*	0F E2 */r*	Right-shifts packed doublewords in an MMX register by the amount specified in an MMX register or 64-bit memory location.
PSRAD *mmx, imm8*	0F 72 /4 *ib*	Right-shifts packed doublewords in an MMX register by the amount specified in an immediate byte value.



Related Instructions

PSLLD, PSLLDQ, PSLLQ, PSLLW, PSRAW, PSRLD, PSRLDQ, PSRLQ, PSRLW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PSRAW Packed Shift Right Arithmetic Words

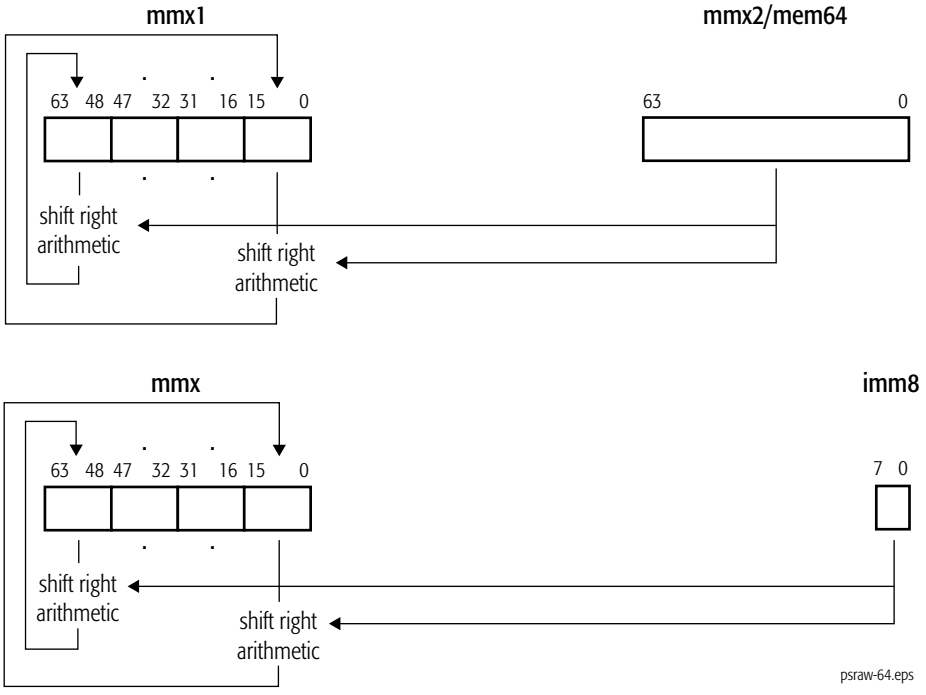
Right-shifts each of the packed 16-bit values in the first source operand by the number of bits specified in the second source operand and writes each shifted value in the corresponding word of the destination (first source). The first source/destination and second source operands are:

- an MMX register and another MMX register or 64-bit memory location, or
- an MMX register and an immediate byte value.

The high-order bits that are emptied by the shift operation are filled with the sign bit of the word's initial value. If the shift value is greater than 15, each word in the destination is filled with the sign bit of the word's initial value.

The PSRAW instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See "CPUID" in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSRAW *mmx1*, *mmx2/mem64*	0F E1 */r*	Right-shifts packed words in an MMX register by the amount specified in an MMX register or 64-bit memory location.
PSRAW *mmx, imm8*	0F 71 /4 *ib*	Right-shifts packed words in an MMX register by the amount specified in an immediate byte value.



Related Instructions

PSLLD, PSLLDQ, PSLLQ, PSLLW, PSRAD, PSRLD, PSRLDQ, PSRLQ, PSRLW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PSRLD Packed Shift Right Logical Doublewords

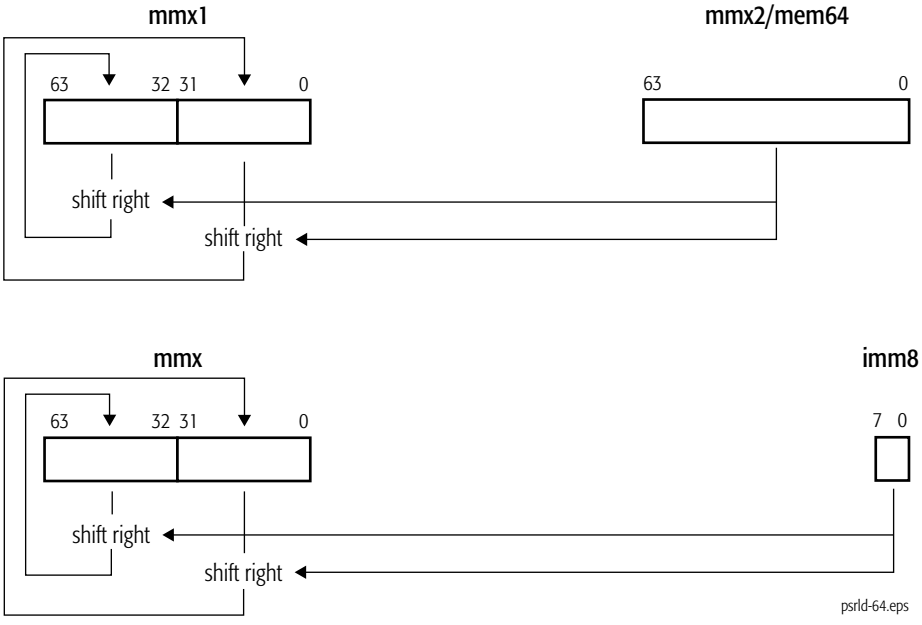
Right-shifts each of the packed 32-bit values in the first source operand by the number of bits specified in the second source operand and writes each shifted value in the corresponding doubleword of the destination (first source). The first source/destination and second source operands are:

- an MMX register and another MMX register or 64-bit memory location, or
- an MMX register and an immediate byte value.

The high-order bits that are emptied by the shift operation are cleared to 0. If the shift value is greater than 31, the destination is cleared to 0.

The PSRLD instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSRLD *mmx1*, *mmx2/mem64*	0F D2 */r*	Right-shifts packed doublewords in an MMX register by the amount specified in an MMX register or 64-bit memory location.
PSRLD *mmx, imm8*	0F 72 /2 *ib*	Right-shifts packed doublewords in an MMX register by the amount specified in an immediate byte value.



Related Instructions

PSLLD, PSLLDQ, PSLLQ, PSLLW, PSRAD, PSRAW, PSRLDQ, PSRLQ, PSRLW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PSRLQ Packed Shift Right Logical Quadwords

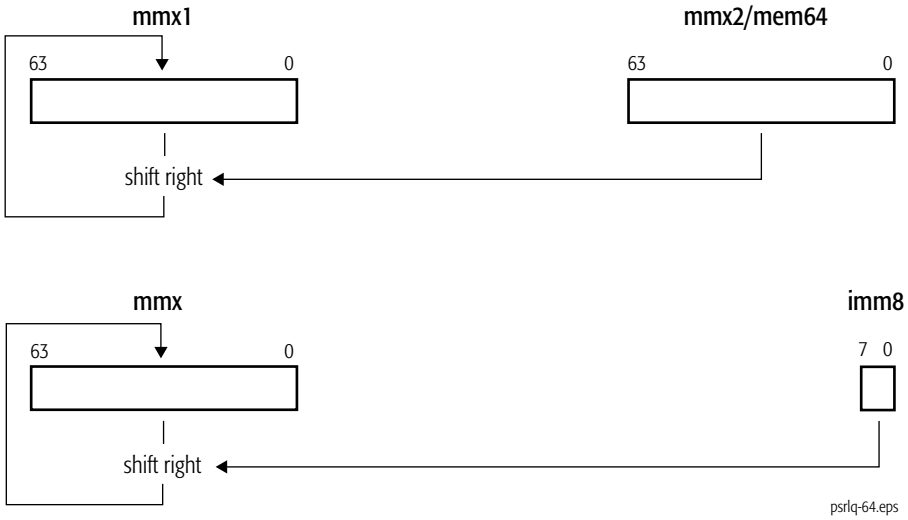
Right-shifts each 64-bit value in the first source operand by the number of bits specified in the second source operand and writes each shifted value in the corresponding quadword of the destination (first source). The first source/destination and second source operands are:

- an MMX register and another MMX register or 64-bit memory location, or
- an MMX register and an immediate byte value.

The high-order bits that are emptied by the shift operation are cleared to 0. If the shift value is greater than 63, the destination is cleared to 0.

The PSRLQ instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSRLQ *mmx1*, *mmx2/mem64*	0F D3 */r*	Right-shifts quadword in an MMX register by the amount specified in an MMX register or 64-bit memory location.
PSRLQ *mmx, imm8*	0F 73 /2 *ib*	Right-shifts quadword in an MMX register by the amount specified in an immediate byte value.



Related Instructions

PSLLD, PSLLDQ, PSLLQ, PSLLW, PSRAD, PSRAW, PSRLD, PSRLDQ, PSRLW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PSRLW Packed Shift Right Logical Words

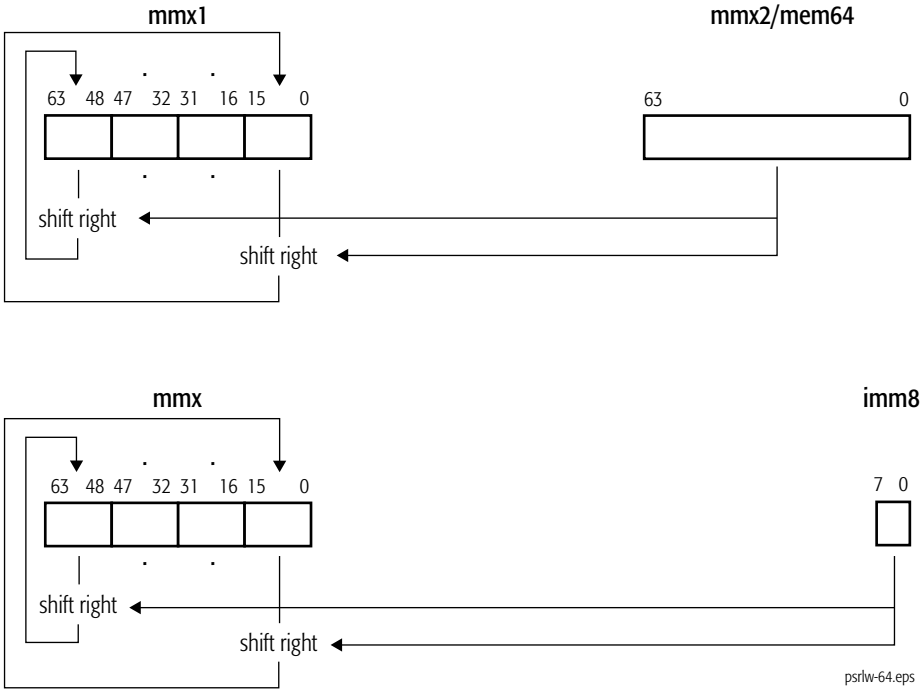
Right-shifts each of the packed 16-bit values in the first source operand by the number of bits specified in the second operand and writes each shifted value in the corresponding word of the destination (first source). The first source/destination and second source operands are:

- an MMX register and another MMX register or 64-bit memory location, or
- an MMX register and an immediate byte value.

The high-order bits that are emptied by the shift operation are cleared to 0. If the shift value is greater than 15, the destination is cleared to 0.

The PSRLW instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSRLW *mmx1*, *mmx2/mem64*	0F D1 */r*	Right-shifts packed words in an MMX register by the amount specified in an MMX register or 64-bit memory location.
PSRLW *mmx*, *imm8*	0F 71 /2 *ib*	Right-shifts packed words in an MMX register by the amount specified in an immediate byte value.



Related Instructions

PSLLD, PSLLDQ, PSLLQ, PSLLW, PSRAD, PSRAW, PSRLD, PSRLDQ, PSRLQ

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

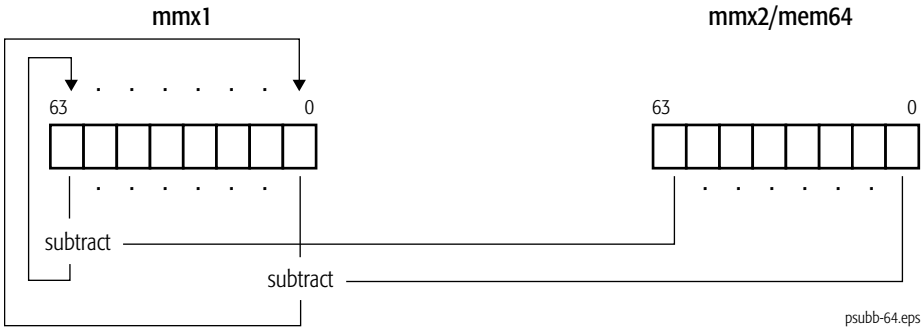
PSUBB Packed Subtract Bytes

Subtracts each packed 8-bit integer value in the second source operand from the corresponding packed 8-bit integer in the first source operand and writes the integer result of each subtraction in the corresponding byte of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

This instruction operates on both signed and unsigned integers. If the result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 8 bits of each result are written in the destination.

The PSUBB instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSUBB *mmx1*, *mmx2/mem64*	0F F8 */r*	Subtracts packed byte integer values in an MMX register or 64-bit memory location from packed byte integer values in another MMX register and writes the result in the destination MMX register.



Related Instructions

PSUBD, PSUBQ, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW, PSUBW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

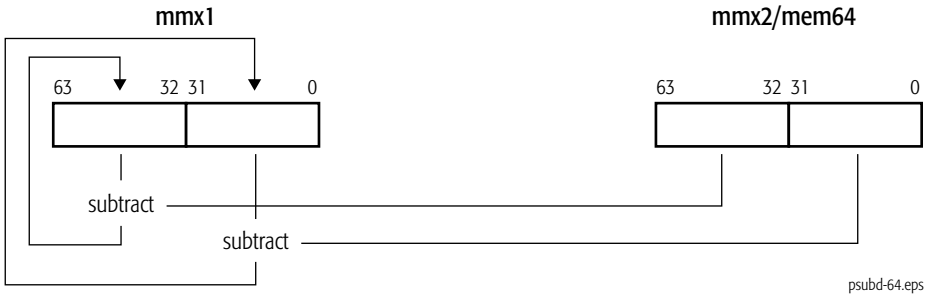
PSUBD Packed Subtract Doublewords

Subtracts each packed 32-bit integer value in the second source operand from the corresponding packed 32-bit integer in the first source operand and writes the integer result of each subtraction in the corresponding doubleword of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

This instruction operates on both signed and unsigned integers. If the result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 32 bits of each result are written in the destination.

The PSUBD instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSUBD *mmx1*, *mmx2/mem64*	0F FA */r*	Subtracts packed 32-bit integer values in an MMX register or 64-bit memory location from packed 32-bit integer values in another MMX register and writes the result in the destination MMX register.



Related Instructions

PSUBB, PSUBQ, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW, PSUBW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PSUBQ Packed Subtract Quadword

Subtracts each packed 64-bit integer value in the second source operand from the corresponding packed 64-bit integer in the first source operand and writes the integer result of each subtraction in the corresponding quadword of the destination (first source). The first source/destination and source operands are an MMX register and another MMX register or 64-bit memory location.

The PSUBQ instruction is an SSE2 instruction; check the status of EDX bit 26 returned by CPUID function 0000_0001h. See “CPUID” in Volume 3 for more information about the CPUID instruction.

This instruction operates on both signed and unsigned integers. If the result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 64 bits of each result are written in the destination.

Mnemonic	Opcode	Description
PSUBQ *mmx1*, *mmx2/mem64*	0F FB */r*	Subtracts packed 64-bit integer values in an MMX register or 64-bit memory location from packed 64-bit integer values in another MMX register and writes the result in the destination MMX register.

Related Instructions

PSUBB, PSUBD, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW, PSUBW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

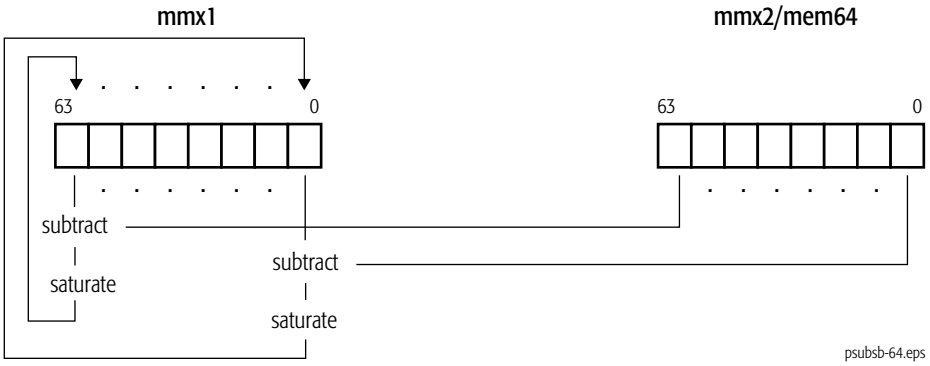
PSUBSB Packed Subtract Signed With Saturation Bytes

Subtracts each packed 8-bit signed integer value in the second source operand from the corresponding packed 8-bit signed integer in the first source operand and writes the signed integer result of each subtraction in the corresponding byte of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

For each packed value in the destination, if the value is larger than the largest signed 8-bit integer, it is saturated to 7Fh, and if the value is smaller than the smallest signed 8-bit integer, it is saturated to 80h.

The PSUBBSB instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSUBSB *mmx1*, *mmx2/mem64*	0F E8 */r*	Subtracts packed byte signed integer values in an MMX register or 64-bit memory location from packed byte integer values in another MMX register and writes the result in the destination MMX register.



Related Instructions

PSUBB, PSUBD, PSUBQ, PSUBSW, PSUBUSB, PSUBUSW, PSUBW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

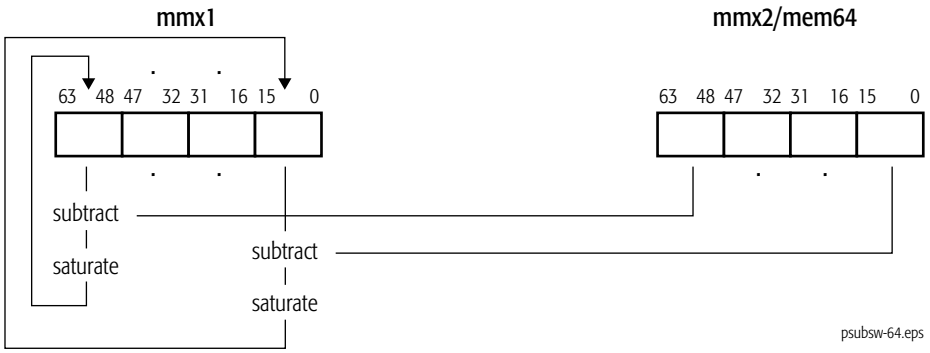
PSUBSW Packed Subtract Signed With Saturation Words

Subtracts each packed 16-bit signed integer value in the second source operand from the corresponding packed 16-bit signed integer in the first source operand and writes the signed integer result of each subtraction in the corresponding word of the destination (first source). The first source/destination and source operands are an MMX register and another MMX register or 64-bit memory location.

For each packed value in the destination, if the value is larger than the largest signed 16-bit integer, it is saturated to 7FFFh, and if the value is smaller than the smallest signed 16-bit integer, it is saturated to 8000h.

The PSUBSW instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSUBSW *mmx1*, *mmx2/mem64*	0F E9 */r*	Subtracts packed 16-bit signed integer values in an MMX register or 64-bit memory location from packed 16-bit integer values in another MMX register and writes the result in the destination MMX register.



Related Instructions

PSUBB, PSUBD, PSUBQ, PSUBSB, PSUBUSB, PSUBUSW, PSUBW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

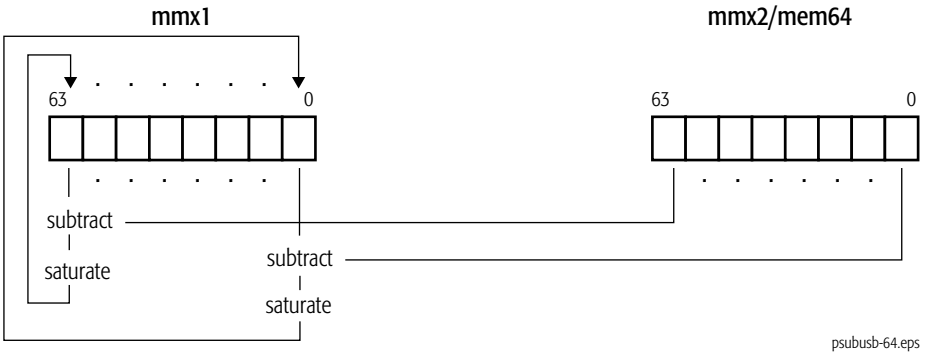
PSUBUSB Packed Subtract Unsigned and Saturate Bytes

Subtracts each packed 8-bit unsigned integer value in the second source operand from the corresponding packed 8-bit unsigned integer in the first source operand and writes the unsigned integer result of each subtraction in the corresponding byte of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

For each packed value in the destination, if the value is smaller than the smallest unsigned 8-bit integer, it is saturated to 00h.

The PSUBUSB instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSUBUSB *mmx1*, *mmx2/mem64*	0F D8 */r*	Subtracts packed byte unsigned integer values in an MMX register or 64-bit memory location from packed byte integer values in another MMX register and writes the result in the destination MMX register.



Related Instructions

PSUBB, PSUBD, PSUBQ, PSUBSB, PSUBSW, PSUBUSW, PSUBW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

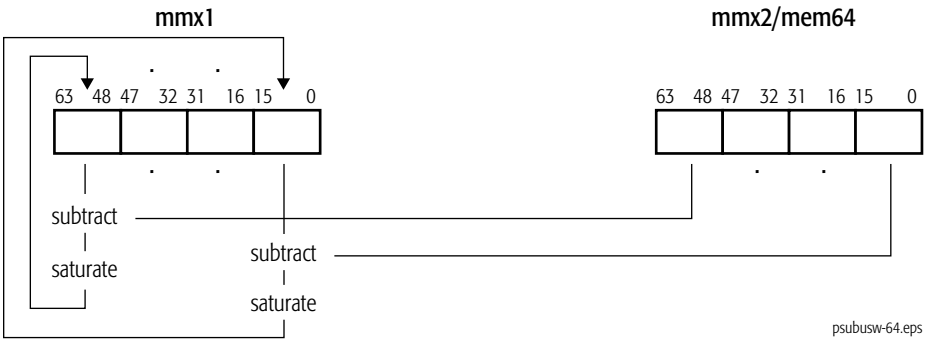
PSUBUSW Packed Subtract Unsigned and Saturate Words

Subtracts each packed 16-bit unsigned integer value in the second source operand from the corresponding packed 16-bit unsigned integer in the first source operand and writes the unsigned integer result of each subtraction in the corresponding word of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

For each packed value in the destination, if the value is smaller than the smallest unsigned 16-bit integer, it is saturated to 0000h.

The PSUBUSW instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSUBUSW *mmx1*, *mmx2/mem64*	0F D9 */r*	Subtracts packed 16-bit unsigned integer values in an MMX register or 64-bit memory location from packed 16-bit integer values in another MMX register and writes the result in the destination MMX register.



Related Instructions

PSUBB, PSUBD, PSUBQ, PSUBSB, PSUBSW, PSUBUSB, PSUBW

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

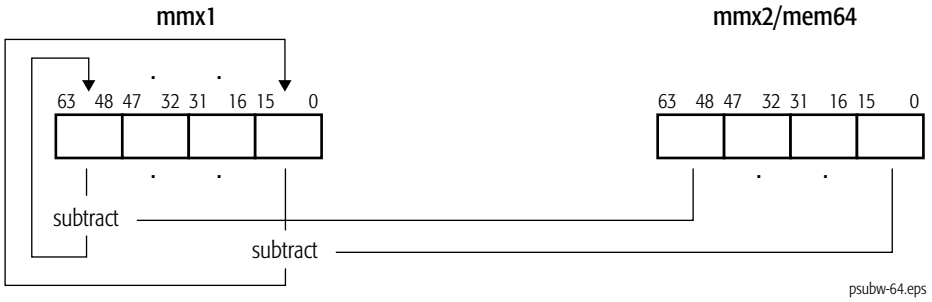
PSUBW Packed Subtract Words

Subtracts each packed 16-bit integer value in the second source operand from the corresponding packed 16-bit integer in the first source operand and writes the integer result of each subtraction in the corresponding word of the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

This instruction operates on both signed and unsigned integers. If the result overflows, the carry is ignored (neither the overflow nor carry bit in rFLAGS is set), and only the low-order 16 bits of the result are written in the destination.

The PSUBW instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSUBW *mmx1*, *mmx2/mem64*	0F F9 */r*	Subtracts packed 16-bit integer values in an MMX register or 64-bit memory location from packed 16-bit integer values in another MMX register and writes the result in the destination MMX register.



Related Instructions

PSUBB, PSUBD, PSUBQ, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW

rFLAGS Affected

None

Exceptions

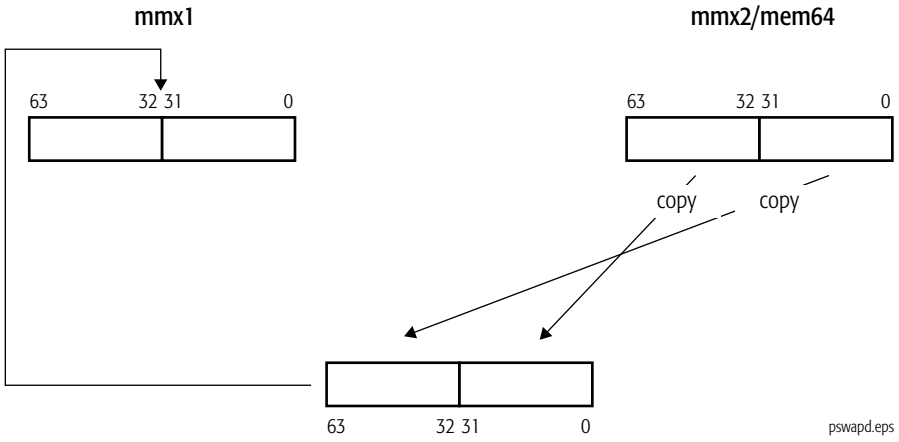
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PSWAPD Packed Swap Doubleword

Swaps (reverses) the two packed 32-bit values in the source operand and writes each swapped value in the corresponding doubleword of the destination. The source operand is an MMX register or 64-bit memory location. The destination is another MMX register.

The PSWAPD instruction is an extension to the AMD 3DNow!™ instruction set. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PSWAPD *mmx1*, *mmx2/mem64*	0F 0F /r BB	Swaps packed 32-bit values in an MMX register or 64-bit memory location and writes each value in the destination MMX register.



Related Instructions

None

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The AMD Extensions to 3DNow!™ are not supported, as indicated by CPUID Fn8000_0001_EDX[3DNowExt] = 0.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

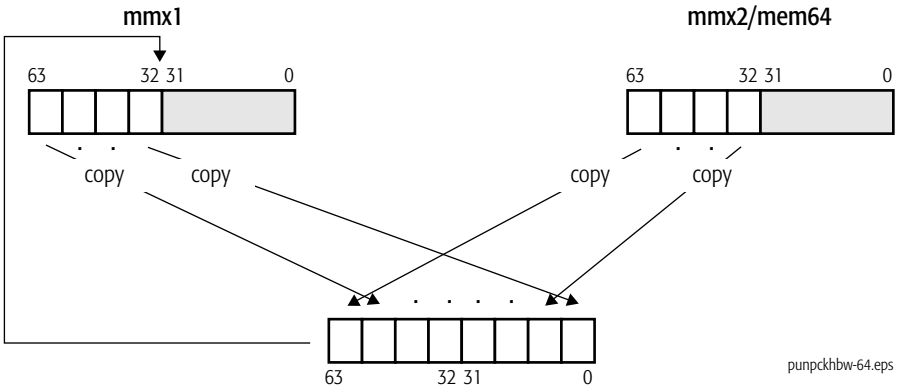
PUNPCKHBW Unpack and Interleave High Bytes

Unpacks the high-order bytes from the first and second source operands and packs them into interleaved-byte words in the destination (first source). The low-order bytes of the source operands are ignored. The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

If the second source operand is all 0s, the destination contains the bytes from the first source operand zero-extended to 16 bits. This operation is useful for expanding unsigned 8-bit values to unsigned 16-bit operands for subsequent processing that requires higher precision.

The PUNPCKHBW instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PUNPCKHBW *mmx1*, *mmx2/mem64*	0F 68 */r*	Unpacks the four high-order bytes in an MMX register and another MMX register or 64-bit memory location and packs them into interleaved bytes in the destination MMX register.



Related Instructions

PUNPCKHDQ, PUNPCKHQDQ, PUNPCKHWD, PUNPCKLBW, PUNPCKLDQ, PUNPCKLQDQ, PUNPCKLWD

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

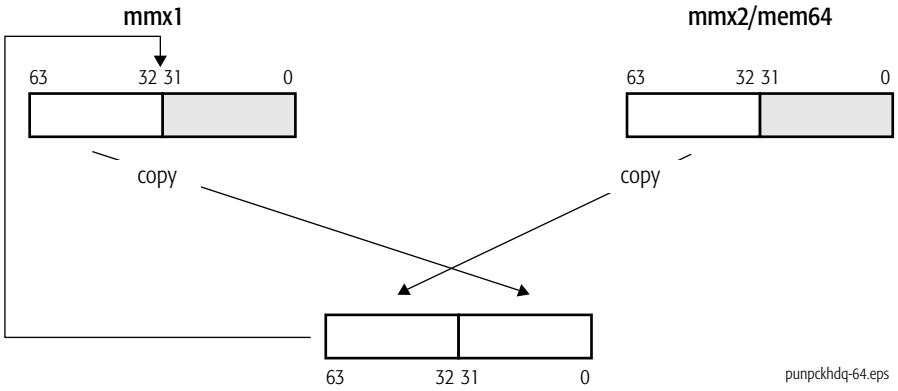
PUNPCKHDQ Unpack and Interleave High Doublewords

Unpacks the high-order doublewords from the first and second source operands and packs them into interleaved-doubleword quadwords in the destination (first source). The low-order doublewords of the source operands are ignored. The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

If the second source operand is all 0s, the destination contains the doubleword(s) from the first source operand zero-extended to 64 bits. This operation is useful for expanding unsigned 32-bit values to unsigned 64-bit operands for subsequent processing that requires higher precision.

The PUNPCKHDQ instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PUNPCKHDQ *mmx1*, *mmx2/mem64*	0F 6A */r*	Unpacks the high-order doubleword in an MMX register and another MMX register or 64-bit memory location and packs them into interleaved doublewords in the destination MMX register.



Related Instructions

PUNPCKHBW, PUNPCKHQDQ, PUNPCKHWD, PUNPCKLBW, PUNPCKLDQ, PUNPCKLQDQ, PUNPCKLWD

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

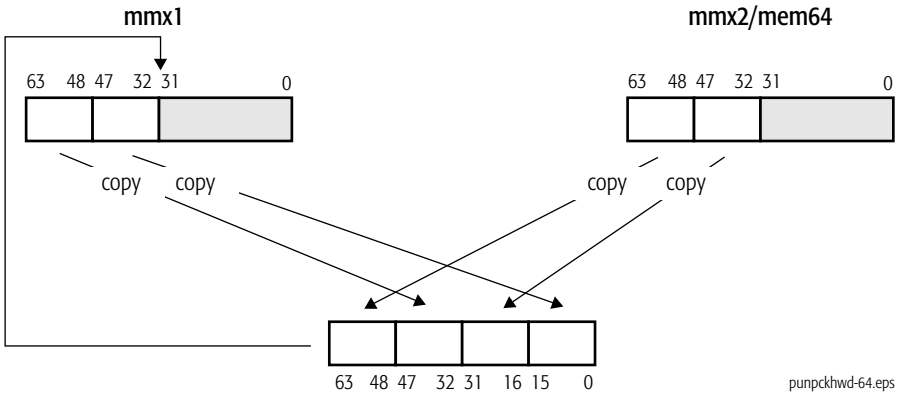
PUNPCKHWD Unpack and Interleave High Words

Unpacks the high-order words from the first and second source operands and packs them into interleaved-word doublewords in the destination (first source). The low-order words of the source operands are ignored. The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

If the second source operand is all 0s, the destination contains the words from the first source operand zero-extended to 32 bits. This operation is useful for expanding unsigned 16-bit values to unsigned 32-bit operands for subsequent processing that requires higher precision.

The PUNPCKHWD instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PUNPCKHWD *mmx1*, *mmx2/mem64*	0F 69 */r*	Unpacks two high-order words in an MMX register and another MMX register or 64-bit memory location and packs them into interleaved words in the destination MMX register.



Related Instructions

PUNPCKHBW, PUNPCKHDQ, PUNPCKHQDQ, PUNPCKLBW, PUNPCKLDQ, PUNPCKLQDQ, PUNPCKLWD

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

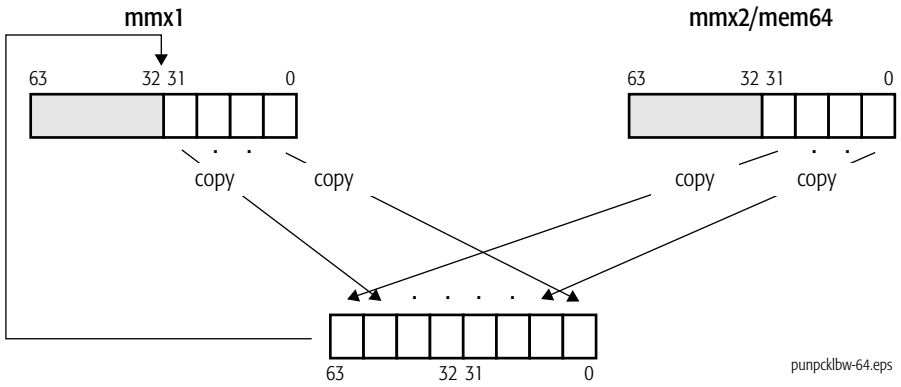
PUNPCKLBW Unpack and Interleave Low Bytes

Unpacks the low-order bytes from the first and second source operands and packs them into interleaved-byte words in the destination (first source). The high-order bytes of the source operands are ignored. The first source/destination operand is an MMX register and the second source operand is another MMX register or 32-bit memory location.

If the second source operand is all 0s, the destination contains the bytes from the first source operand zero-extended to 16 bits. This operation is useful for expanding unsigned 8-bit values to unsigned 16-bit operands for subsequent processing that requires higher precision.

The PUNPCKLBW instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PUNPCKLBW *mmx1*, *mmx2*/*mem32*	0F 60 */r*	Unpacks the four low-order bytes in an MMX register and another MMX register or 32-bit memory location and packs them into interleaved bytes in the destination MMX register.



Related Instructions

PUNPCKHBW, PUNPCKHDQ, PUNPCKHQDQ, PUNPCKHWD, PUNPCKLDQ, PUNPCKLQDQ, PUNPCKLWD

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

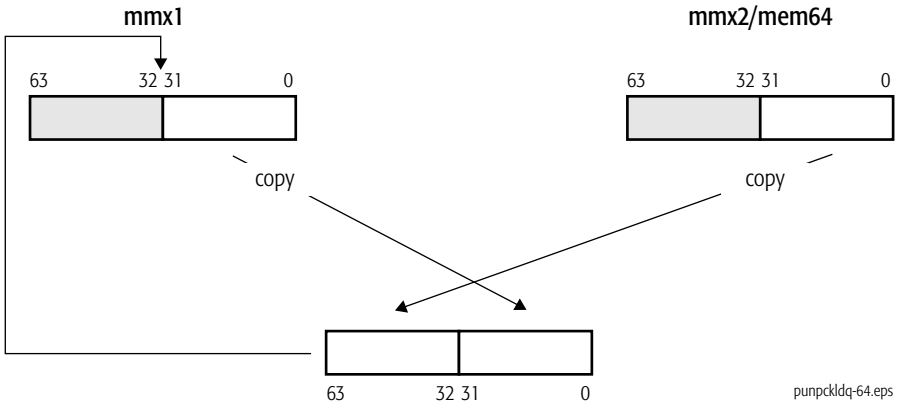
PUNPCKLDQ Unpack and Interleave Low Doublewords

Unpacks the low-order doublewords from the first and second source operands and packs them into interleaved-doubleword quadwords in the destination (first source). The high-order doublewords of the source operands are ignored. The first source/destination operand is an MMX register and the second source operand is another MMX register or 32-bit memory location.

If the second source operand is all 0s, the destination contains the doubleword(s) from the first source operand zero-extended to 64 bits. This operation is useful for expanding unsigned 32-bit values to unsigned 64-bit operands for subsequent processing that requires higher precision.

The PUNPCKLDQ instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PUNPCKLDQ *mmx1*, *mmx2/mem32*	0F 62 */r*	Unpacks the low-order doubleword in an MMX register and another MMX register or 32-bit memory location and packs them into interleaved doublewords in the destination MMX register.



Related Instructions

PUNPCKHBW, PUNPCKHDQ, PUNPCKHQDQ, PUNPCKHWD, PUNPCKLBW, PUNPCKLQDQ, PUNPCKLWD

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

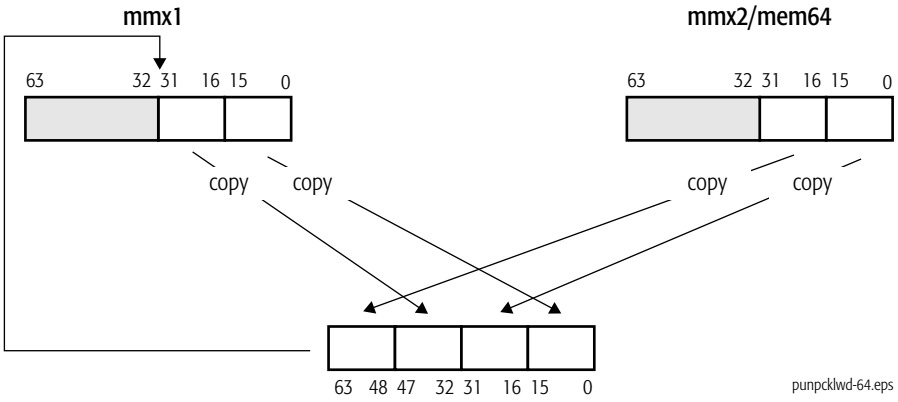
PUNPCKLWD Unpack and Interleave Low Words

Unpacks the low-order words from the first and second source operands and packs them into interleaved-word doublewords in the destination (first source). The high-order words of the source operands are ignored. The first source/destination operand is an MMX register and the second source operand is another MMX register or 32-bit memory location.

If the second source operand is all 0s, the destination contains the words from the first source operand zero-extended to 32 bits. This operation is useful for expanding unsigned 16-bit values to unsigned 32-bit operands for subsequent processing that requires higher precision.

The PUNPCKLWD instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PUNPCKLWD *mmx1*, *mmx2/mem32*	0F 61 */r*	Unpacks the two low-order words in an MMX register and another MMX register or 32-bit memory location and packs them into interleaved words in the destination MMX register.



Related Instructions

PUNPCKHBW, PUNPCKHDQ, PUNPCKHQDQ, PUNPCKHWD, PUNPCKLBW, PUNPCKLDQ, PUNPCKLQDQ

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

PXOR Packed Logical Bitwise Exclusive OR

Performs a bitwise exclusive OR of the values in the first and second source operands and writes the result in the destination (first source). The first source/destination operand is an MMX register and the second source operand is another MMX register or 64-bit memory location.

The PXOR instruction is an MMX™ instruction. The presence of this instruction set is indicated by CPUID feature bits. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
PXOR *mmx1*, *mmx2*/*mem64*	0F EF */r*	Performs bitwise logical XOR of values in an MMX register and in another MMX register or 64-bit memory location and writes the result in the destination MMX register.

Related Instructions

PAND, PANDN, POR

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The emulate bit (EM) of CR0 was set to 1.
	X	X	X	The MMX™ instructions are not supported, as indicated by EDX[MMX] = 0, returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The task-switch bit (TS) of CR0 was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

2 x87 Floating-Point Instruction Reference

This chapter describes the function, mnemonic syntax, opcodes, condition codes, affected flags, and possible exceptions generated by the x87 floating-point instructions. The x87 floating-point instructions are used in legacy floating-point applications. Most of these instructions load, store, or operate on data located in the x87 ST(0)–ST(7) stack registers (the FPR0–FPR7 physical registers). The remaining instructions within this category are used to manage the x87 floating-point environment.

The AMD64 architecture requires support of the x87 floating-point instruction subset including the floating-point conditional moves and the FCOMI(P) and FUCOMI(P) instructions. On compliant processor implementations both the FPU and the CMOV feature flags are set. These are indicated by EDX[FPU] (bit 0) and EDX[CMOV] (bit 15) respectively returned by CPUID Fn0000_0001 or CPUID Fn8000_0001.

This is augmented by instructions that are members of the MMX, 3DNow!™, SSE3, and FXSR subsets. Support for the following instructions is implementation-specific:

- EMMS, which is an MMX instruction. Support for this instruction is indicated by CPUID Fn0000_0001_EDX[MMX] = 1 or CPUID Fn8000_0001_EDX[MMX] = 1.
- FEMMS, which is a 3DNow!™ instruction. Support for this instruction is indicated by CPUID Fn8000_0001_EDX[3DNow] = 1.
- FISTTP, which is an SSE3 instruction. Support for this instruction is indicated by CPUID Fn0000_0001_ECX[SSE3] = 1.
- FXSAVE / FXRSTOR. Support for these instructions is indicated by CPUID Fn8000_0001_EDX[FXSR] = 1 or CPUID Fn0000_0001_EDX[FXSR] = 1

EMMS and FEMMS are described in Chapter 1, “64-Bit Media Instruction Reference”, on page 1.

The x87 instructions can be used in legacy mode or long mode. Their use in long mode is available if the following feature bit is set:

- Long Mode, as indicated by CPUID Fn8000_0001_EDX[LM] = 1.

Compilation of x87 media programs for execution in 64-bit mode offers two primary advantages: access to the 64-bit virtual address space and access to the RIP-relative addressing mode.

For further information about the x87 floating-point instructions and register resources, see:

- “x87 Floating-Point Programming” in Volume 1.
- “SSE, MMX, and x87 Programming” in Volume 2.
- “Summary of Registers and Data Types” in Volume 3.
- “Notation” in Volume 3.
- “Instruction Prefixes” in Volume 3.

For information on using the CPUID instruction, see the instruction description in Volume 3.

F2XM1 Floating-Point Compute 2^x-1

Raises 2 to the power specified by the value in ST(0), subtracts 1, and stores the result in ST(0). The source value must be in the range –1.0 to +1.0. The result is undefined for source values outside this range.

This instruction, when used in conjunction with the FYL2X instruction, can be applied to calculate $z = x^y$ by taking advantage of the log property $x^y = 2^{y*\log_2 x}$.

Mnemonic	Opcode	Description
F2XM1	D9 F0	Replace ST(0) with $(2^{ST(0)} - 1)$.

Related Instructions

FYL2X, FYL2XP1

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) were set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FABS Floating-Point Absolute Value

Converts the value in ST(0) to its absolute value by clearing the sign bit. The resulting value depends upon the type of number used as the source value:

Source Value (ST(0))	Result (ST(0))
-∞	+∞
-FiniteReal	+FiniteReal
-0	+0
+0	+0
+FiniteReal	+FiniteReal
+∞	+∞
NaN	NaN

This operation applies even if the value in ST(0) is negative zero or negative infinity.

Mnemonic	Opcode	Description
FABS	D9 E1	Replace ST(0) with its absolute value.

Related Instructions

FPREM, FRNDINT, FXTRACT, FCHS

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.

FADD
FADDP
FIADD

Floating-Point Add

Adds two values and stores the result in a floating-point register. If two operands are specified, the values are in ST(0) and another floating-point register and the instruction stores the result in the first register specified. If one operand is specified, the instruction adds the 32-bit or 64-bit value in the specified memory location to the value in ST(0).

The FADDP instruction adds the value in ST(0) to the value in another floating-point register and pops the register stack. If two operands are specified, the first operand is the other register. If no operand is specified, then the other register is ST(1).

The FIADD instruction reads a 16-bit or 32-bit signed integer value from the specified memory location, converts it to double-extended-real format, and adds it to the value in ST(0).

Mnemonic	Opcode	Description
FADD ST(0),ST(*i*)	D8 C0+*i*	Replace ST(0) with ST(0) + ST(*i*).
FADD ST(*i*),ST(0)	DC C0+*i*	Replace ST(*i*) with ST(0) + ST(*i*).
FADD *mem32real*	D8 /0	Replace ST(0) with ST(0) + *mem32real*.
FADD *mem64real*	DC /0	Replace ST(0) with ST(0) + *mem64real*.
FADDP	DE C1	Replace ST(1) with ST(0) + ST(1), and pop the x87 register stack.
FADDP ST(*i*),ST(0)	DE C0+*i*	Replace ST(*i*) with ST(0) + ST(*i*), and pop the x87 register stack.
FIADD *mem16int*	DE /0	Replace ST(0) with ST(0) + *mem16int*.
FIADD *mem32int*	DA /0	Replace ST(0) with ST(0) + *mem32int*.

Related Instructions

None

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	+infinity was added to –infinity.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Overflow exception (OE)	X	X	X	A rounded result was too large to fit into the format of the destination operand.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FBLD Floating-Point Load Binary-Coded Decimal

Converts a 10-byte packed BCD value in memory into double-extended-precision format, and pushes the result onto the x87 stack. In the process, it preserves the sign of the source value.

The packed BCD digits should be in the range 0 to 9. Attempting to load invalid digits (Ah through Fh) produces undefined results.

Mnemonic	Opcode	Description
FBLD *mem80dec*	DF /4	Convert a packed BCD value to floating-point and push the result onto the x87 register stack.

Related Instructions

FBSTP

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	1	x87 stack overflow, if an x87 register stack fault was detected.
	0	If no other flags are set.
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack overflow occurred.

FBSTP Floating-Point Store Binary-Coded Decimal and Pop

Converts the value in ST(0) to an 18-digit packed BCD integer, stores the result in the specified memory location, and pops the register stack. It rounds a non-integral value to an integer value, depending on the rounding mode specified by the RC field of the x87 control word.

The operand specifies the memory address of the first byte of the resulting 10-byte value.

Mnemonic	Opcode	Description
FBSTP *mem80dec*	DF /6	Convert the floating-point value in ST(0) to BCD, store the result in mem80, and pop the x87 register stack.

Related Instructions

FBLD

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a nonwritable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value, a QNaN value, ±infinity or an unsupported format.
	X	X	X	A source operand was too large to fit in the destination format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FCHS Floating-Point Change Sign

Compliments the sign bit of ST(0), changing the value from negative to positive or vice versa. This operation applies to positive and negative floating point values, as well as –0 and +0, NaNs, and +∞ and –∞.

Mnemonic	Opcode	Description
FCHS	D9 E0	Reverse the sign bit of ST(0).

Related Instructions

FABS, FPREM, FRNDINT, FXTRACT

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.

FCLEX (FNCLEX) Floating-Point Clear Flags

Clears the following flags in the x87 status word:

- Floating-point exception flags (PE, UE, OE, ZE, DE, and IE)
- Stack fault flag (SF)
- Exception summary status flag (ES)
- Busy flag (B)

It leaves the four condition-code bits undefined. It does not check for possible floating-point exceptions before clearing the flags.

Assemblers usually provide an FCLEX macro that expands into the instruction sequence

```
WAIT                    ; Opcode 9B
FNCLEX destination      ; Opcode DB E2
```

The WAIT (9Bh) instruction checks for pending x87 exceptions and calls an exception handler, if necessary. The FNCLEX instruction then clears all the relevant x87 exception flags.

Mnemonic	Opcode	Description
FCLEX	9B DB E2	Perform a WAIT (9B) to check for pending floating-point exceptions, and then clear the floating-point exception flags.
FNCLEX	DB E2	Clear the floating-point flags without checking for pending unmasked floating-point exceptions.

Related Instructions

WAIT

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	U	
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.

FCMOV*cc* Floating-Point Conditional Move

Tests the flags in the rFLAGS register and, depending upon the values encountered, moves the value in another stack register to ST(0).

This set of instructions includes the mnemonics FCMOVB, FCMOVBE, FCMOVE, FCMOVNB, FCMOVNBE, FCMOVNE, FCMOVNU, and FCMOVU.

Support for the FCMOV*cc* instruction is indicated when both EDX[FPU] (bit 0) and EDX[CMOV] (bit 15) are set, as returned by either CPUID function 0000_0001h or function 8000_0001h.

Mnemonic	Opcode	Description
FCMOVB ST(0),ST(*i*)	DA C0+*i*	Move the contents of ST(*i*) into ST(0) if below (CF = 1).
FCMOVBE ST(0),ST(*i*)	DA D0+*i*	Move the contents of ST(*i*) into ST(0) if below or equal (CF = 1 or ZF = 1).
FCMOVE ST(0),ST(*i*)	DA C8+*i*	Move the contents of ST(*i*) into ST(0) if equal (ZF = 1).
FCMOVNB ST(0),ST(*i*)	DB C0+*i*	Move the contents of ST(*i*) into ST(0) if not below (CF = 0).
FCMOVNBE ST(0),ST(*i*)	DB D0+*i*	Move the contents of ST(*i*) into ST(0) if not below or equal (CF = 0 and ZF = 0).
FCMOVNE ST(0),ST(*i*)	DB C8+*i*	Move the contents of ST(*i*) into ST(0) if not equal (ZF = 0).
FCMOVNU ST(0),ST(*i*)	DB D8+*i*	Move the contents of ST(*i*) into ST(0) if not unordered (PF = 0).
FCMOVU ST(0),ST(*i*)	DA D8+*i*	Move the contents of ST(*i*) into ST(0) if unordered (PF = 1).

Related Instructions

None

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The Conditional Move instructions are not supported, as indicated by EDX[FPU] and EDX[CMOV] returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.

FCOM
FCOMP
FCOMPP
Floating-Point Compare

Compares the specified value to the value in ST(0) and sets the C0, C2, and C3 condition code flags in the x87 status word as shown in the x87 Condition Code table below. The specified value can be in a floating-point register or a memory location.

The no-operand version compares the value in ST(1) with the value in ST(0).

The comparison operation ignores the sign of zero (–0.0 = +0.0).

After performing the comparison operation, the FCOMP instruction pops the x87 register stack and the FCOMPP instruction pops the x87 register stack twice.

If either or both of the compared values is a NaN or is in an unsupported format, the FCOMx instruction sets the invalid-operation exception (IE) bit in the x87 status word to 1, and sets the condition flags to 'unordered.'

The FUCOMx instructions perform the same operations as the FCOMx instructions, but do not set the IE bit for QNaNs.

Mnemonic	Opcode	Description
FCOM	D8 D1	Compare the contents of ST(0) to the contents of ST(1) and set condition flags to reflect the results of the comparison.
FCOM ST(*i*)	D8 D0+*i*	Compare the contents of ST(0) to the contents of ST(*i*) and set condition flags to reflect the results of the comparison.
FCOM *mem32real*	D8 /2	Compare the contents of ST(0) to the contents of *mem32real* and set condition flags to reflect the results of the comparison.
FCOM *mem64real*	DC /2	Compare the contents of ST(0) to the contents of *mem64real* and set condition flags to reflect the results of the comparison.
FCOMP	D8 D9	Compare the contents of ST(0) to the contents of ST(1), set condition flags to reflect the results of the comparison, and pop the x87 register stack.
FCOMP ST(*i*)	D8 D8+*i*	Compare the contents of ST(0) to the contents of ST(*i*), set condition flags to reflect the results of the comparison, and pop the x87 register stack.
FCOMP *mem32real*	D8 /3	Compare the contents of ST(0) to the contents of *mem32real*, set condition flags to reflect the results of the comparison, and pop the x87 register stack.

FCOMP *mem64real*	DC /3	Compare the contents of ST(0) to the contents of *mem64real*, set condition flags to reflect the results of the comparison, and pop the x87 register stack.
FCOMPP	DE D9	Compare the contents of ST(0) to the contents of ST(1), set condition flags to reflect the results of the comparison, and pop the x87 register stack twice.

Related Instructions

FCOMI, FCOMIP, FICOM, FICOMP, FTST, FUCOMI, FUCOMIP, FXAM

rFLAGS Affected

None

x87 Condition Code

C3	C2	C1	C0	Compare Result
0	0	0	0	ST(0) > source
0	0	0	1	ST(0) < source
1	0	0	0	ST(0) = source
1	1	0	1	Operands were unordered

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value, a QNaN value, or an unsupported format.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.

FCOMI FCOMIP Floating-Point Compare and Set Flags

Compares the value in ST(0) with the value in another floating-point register and sets the zero flag (ZF), parity flag (PF), and carry flag (CF) in the rFLAGS register based on the result as shown in the table in the x87 Condition Code section.

The comparison operation ignores the sign of zero (–0.0 = +0.0).

After performing the comparison operation, FCOMIP pops the x87 register stack.

If either or both of the compared values is a NaN or is in an unsupported format, the FCOMIx instruction sets the invalid-operation exception (IE) bit in the x87 status word to 1 and sets the flags to "unordered."

The FUCOMIx instructions perform the same operations as the FCOMIx instructions, but do not set the IE bit for QNaNs.

Support for the FCOMI*x* instruction can be determined by executing either CPUID Fn0000_0001 or CPUID Fn8000_0001. Support is indicated when both EDX[FPU] (bit 0) and EDX[CMOV] (bit 15) are set.

Mnemonic	Opcode	Description
FCOMI ST(0),ST(*i*)	DB F0+*i*	Compare the contents of ST(0) with the contents of ST(*i*) and set status flags to reflect the results of the comparison.
FCOMIP ST(0),ST(*i*)	DF F0+*i*	Compare the contents of ST(0) with the contents of ST(*i*), set status flags to reflect the results of the comparison, and pop the x87 register stack.

Related Instructions

FCOM, FCOMPP, FICOM, FICOMP, FTST, FUCOMI, FUCOMIP, FXAM

rFLAGS Affected

ZF	PF	CF	Compare Result
0	0	0	ST(0) > source
0	0	1	ST(0) < source
1	0	0	ST(0) = source
1	1	1	Operands were unordered

x87 Condition Code

x87 Condition Code	Value	Description
C0		
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
C2		
C3		

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The conditional move instructions are not supported, as indicated by EDX[FPU] and EDX[CMOV] returned by CPUID Fn0000_0001 or Fn8000_0001.
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value, a QNaN value, or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.

FCOS Floating-Point Cosine

Computes the cosine of the radian value in ST(0) and stores the result in ST(0).

If the radian value lies outside the valid range of -2^{63} to $+2^{63}$ radians, the instruction sets the C2 flag in the x87 status word to 1 to indicate the value is out of range and does not change the value in ST(0).

Mnemonic	Opcode	Description
FCOS	D9 FF	Replace ST(0) with the cosine of ST(0).

Related Instructions

FPTAN, FPATAN, FSIN, FSINCOS

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	0	Source operand was in range.
	1	Source operand was out of range.
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

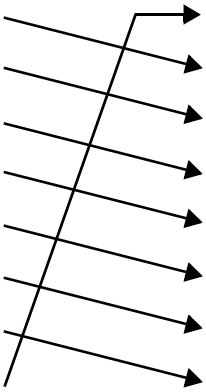
Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FDECSTP Floating-Point Decrement Stack-Top Pointer

Decrements the top-of-stack pointer (TOP) field of the x87 status word. If the TOP field contains 0, it is set to 7. In other words, this instruction rotates the stack by one position.

Mnemonic	Opcode	Description
FDECSTP	D9 F6	Decrement the TOP field in the x87 status word.

	Before FDECSTP			After FDECSTP	
Data Register	Value	Stack Pointer		Stack Pointer	Value
7	num1	ST(7)		ST(0)	num1
6	num2	ST(6)		ST(7)	num2
5	num3	ST(5)		ST(6)	num3
4	num4	ST(4)		ST(5)	num4
3	num5	ST(3)		ST(4)	num5
2	num6	ST(2)		ST(3)	num6
1	num7	ST(1)		ST(2)	num7
0	num8	ST(0)		ST(1)	num8

Related Instructions

FINCSTP

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

FDIV
FDIVP
FIDIV

Floating-Point Divide

Divides the value in a floating-point register by the value in another register or a memory location and stores the result in the register containing the dividend. For the FDIV and FDIVP instructions, the divisor value in memory can be stored in single-precision or double-precision floating-point format.

If only one operand is specified, the instruction divides the value in ST(0) by the value in the specified memory location.

If no operands are specified, the FDIVP instruction divides the value in ST(1) by the value in ST(0), stores the result in ST(1), and pops the x87 register stack.

The FIDIV instruction converts a divisor in word integer or short integer format to double-extended-precision floating-point format before performing the division. It treats an integer 0 as +0.

If the zero-divide exception is not masked (ZM bit cleared to 0 in the x87 control word) and the operation causes a zero-divide exception (sets the ZE bit in the x87 status word to 1), the operation stores no result. If the zero-divide exception is masked (ZM bit set to 1), a zero-divide exception causes ±∞ to be stored.

The sign of the operands, even if one of the operands is 0, determines the sign of the result.

Mnemonic	Opcode	Description
FDIV ST(0),ST(*i*)	D8 F0+*i*	Replace ST(0) with ST(0)/ST(*i*).
FDIV ST(*i*),ST(0)	DC F8+*i*	Replace ST(*i*) with ST(*i*)/ST(0).
FDIV *mem32real*	D8 /6	Replace ST(0) with ST(0)/*mem32real*.
FDIV *mem64real*	DC /6	Replace ST(0) with ST(0)/*mem64real*.
FDIVP	DE F9	Replace ST(1) with ST(1)/ST(0), and pop the x87 register stack.
FDIVP ST(*i*),ST(0)	DE F8+*i*	Replace ST(*i*) with ST(*i*)/ST(0), and pop the x87 register stack.
FIDIV *mem16int*	DE /6	Replace ST(0) with ST(0)/*mem16int*.
FIDIV *mem32int*	DA /6	Replace ST(0) with ST(0)/*mem32int*.

Related Instructions

FDIVR, FDIVRP, FIDIVR

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	±infinity was divided by ±infinity.
	X	X	X	±zero was divided by ±zero.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Zero-divide exception (ZE)	X	X	X	A non-zero value was divided by ±zero.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Overflow exception (OE)	X	X	X	A rounded result was too large to fit into the format of the destination operand.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FDIVR
FDIVRP
FIDIVR
Floating-Point Divide Reverse

Divides a value in a floating-point register or a memory location by the value in a floating-point register and stores the result in the register containing the divisor. For the FDIVR and FDIVRP instructions, a dividend value in memory can be stored in single-precision or double-precision floating-point format.

If one operand is specified, the instruction divides the value at the specified memory location by the value in ST(0). If two operands are specified, it divides the value in ST(0) by the value in another x87 stack register or vice versa.

The FIDIVR instruction converts a dividend in word integer or short integer format to double-extended-precision format before performing the division.

The FDIVRP instruction pops the x87 register stack after performing the division operation. If no operand is specified, the FDIVRP instruction divides the value in ST(0) by the value in ST(1).

If the zero-divide exception is not masked (ZM bit cleared to 0 in the x87 control word) and the operation causes a zero-divide exception (sets the ZE bit in the x87 status word to 1), the operation stores no result. If the zero-divide exception is masked (ZM bit set to 1), a zero-divide exception causes ±∞ to be stored.

The sign of the operands, even if one of the operands is 0, determines the sign of the result.

Mnemonic	Opcode	Description
FDIVR ST(0),ST(*i*)	D8 F8+*i*	Replace ST(0) with ST(*i*)/ST(0).
FDIVR ST(*i*), ST(0)	DC F0+*i*	Replace ST(*i*) with ST(0)/ST(*i*).
FDIVR *mem32real*	D8 /7	Replace ST(0) with *mem32real*/ST(0).
FDIVR *mem64real*	DC /7	Replace ST(0) with *mem64real*/ST(0).
FDIVRP	DE F1	Replace ST(1) with ST(0)/ST(1), and pop the x87 register stack.
FDIVRP ST(*i*), ST(0)	DE F0 +*i*	Replace ST(*i*) with ST(0)/ST(*i*), and pop the x87 register stack.
FIDIVR *mem16int*	DE /7	Replace ST(0) with *mem16int*/ST(0).
FIDIVR *mem32int*	DA /7	Replace ST(0) with *mem32int*/ST(0).

Related Instructions

FDIV, FDIVP, FIDIV

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or is non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or is non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	±infinity was divided by ±infinity.
	X	X	X	±zero was divided by ±zero.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Zero-divide exception (ZE)	X	X	X	A non-zero value was divided by ±zero.
Overflow exception (OE)	X	X	X	A rounded result was too large to fit into the format of the destination operand.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FFREE Floating-Point Free Register

Frees the specified x87 stack register by marking its tag register entry as empty. The instruction does not affect the contents of the freed register or the top-of-stack pointer (TOP).

Mnemonic	Opcode	Description
FFREE ST(*i*)	DD C0+*i*	Set the tag for x87 stack register *i* to empty (11b).

Related Instructions

FLD, FST, FSTP

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	U	
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

FICOM FICOMP Floating-Point Integer Compare

Converts a 16-bit or 32-bit signed integer value to double-extended-precision format, compares it to the value in ST(0), and sets the C0, C2, and C3 condition code flags in the x87 status word to reflect the results.

The comparison operation ignores the sign of zero (–0.0 = +0.0).

After performing the comparison operation, the FICOMP instruction pops the x87 register stack.

If ST(0) is a NaN or is in an unsupported format, the instruction sets the condition flags to "unordered."

Mnemonic	Opcode	Description
FICOM *mem16int*	DE /2	Convert the contents of *mem16int* to double-extended-precision format, compare the result to the contents of ST(0), and set condition flags to reflect the results of the comparison.
FICOM *mem32int*	DA /2	Convert the contents of *mem32int* to double-extended-precision format, compare the result to the contents of ST(0), and set condition flags to reflect the results of the comparison.
FICOMP *mem16int*	DE /3	Convert the contents of *mem16int* to double-extended-precision format, compare the result to the contents of ST(0), set condition flags to reflect the results of the comparison, and pop the x87 register stack.
FICOMP *mem32int*	DA /3	Convert the contents of *mem32int* to double-extended-precision format, compare the result to the contents of ST(0), set condition flags to reflect the results of the comparison, and pop the x87 register stack.

Related Instructions

FCOM, FCOMPP, FCOMI, FCOMIP, FTST, FUCOMI, FUCOMIP, FXAM

rFLAGS Affected

None

x87 Condition Code

C3	C2	C1	C0	Compare Result
0	0	0	0	ST(0) > source
0	0	0	1	ST(0) < source
1	0	0	0	ST(0) = source
1	1	0	1	Operands were unordered

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value, a QNaN value, or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.

FILD Floating-Point Load Integer

Converts a signed-integer in memory to double-extended-precision format and pushes the value onto the x87 register stack. The value can be a 16-bit, 32-bit, or 64- bit integer value. Signed values from memory can always be represented exactly in x87 registers without rounding.

Mnemonic	Opcode	Description
FILD *mem16int*	DF /0	Push the contents of *mem16int* onto the x87 register stack.
FILD *mem32int*	DB /0	Push the contents of *mem32int* onto the x87 register stack.
FILD *mem64int*	DF /5	Push the contents of *mem64int* onto the x87 register stack.

Related Instructions

FLD, FST, FSTP, FIST, FISTP, FBLD, FBSTP

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No stack overflow.
	1	x87 stack overflow, if an x87 register stack fault was detected.
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

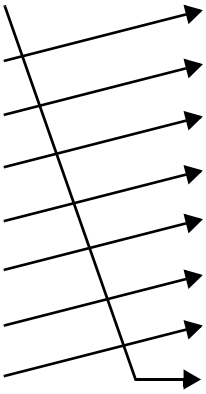
Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack overflow occurred.

FINCSTP Floating-Point Increment Stack-Top Pointer

Increments the top-of-stack pointer (TOP) field of the x87 status word. If the TOP field contains 7, it is cleared to 0. In other words, this instruction rotates the stack by one position.

Mnemonic	Opcode	Description
FINCSTP	D9 F7	Increment the TOP field in the x87 status word.

	Before FINCSTP			After FINCSTP	
Data Register	Value	Stack Pointer		Stack Pointer	Value
7	num1	ST(7)		ST(6)	num1
6	num2	ST(6)		ST(5)	num2
5	num3	ST(5)		ST(4)	num3
4	num4	ST(4)		ST(3)	num4
3	num5	ST(3)		ST(2)	num5
2	num6	ST(2)		ST(1)	num6
1	num7	ST(1)		ST(0)	num7
0	num8	ST(0)		ST(7)	num8

Related Instructions

FDECSTP

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

FINIT
FNINIT
Floating-Point Initialize

Sets the x87 control word register, status word register, tag word register, instruction pointer, and data pointer to their default states as follows:

- Sets the x87 control word to 037Fh—round to nearest (RC = 00b); double-extended-precision (PC = 11b); all exceptions masked (PM, UM, OM, ZM, DM, and IM all set to 1).
- Clears all bits in the x87 status word (TOP is set to 0, which maps ST(0) onto FPR0).
- Marks all x87 stack registers as empty (11b) in the x87 tag register.
- Clears the instruction pointer and the data pointer.

These instructions do not actually zero out the x87 stack registers.

Assemblers usually provide an FINIT macro that expands into the instruction sequence

```
WAIT                    ; Opcode 9B
FNINIT destination      ; Opcode DB E3
```

The WAIT (9Bh) instruction checks for pending x87 exceptions and calls an exception handler, if necessary. The FNINIT instruction then resets the x87 environment to its default state.

Mnemonic	Opcode	Description
FINIT	9B DB E3	Perform a WAIT (9B) to check for pending floating-point exceptions and then initialize the x87 unit.
FNINIT	DB E3	Initialize the x87 unit without checking for unmasked floating-point exceptions.

Related Instructions

FWAIT, WAIT

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	0	
C1	0	
C2	0	
C3	0	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.

FIST
FISTP
Floating-Point Integer Store

Converts the value in ST(0) to a signed integer, rounds it if necessary, and copies it to the specified memory location. The rounding control (RC) field of the x87 control word determines the type of rounding used.

The FIST instruction supports 16-bit and 32-bit values. The FISTP instructions supports 16-bit, 32-bit, and 64-bit values.

The FISTP instruction pops the stack after storing the rounded value in memory.

If the value is too large for the destination location, is a NaN, or is in an unsupported format, the instruction sets the invalid-operation exception (IE) bit in the x87 status word to 1. Then, if the exception is masked (IM bit set to 1 in the x87 control word), the instruction stores the integer indefinite value. If the exception is unmasked (IM bit cleared to 0), the instruction does not store the value.

Mnemonic	Opcode	Description
FIST *mem16int*	DF /2	Convert the contents of ST(0) to integer and store the result in *mem16int*.
FIST *mem32int*	DB /2	Convert the contents of ST(0) to integer and store the result in *mem32int*.
FISTP *mem16int*	DF /3	Convert the contents of ST(0) to integer, store the result in *mem16int*, and pop the x87 register stack.
FISTP *mem32int*	DB /3	Convert the contents of ST(0) to integer, store the result in *mem32int*, and pop the x87 register stack.
FISTP *mem64int*	DF /7	Convert the contents of ST(0) to integer, store the result in *mem64int*, and pop the x87 register stack.

Table 2-1 shows the results of storing various types of numbers as integers.

Table 2-1. Storing Numbers as Integers

ST(0)	DEST
-∞	Invalid-operation (IE) exception
–Finite-real < –1	–Integer (Invalid-operation (IE) exception if the integer is too large for the destination)
–1 < –Finite-real< –0	0 or –1, depending on the rounding mode
-0	0
+0	0
+0 < +Finite-real < +1	0 or +1, depending on the rounding mode

Table 2-1. Storing Numbers as Integers (continued)

ST(0)	DEST
+Finite-real > +1	+Integer (Invalid-operation (IE) exception if the integer is too large for the destination)
+∞	Invalid-operation (IE) exception
NaN	Invalid-operation (IE) exception

Related Instructions

FLD, FST, FSTP, FILD, FBLD, FBSTP, FISTTP

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a nonwritable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	The source operand was too large for the destination format.
	X	X	X	A source operand was an SNaN value, a QNaN value, +-infinity, or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FISTTP Floating Point Integer Truncate and Store

Converts a floating-point value in ST(0) to an integer by truncating the fractional part of the number and storing the integer result to the memory address specified by the destination operand. FISTTP then pops the floating point register stack. The FISTTP instruction ignores the rounding mode specified by the x87 control word.

The FISTTP instruction applies to 16-bit, 32-bit, and 64-bit operands.

The FISTTP instruction is an SSE3 instruction. Support for this instruction subset is indicated by CPUID Fn0000_0001_ECX[SSE3] = 1. See “CPUID” in Volume 3 for more information about the CPUID instruction.

Mnemonic	Opcode	Description
FISTTP *mem16int*	DF /1	Store the truncated floating-point value in ST(0) in memory location *mem16int* and pop the floating-point register stack.
FISTTP *mem32int*	DB /1	Store the truncated floating-point value in ST(0) in memory location *mem32int* and pop the floating-point register stack.
FISTTP *mem64int*	DD /1	Store the truncated floating-point value in ST(0) in memory location *mem64int* and pop the floating-point register stack.

Table 2-2 shows the results of storing various types of numbers as integers.

Table 2-2. Storing Numbers as Integers

ST(0)	DESTINATION
-∞	Invalid-operation (IE) exception
–Finite-real ≤ –1	–Integer (Invalid-operation (IE) exception if the integer is too large for the destination)
-1 < Finite-real < +1	0
+Finite-real ≥ +1	+Integer (Invalid-operation (IE) exception if the integer is too large for the destination)
+∞	Invalid-operation (IE) exception
NaN	Invalid-operation (IE) exception

Related Instructions

FLD, FST, FSTP, FILD, FBLD, FBSTP, FISTP

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value*	Description
C0	U	
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
		FP number is rounded down (always done since the instruction forces truncate mode).
C2	U	
C3	U	

Note: **A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
#UD	X	X	X	The SSE3 instructions are not supported, as indicated by CPUID Fn0000_0001_ECX[SSE3] = 0.
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a nonwritable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	The source operand was too large for the destination format.
	X	X	X	A source operand was an SNaN value, a QNaN value,+-infinity, or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FLD Floating-Point Load

Pushes a value in memory or in a floating-point register onto the register stack. If in memory, the value can be a single-precision, double-precision, or double-extended-precision floating-point value. The operation converts a single-precision or double-precision value to double-extended-precision format before pushing it onto the stack.

Mnemonic	Opcode	Description
FLD ST(*i*)	D9 C0+*i*	Push the contents of ST(*i*) onto the x87 register stack.
FLD *mem32real*	D9 /0	Push the contents of *mem32real* onto the x87 register stack.
FLD *mem64real*	DD /0	Push the contents of *mem64real* onto the x87 register stack.
FLD *mem80real*	DB /5	Push the contents of *mem80real* onto the x87 register stack.

Related Instructions

FFREE, FST, FSTP, FILD, FIST, FISTP, FBLD, FBSTP

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
	1	x87 stack overflow, if an x87 register stack fault was detected.
	0	No x87 stack fault.
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value. This exception does not occur if the source operand was in double-extended-precision format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
	X	X	X	An x87 stack overflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value. This exception does not occur if the source operand was in double-extended-precision format.

FLD1 Floating-Point Load +1.0

Pushes the floating-point value +1.0 onto the register stack.

Mnemonic	Opcode	Description
FLD1	D9 E8	Push +1.0 onto the x87 register stack.

Related Instructions

FLD, FLDZ, FLDPI, FLDL2T, FLDL2E, FLDLG2, FLDLN2

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No x87 stack fault occurred.
	1	x87 stack overflow, if an x87 register stack fault was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack overflow occurred.

FLDCW Floating-Point Load x87 Control Word

Loads a 16-bit value from the specified memory location into the x87 control word. If the new x87 control word unmasks any pending floating point exceptions, then they are handled upon execution of the next x87 floating-point or 64-bit media instruction.

To avoid generating exceptions when loading a new control word, use the FCLEX or FNCLEX instruction to clear any pending exceptions.

Mnemonic	Opcode	Description
FLDCW *mem2env*	D9 /5	Load the contents of *mem2env* into the x87 control word.

Related Instructions

FSTCW, FNSTCW, FSTSW, FNSTSW, FSTENV, FNSTENV, FLDENV, FCLEX, FNCLEX

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	U	
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

FLDENV Floating-Point Load x87 Environment

Restores the x87 environment from memory starting at the specified address. The x87 environment consists of the x87 control, status, and tag word registers, the last non-control x87 instruction pointer, the last x87 data pointer, and the opcode of the last completed non-control x87 instruction.

The FLDENV instruction takes a memory operand that specifies the starting address of either a 14-byte or 28-byte area in memory. The 14-byte operand is required for a 16-bit operand-size; the 28-byte memory area is required for both 32-bit and 64-bit operand sizes. The layout of the saved x87 environment within the specified memory area depends on whether the processor is operating in protected or real mode. See “Media and x87 Processor State” in Volume 2 for details on how this instruction loads the x87 environment from memory. (Because FSTENV does not save the full 64-bit data and instruction pointers, 64-bit applications should use FXSAVE/FXRSTOR, rather than FSTENV/FLDENV.)

The environment to be loaded is typically stored by a previous FNSTENV or FSTENV instruction. The FLDENV instruction should be executed in the same operating mode as the instruction that stored the x87 environment.

If FLDENV results in set exception flags in the loaded x87 status word register, and these exceptions are unmasked in the x87 control word register, a floating-point exception occurs when the next floating-point instruction is executed (except for the no-wait floating-point instructions).

To avoid generating exceptions when loading a new environment, use the FCLEX or FNCLEX instruction to clear the exception flags in the x87 status word before storing that environment.

Mnemonic	Opcode	Description
FLDENV *mem14/28env*	D9 /4	Load the complete contents of the x87 environment from *mem14/28env*.

Related Instructions

FSTENV, FNSTENV, FCLEX, FNCLEX

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	M	Loaded from memory.
C1	M	Loaded from memory.
C2	M	Loaded from memory.
C3	M	Loaded from memory.
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

FLDL2E Floating-Point Load Log_2 e

Pushes log_2e onto the x87 register stack. The value in ST(0) is the result, in double-extended-precision format, of rounding an internal 66-bit constant according to the setting of the RC field in the x87 control word register.

Mnemonic	Opcode	Description
FLDL2E	D9 EA	Push log_2e onto the x87 register stack.

Related Instructions

FLD, FLD1, FLDZ, FLDPI, FLDL2T, FLDLG2, FLDLN2

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No x87 stack fault occurred.
	1	x87 stack overflow, if an x87 register stack fault was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack overflow occurred.

FLDL2T Floating-Point Load Log_2 10

Pushes log_2 10 onto the x87 register stack. The value in ST(0) is the result, in double-extended-precision format, of rounding an internal 66-bit constant according to the setting of the RC field in the x87 control word register.

Mnemonic	Opcode	Description
FLDL2T	D9 E9	Push $log_2$10 onto the x87 register stack.

Related Instructions

FLD, FLD1, FLDZ, FLDPI, FLDL2E, FLDLG2, FLDLN2

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No x87 stack fault occurred.
	1	x87 stack overflow, if an x87 register stack fault was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack overflow occurred.

FLDLG2 Floating-Point Load Log_{10} 2

Pushes $\log_{10}$ 2 onto the x87 register stack. The value in ST(0) is the result, in double-extended-precision format, of rounding an internal 66-bit constant according to the setting of the RC field in the x87 control word register.

Mnemonic	Opcode	Description
FLDLG2	D9 EC	Push $\log_{10}$2 onto the x87 register stack.

Related Instructions

FLD, FLD1, FLDZ, FLDPI, FLDL2T, FLDL2E, FLDLN2

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No x87 stack fault occurred.
	1	x87 stack overflow, if an x87 register stack fault was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack overflow occurred.

FLDLN2 Floating-Point Load Ln 2

Pushes $\log_e 2$ onto the x87 register stack. The value in ST(0) is the result, in double-extended-precision format, of rounding an internal 66-bit constant according to the setting of the RC field in the x87 control word register.

Mnemonic	Opcode	Description
FLDLN2	D9 ED	Push $\log_e 2$ onto the x87 register stack.

Related Instructions

FLD, FLD1, FLDZ, FLDPI, FLDL2T, FLDL2E, FLDLG2

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No x87 stack fault occurred.
	1	x87 stack overflow, if an x87 register stack fault was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack overflow occurred.

FLDPI Floating-Point Load Pi

Pushes π onto the x87 register stack. The value in ST(0) is the result, in double-extended-precision format, of rounding an internal 66-bit constant according to the setting of the RC field in the x87 control word register.

Mnemonic	Opcode	Description
FLDPI	D9 EB	Push π onto the x87 register stack.

Related Instructions

FLD, FLD1, FLDZ, FLDL2T, FLDL2E, FLDLG2, FLDLN2

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No x87 stack fault occurred.
	1	x87 stack overflow, if an x87 register stack fault was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack overflow occurred.

FLDZ Floating-Point Load +0.0

Pushes +0.0 onto the x87 register stack.

Mnemonic	Opcode	Description
FLDZ	D9 EE	Push zero onto the x87 register stack.

Related Instructions

FLD, FLD1, FLDPI, FLDL2T, FLDL2E, FLDLG2, FLDLN2

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No x87 stack fault occurred.
	1	x87 stack overflow, if an x87 register stack fault was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack overflow occurred.

FMUL
FMULP
FIMUL

Floating-Point Multiply

Multiplies the value in a floating-point register by the value in a memory location or another stack register and stores the result in the first register. The instruction converts a single-precision or double-precision value in memory to double-extended-precision format before multiplying.

If one operand is specified, the instruction multiplies the value in the ST(0) register by the value in the specified memory location and stores the result in the ST(0) register.

If two operands are specified, the instruction multiplies the value in the ST(0) register by the value in another specified floating-point register and stores the result in the register specified in the first operand.

The FMULP instruction pops the x87 stack after storing the product. The no-operand version of the FMULP instruction multiplies the value in the ST(1) register by the value in the ST(0) register and stores the product in the ST(1) register.

The FIMUL instruction converts a short-integer or word-integer value in memory to double-extended-precision format, multiplies it by the value in ST(0), and stores the product in ST(0).

Mnemonic	Opcode	Description
FMUL ST(0),ST(*i*)	D8 C8+*i*	Replace ST(0) with ST(0) ∗ ST(*i*).
FMUL ST(*i*),ST(0)	DC C8+*i*	Replace ST(*i*) with ST(0) ∗ ST(*i*).
FMUL *mem32real*	D8 /1	Replace ST(0) with *mem32real* ∗ ST(0).
FMUL *mem64real*	DC /1	Replace ST(0) with *mem64real* ∗ ST(0).
FMULP	DE C9	Replace ST(1) with ST(0) ∗ ST(1), and pop the x87 register stack.
FMULP ST(*i*),ST(0)	DE C8+*i*	Replace ST(*i*) with ST(0) ∗ ST(*i*), and pop the x87 register stack.
FIMUL *mem16int*	DE /1	Replace ST(0) with *mem16int* ∗ ST(0).
FIMUL *mem32int*	DA /1	Replace ST(0) with *mem32int* ∗ ST(0).

Related Instructions

None

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	±infinity was multiplied by ±zero.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Overflow exception (OE)	X	X	X	A rounded result was too large to fit into the format of the destination operand.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FNOP Floating-Point No Operation

Performs no operation. This instruction affects only the rIP register. It does not otherwise affect the processor context.

Mnemonic	Opcode	Description
FNOP	D9 D0	Perform no operation.

Related Instructions

FWAIT, NOP

rFLAGS Affected

None

x87 Condition Code

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

FPATAN Floating-Point Partial Arctangent

Computes the arctangent of the ordinate (Y) in ST(1) divided by the abscissa (X) in ST(0), which is the angle in radians between the X axis and the radius vector from the origin to the point (X, Y). It then stores the result in ST(1) and pops the x87 register stack. The resulting value has the same sign as the ordinate value and a magnitude less than or equal to π.

There is no restriction on the range of values that FPATAN can accept. Table 2-3 shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur:

Table 2-3. Computing Arctangent of Numbers

		X (ST(0))						
		–∞	**–Finite**	**–0**	**+0**	**+Finite**	**+∞**	**NaN**
Y (ST(1))	**–∞**	–3π/4	–π/2	–π/2	–π/2	–π/2	–π/4	NaN
	–Finite	–π	–π to –π/2	–π/2	–π/2	–π/2 to –0	—0	NaN
	–0	–π	–π	–π	–0	–0	—0	NaN
	+0	+π	+π	+π	+0	+0	+0	NaN
	+Finite	+π	+π to +π/2	+π/2	+π/2	+π/2 to +0	+0	NaN
	+∞	+3π/4	+π/2	+π/2	+π/2	+π/2	+π/4	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Mnemonic	Opcode	Description
FPATAN	D9 F3	Compute arctan(ST(1)/ST(0)), store the result in ST(1), and pop the x87 register stack.

Related Instructions

FCOS, FPTAN, FSIN, FSINCOS

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FPREM Floating-Point Partial Remainder

Computes the exact remainder obtained by dividing the value in ST(0) by that in ST(1), and stores the result in ST(0). It computes the remainder by an iterative subtract-and-shift long division algorithm in which one quotient bit is calculated in each iteration.

If the exponent difference between ST(0) and ST(1) is less than 64, the instruction computes all integer bits of the quotient, guaranteeing that the remainder is less in magnitude than the divisor in ST(1). If the exponent difference is equal to or greater than 64, it computes only the subset of integer quotient bits numbering between 32 and 63, returns a partial remainder, and sets the C2 condition code bit to 1.

FPREM is supported for software that was written for early x87 coprocessors. Unlike the FPREM1 instruction, FPREM does not compute the partial remainder as specified in IEEE Standard 754.

Mnemonic	Opcode	Description
FPREM	D9 F8	Compute the remainder of the division of ST(0) by ST(1) and store the result in ST(0).

Action

```
ExpDiff = Exponent(ST(0)) - Exponent(ST(1))
IF (ExpDiff < 0)
{
   SW.C2 = 0
   {SW.C0, SW.C3, SW.C1} = 0
}
ELSIF (ExpDiff < 64)
{
   Quotient = Truncate(ST(0)/ST(1))
   ST(0) = ST(0) - (ST(1) * Quotient)
   SW.C2 = 0
   {SW.C0, SW.C3, SW.C1} = Quotient mod 8
}
ELSE
{
   N = 32 + (ExpDiff mod 32)
   Quotient = Truncate ((ST(0)/ST(1))/2^(ExpDiff-N))
   ST(0) = ST(0) - (ST(1) * Quotient * 2^(ExpDiff-N))
   SW.C2 = 1
   {SW.C0, SW.C3, SW.C1} = 0
}
```

Related Instructions

FPREM1, FABS, FRNDINT, FXTRACT, FCHS

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	M	Set equal to the value of bit 2 of the quotient.
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
	M	Set equal to the value of bit 0 of the quotient, if there was no fault.
C2	0	FPREM generated the partial remainder.
	1	The source operands differed by more than a factor of 2^{64}, so the result is incomplete.
C3	M	Set equal to the value of bit 1 of the quotient.

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	ST(0) was ±infinity.
	X	X	X	ST(1) was ±zero.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.

FPREM1 Floating-Point Partial Remainder

Computes the IEEE Standard 754 remainder obtained by dividing the value in ST(0) by that in ST(1), and stores the result in ST(0). Unlike FPREM, it rounds the integer quotient to the nearest even integer and returns the remainder corresponding to the back multiply of the rounded quotient.

If the exponent difference between ST(0) and ST(1) is less than 64, the instruction computes all integer as well as additional fractional bits of the quotient to do the rounding. The remainder returned is a complete remainder and is less than or equal to one half of the magnitude of the divisor. If the exponent difference is equal to or greater than 64, it computes only the subset of integer quotient bits numbering between 32 and 63, returns the partial remainder, and sets the C2 condition code bit to 1.

Rounding control has no effect. FPREM1 results are exact.

Mnemonic	Opcode	Description
FPREM1	D9 F5	Compute the IEEE standard 754 remainder of the division of ST(0) by ST(1) and store the result in ST(0).

Action

```
ExpDiff = Exponent(ST(0)) - Exponent(ST(1))
IF (ExpDiff < 0)
{
   SW.C2 = 0
   {SW.C0, SW.C3, SW.C1} = 0
}
ELSIF (ExpDiff < 64)
{
   Quotient = Integer obtained by rounding (ST(0)/ST(1))
          to nearest even integer
   ST(0) = ST(0) - (ST(1) * Quotient)
   SW.C2 = 0
   {SW.C0, SW.C3, SW.C1} = Quotient mod 8
}
ELSE
{
   N = 32 + (ExpDiff mod 32)
   Quotient = Truncate ((ST(0)/ST(1))/2^(ExpDiff-N))
   ST(0) = ST(0) - (ST(1) * Quotient * 2^(ExpDiff-N))
   SW.C2 = 1
   {SW.C0, SW.C3, SW.C1} = 0
}
```

Related Instructions

FPREM, FABS, FRNDINT, FXTRACT, FCHS

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	M	Set equal to the value of bit 2 of the quotient.
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
	M	Set equal to the value of the bit 0 of the quotient, if there was no fault.
C2	0	FPREM1 generated the partial remainder.
	1	The source operands differed by more than a factor of 2^{64}, so the result is incomplete.
C3	M	Set equal to the value of bit 1 of the quotient.
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	ST(0) was ±infinity.
	X	X	X	ST(1) was ±zero.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.

FPTAN Floating-Point Partial Tangent

Computes the tangent of the radian value in ST(0), stores the result in ST(0), and pushes a value of 1.0 onto the x87 register stack.

The source value must be between -2^{63} and $+2^{63}$ radians. If the source value lies outside the specified range, the instruction sets the C2 bit of the x87 status word to 1 and does not change the value in ST(0).

Mnemonic	Opcode	Description
FPTAN	D9 F2	Replace ST(0) with the tangent of ST(0), then push 1.0 onto the x87 register stack.

Related Instructions

FCOS, FPATAN, FSIN, FSINCOS

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
	1	x87 stack overflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	0	Source operand was in range.
	1	Source operand was out of range.
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	A source operand was ±infinity
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
	X	X	X	An x87 stack overflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Overflow exception (OE)	X	X	X	A rounded result was too large to fit into the format of the destination operand.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FRNDINT Floating-Point Round to Integer

Rounds the value in ST(0) to an integer, depending on the setting of the rounding control (RC) field of the x87 control word, and stores the result in ST(0).

If the initial value in ST(0) is ∞, the instruction does not change ST(0). If the value in ST(0) is not an integer, it sets the precision exception (PE) bit of the x87 status word to 1.

Mnemonic	Opcode	Description
FRNDINT	D9 FC	Round the contents of ST(0) to an integer.

Related Instructions

FABS, FPREM, FXTRACT, FCHS

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Precision exception (PE)	X	X	X	The source operand was not an integral value.

FRSTOR Floating-Point Restore x87 and MMX™ State

Restores the complete x87 state from memory starting at the specified address, as stored by a previous call to F(N)SAVE.

The FRSTOR instruction takes a memory operand that specifies the starting address of either a 94-byte or 108-byte area in memory. The 94-byte operand is required for a 16-bit operand-size; the 108-byte memory area is required for both 32-bit and 64-bit operand sizes. The layout of the saved x87 state within the specified memory area depends on whether the processor is operating in protected or real mode. See “Media and x87 Processor State” in Volume 2 for details on how this instruction stores the x87 environment in memory. (Because FSAVE does not save the full 64-bit data and instruction pointers, 64-bit applications should use FXSAVE/FXRSTOR, rather than FSAVE/FRSTOR.)

Because the MMX registers are mapped onto the low 64 bits of the x87 floating-point registers, this operation also restores the MMX state.

If FRSTOR results in set exception flags in the loaded x87 status word register, and these exceptions are unmasked in the x87 control word register, a floating-point exception occurs when the next floating-point instruction is executed (except for the no-wait floating-point instructions).

To avoid generating exceptions when loading a new environment, use the FCLEX or FNCLEX instruction to clear the exception flags in the x87 status word before storing that environment.

For details about the memory image restored by FRSTOR, see “Media and x87 Processor State” in Volume 2.

Mnemonic	Opcode	Description
FRSTOR *mem94/108env*	DD /4	Load the x87 state from *mem94/108env*.

Related Instructions

FSAVE, FNSAVE, FXSAVE, FXRSTOR

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	M	Loaded from memory.
C1	M	Loaded from memory.
C2	M	Loaded from memory.
C3	M	Loaded from memory.

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

FSAVE FNSAVE Floating-Point Save x87 and MMX™ State

Stores the complete x87 state to memory starting at the specified address and reinitializes the x87 state.

The FSAVE instruction takes a memory operand that specifies the starting address of either a 94-byte or 108-byte area in memory. The 94-byte operand is required for a 16-bit operand-size; the 108-byte memory area is required for both 32-bit and 64-bit operand sizes. The layout of the saved x87 state within the specified memory area depends on whether the processor is operating in protected or real mode. See “Media and x87 Processor State” in Volume 2 for details on how this instruction stores the x87 environment in memory. (Because FSAVE does not save the full 64-bit data and instruction pointers, 64-bit applications should use FXSAVE/FXRSTOR, rather than FSAVE/FRSTOR.)

Because the MMX registers are mapped onto the low 64 bits of the x87 floating-point registers, this operation also saves the MMX state.

The FNSAVE instruction does not wait for pending unmasked x87 floating-point exceptions to be processed.

Assemblers usually provide an FSAVE macro that expands into the instruction sequence

```
WAIT                    ; Opcode 9B
FNSAVE destination      ; Opcode DD /6
```

The WAIT (9Bh) instruction checks for pending x87 exceptions and calls an exception handler, if necessary. The FNSAVE instruction then stores the x87 state to the specified destination.

Mnemonic	Opcode	Description
FSAVE *mem94/108env*	9B DD /6	Copy the x87 state to *mem94/108env* after checking for pending floating-point exceptions, then reinitialize the x87 state.
FNSAVE *mem94/108env*	DD /6	Copy the x87 state to *mem94/108env* without checking for pending floating-point exceptions, then reinitialize the x87 state.

Related Instructions

FRSTOR, FXSAVE, FXRSTOR

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	0	
C1	0	
C2	0	
C3	0	

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a nonwritable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

FSCALE Floating-Point Scale

Multiplies the floating-point value in ST(0) by 2 to the power of the integer portion of the floating-point value in ST(1).

This instruction provides an efficient method of multiplying (or dividing) by integral powers of 2 because, typically, it simply adds the integer value to the exponent of the value in ST(0), leaving the significand unaffected. However, if the value in ST(0) is a denormal value, the mantissa is also modified and the result may end up being a normalized number. Likewise, if overflow or underflow results from a scale operation, the mantissa of the resulting value will be different from that of the source.

The FSCALE instruction performs the reverse operation to that of the FXTRACT instruction.

Mnemonic	Opcode	Description
FSCALE	D9 FD	Replace ST(0) with ST(0) $*$ $2^{rndint(ST(1))}$

Related Instructions

FSQRT, FPREM, FPREM1, FRNDINT, FXTRACT, FABS, FCHS

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	Undefined.
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	Undefined.
C3	U	Undefined
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Overflow exception (OE)	X	X	X	A rounded result was too large to fit into the format of the destination operand.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FSIN Floating-Point Sine

Computes the sine of the radian value in ST(0) and stores the result in ST(0).

The source value must be in the range -2^{63} to $+2^{63}$ radians. If the value lies outside this range, the instruction sets the C2 bit in the x87 status word to 1 and does not change the value in ST(0).

Mnemonic	Opcode	Description
FSIN	D9 FE	Replace ST(0) with the sine of ST(0).

Related Instructions

FCOS, FPATAN, FPTAN, FSINCOS

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	0	Source operand was in range.
	1	Source operand was out of range.
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	A source operand was ±infinity.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FSINCOS Floating-Point Sine and Cosine

Computes the sine and cosine of the value in ST(0), stores the sine in ST(0), and pushes the cosine onto the x87 register stack. The source value must be in the range -2^{63} to $+2^{63}$ radians.

If the source operand is outside this range, the instruction sets the C2 bit in the x87 status word to 1 and does not change the value in ST(0).

Mnemonic	Opcode	Description
FSINCOS	D9 FB	Replace ST(0) with the sine of ST(0), then push the cosine of ST(0) onto the x87 register stack.

Related Instructions

FCOS, FPATAN, FPTAN, FSIN

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	1	x87 stack overflow, if an x87 register stack fault was detected.
	0	Result in ST(1) was rounded down, if a precision exception was detected.
	1	Result in ST(1) was rounded up, if a precision exception was detected.
C2	0	Source operand was in range.
	1	Source operand was out of range.
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	A source operand was ±infinity.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
	X	X	X	An x87 stack overflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FSQRT Floating-Point Square Root

Computes the square root of the value in ST(0) and stores the result in ST(0). Taking the square root of +infinity returns +infinity.

Mnemonic	Opcode	Description
FSQRT	D9 FA	Replace ST(0) with the square root of ST(0).

Related Instructions

FSCALE, FPREM, FPREM1, FRNDINT, FXTRACT, FABS, FCHS

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	A source operand was a negative value (not including -zero).
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FST
FSTP
Floating-Point Store Stack Top

Copies the value in ST(0) to the specified floating-point register or memory location.

The FSTP instruction pops the x87 stack after copying the value. The instruction FSTP ST(0) is the same as popping the stack with no data transfer.

If the specified destination is a single-precision or double-precision memory location, the instruction converts the value to the appropriate precision format. It does this by rounding the significand of the source value as specified by the rounding mode determined by the RC field of the x87 control word and then converting to the format of destination. It also converts the exponent to the width and bias of the destination format.

If the value is too large for the destination format, the instruction sets the overflow exception (OE) bit of the x87 status word. Then, if the overflow exception is unmasked (OM bit cleared to 0 in the x87 control word), the instruction does not perform the store.

If the value is a denormal value, the instruction sets the underflow exception (UE) bit in the x87 status word.

If the value is ±0, ±∞, or a NaN, the instruction truncates the least significant bits of the significand and exponent to fit the destination location.

Mnemonic	Opcode	Description
FST ST(*i*)	DD D0+*i*	Copy the contents of ST(0) to ST(*i*).
FST *mem32real*	D9 /2	Copy the contents of ST(0) to *mem32real*.
FST *mem64real*	DD /2	Copy the contents of ST(0) to *mem64real.*
FSTP ST(*i*)	DD D8+*i*	Copy the contents of ST(0) to ST(*i*) and pop the x87 register stack.
FSTP *mem32real*	D9 /3	Copy the contents of ST(0) to *mem32real* and pop the x87 register stack
FSTP *mem64real*	DD /3	Copy the contents of ST(0) to *mem64real* and pop the x87 register stack.
FSTP *mem80real*	DB /7	Copy the contents of ST(0) to *mem80real* and pop the x87 register stack.

Related Instructions

FFREE, FLD, FILD, FIST, FISTP, FBLD, FBSTP

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
	1	x87 stack overflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a nonwritable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
	X	X	X	An x87 stack overflow occurred.
Overflow exception (OE)	X	X	X	A rounded result was too large to fit into the format of the destination operand.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FSTCW (FNSTCW) Floating-Point Store Control Word

Stores the x87 control word in the specified 2-byte memory location. The FNSTCW instruction does not check for possible floating-point exceptions before copying the image of the x87 status register.

Assemblers usually provide an FSTCW macro that expands into the instruction sequence:

```
WAIT                    ; Opcode 9B
FNSTCW destination      ; Opcode D9 /7
```

The WAIT (9Bh) instruction checks for pending x87 exception and calls an exception handler, if necessary. The FNSTCW instruction then stores the state of the x87 control register to the desired destination.

Mnemonic	Opcode	Description
FSTCW *mem2env*	9B D9 /7	Perform a WAIT (9B) to check for pending floating-point exceptions, then copy the x87 control word to *mem2env*.
FNSTCW *mem2env*	D9 /7	Copy the x87 control word to *mem2env* without checking for floating-point exceptions.

Related Instructions

FSTSW, FNSTSW, FSTENV, FNSTENV

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	U	
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a nonwritable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

FSTENV (FNSTENV) Floating-Point Store Environment

Stores the current x87 environment to memory starting at the specified address, and then masks all floating-point exceptions. The x87 environment consists of the x87 control, status, and tag word registers, the last non-control x87 instruction pointer, the last x87 data pointer, and the opcode of the last completed non-control x87 instruction.

The FSTENV instruction takes a memory operand that specifies the start of either a 14-byte or 28-byte area in memory. The 14-byte operand is required for a 16-bit operand-size; the 28-byte memory area is required for both 32-bit and 64-bit operand sizes. The layout of the saved x87 environment within the specified memory area depends on whether the processor is operating in protected or real mode. See "Media and x87 Processor State" in Volume 2 for details on how this instruction stores the x87 environment in memory. (Because FLDENV/FSTENV do not save the full 64-bit data and instruction pointers, 64-bit applications should use FXSAVE/FXRSTOR, rather than FLDENV/FSTENV.)

The FNSTENV instruction does not check for possible floating-point exceptions before storing the environment.

Assemblers usually provide an FSTENV macro that expands into the instruction sequence

```
WAIT                    ; Opcode 9B
FNSTENV destination     ; Opcode D9 /6
```

The WAIT (9Bh) instruction checks for pending x87 exceptions and calls an exception handler if necessary. The FNSTENV instruction then stores the state of the x87 environment to the specified destination.

Exception handlers often use these instructions because they provide access to the x87 instruction and data pointers. An exception handler typically saves the environment on the stack. The instructions mask all floating-point exceptions after saving the environment to prevent those exceptions from interrupting the exception handler.

Mnemonic	Opcode	Description
FSTENV *mem14/28env*	9B D9 /6	Perform a WAIT (9B) to check for pending floating-point exceptions, then copy the x87 environment to *mem14/28env* and mask the floating-point exceptions.
FNSTENV *mem14/28env*	D9 /6	Copy the x87 environment to *mem14/28env* without checking for pending floating-point exceptions, and mask the exceptions.

Related Instructions

FLDENV, FSTSW, FNSTSW, FSTCW, FNSTCW

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	U	
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a nonwritable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

FSTSW (FNSTSW) Floating-Point Store Status Word

Stores the current state of the x87 status word register in either the AX register or a specified two-byte memory location. The image of the status word placed in the AX register always reflects the result after the execution of the previous x87 instruction.

The AX form of the instruction is useful for performing conditional branching operations based on the values of x87 condition flags.

The FNSTSW instruction does not check for possible floating-point exceptions before storing the x87 status word.

Assemblers usually provide an FSTSW macro that expands into the instruction sequence:

```
WAIT                    ; Opcode 9B
FNSTSW destination      ; Opcode DD /7 or DF E0
```

The WAIT (9Bh) instruction checks for pending x87 exceptions and calls an exception handler if necessary. The FNSTSW instruction then stores the state of the x87 status register to the desired destination.

Mnemonic	Opcode	Description
FSTSW AX	9B DF E0	Perform a WAIT (9B) to check for pending floating-point exceptions, then copy the x87 status word to the AX register.
FSTSW *mem2env*	9B DD /7	Perform a WAIT (9B) to check for pending floating-point exceptions, then copy the x87 status word to *mem12byte*.
FNSTSW AX	DF E0	Copy the x87 status word to the AX register without checking for pending floating-point exceptions.
FNSTSW *mem2env*	DD /7	Copy the x87 status word to *mem12byte* without checking for pending floating-point exceptions.

Related Instructions

FSTCW, FNSTCW, FSTENV, FNSTENV

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	U	
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a nonwritable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

FSUB
FSUBP
FISUB
Floating-Point Subtract

Subtracts the value in a floating-point register or memory location from the value in another register and stores the result in that register.

If no operands are specified, the instruction subtracts the value in ST(0) from that in ST(1) and stores the result in ST(1).

If one operand is specified, it subtracts a floating-point or integer value in memory from the contents of ST(0) and stores the result in ST(0).

If two operands are specified, it subtracts the value in ST(0) from the value in another floating-point register or vice versa.

The FSUBP instruction pops the x87 register stack after performing the subtraction.

The no-operand version of the instruction always pops the register stack. In some assemblers, the mnemonic for this instruction is FSUB rather than FSUBP.

The FISUB instruction converts a signed integer value to double-extended-precision format before performing the subtraction.

Mnemonic	Opcode	Description
FSUB ST(0),ST(*i*)	D8 E0+*i*	Replace ST(0) with ST(0) – ST(*i*).
FSUB ST(*i*),ST(0)	DC E8+*i*	Replace ST(*i*) with ST(*i*) – ST(0).
FSUB *mem32real*	D8 /4	Replace ST(0) with ST(0) – *mem32real*.
FSUB *mem64real*	DC /4	Replace ST(0) with ST(0) – *mem64real*.
FSUBP	DE E9	Replace ST(1) with ST(1) – ST(0) and pop the x87 register stack.
FSUBP ST(*i*),ST(0)	DE E8+*i*	Replace ST(*i*) with ST(*i*) – ST(0), and pop the x87 register stack.
FISUB *mem16int*	DE /4	Replace ST(0) with ST(0) – *mem16int*.
FISUB *mem32int*	DA /4	Replace ST(0) with ST(0) – *mem32int*.

Related Instructions

FSUBRP, FISUBR, FSUBR

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	+infinity was subtracted from +infinity.
	X	X	X	–infinity was subtracted from –infinity.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Overflow exception (OE)	X	X	X	A rounded result was too large to fit into the format of the destination operand.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FSUBR
FSUBRP
FISUBR

Floating-Point Subtract Reverse

Subtracts the value in a floating-point register from the value in another register or a memory location, and stores the result in the first specified register. Values in memory can be in single-precision or double-precision floating-point, word integer, or short integer format.

If one operand is specified, the instruction subtracts the value in ST(0) from the value in memory and stores the result in ST(0).

If two operands are specified, it subtracts the value in ST(0) from the value in another floating-point register or vice versa.

The FSUBRP instruction pops the x87 register stack after performing the subtraction.

The no-operand version of the instruction always pops the register stack. In some assemblers, the mnemonic for this instruction is FSUBR rather than FSUBRP.

The FISUBR instruction converts a signed integer operand to double-extended-precision format before performing the subtraction.

The FSUBR instructions perform the reverse operations of the FSUB instructions.

Mnemonic	Opcode	Description
FSUBR ST(0),ST(*i*)	D8 E8+*i*	Replace ST(0) with ST(*i*) - ST(0).
FSUBR ST(*i*),ST(0)	DC E0+*i*	Replace ST(*i*) with ST(0) - ST(*i*).
FSUBR *mem32real*	D8 /5	Replace ST(0) with *mem32real* - ST(0).
FSUBR *mem64real*	DC /5	Replace ST(0) with *mem64real* - ST(0).
FSUBRP	DE E1	Replace ST(1) with ST(0) - ST(1) and pop x87 stack.
FSUBRP ST(*i*),ST(0)	DE E0+*i*	Replace ST(*i*) with ST(0) - ST(*i*) and pop x87 stack.
FISUBR *mem16int*	DE /5	Replace ST(0) with *mem16int* - ST(0).
FISUBR *mem32int*	DA /5	Replace ST(0) with *mem32int* - ST(0).

Related Instructions

FSUB, FSUBP, FISUB

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	+infinity was subtracted from +infinity.
	X	X	X	–infinity was subtracted from –infinity.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Overflow exception (OE)	X	X	X	A rounded result was too large to fit into the format of the destination operand.

Exception	Real	Virtual 8086	Protected	Cause of Exception
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FTST Floating-Point Test with Zero

Compares the value in ST(0) with 0.0, and sets the condition code flags in the x87 status word as shown in the x87 Condition Code table below. The instruction ignores the sign distinction between –0.0 and +0.0.

Mnemonic	Opcode	Description
FTST	D9 E4	Compare ST(0) to 0.0.

Related Instructions

FCOM, FCOMP, FCOMPP, FCOMI, FCOMIP, FICOM, FICOMP, FUCOMI, FUCOMIP, FUCOM, FUCOMP, FUCOMPP, FXAM

rFLAGS Affected

None

x87 Condition Code

C3	C2	C1	C0	Compare Result
0	0	0	0	ST(0) > 0.0
0	0	0	1	ST(0) < 0.0
1	0	0	0	ST(0) = 0.0
1	1	0	1	ST(0) was unordered

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was a SNaN value, a QNaN value, or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.

FUCOM
FUCOMP
FUCOMPP
Floating-Point Unordered Compare

Compares the value in ST(0) to the value in another x87 register, and sets the condition codes in the x87 status word as shown in the x87 Condition Code table below.

If no source operand is specified, the instruction compares the value in ST(0) to that in ST(1).

After making the comparison, the FUCOMP instruction pops the x87 stack register and the FUCOMPP instruction pops the x87 stack register twice.

The instruction carries out the same comparison operation as the FCOM instructions, but sets the invalid-operation exception (IE) bit in the x87 status word to 1 when either or both operands are an SNaN or are in an unsupported format. If either or both operands is a QNaN, it sets the condition code flags to unordered, but does not set the IE bit. The FCOM instructions, on the other hand, raise an IE exception when either or both of the operands are a NaN value or are in an unsupported format.

Support for the FCOM(P(P)) instruction can be determined by executing either CPUID function 0000_0001h or CPUID function 8000_0001. Support is indicated when both the EDX[FPU] (bit 0) and EDX[CMOV] (bit 15) feature flags are set.

Mnemonic	Opcode	Description
FUCOM	DD E1	Compare ST(0) to ST(1) and set condition code flags to reflect the results of the comparison.
FUCOM ST(*i*)	DD E0+*i*	Compare ST(0) to ST(*i*) and set condition code flags to reflect the results of the comparison.
FUCOMP	DD E9	Compare ST(0) to ST(1), set condition code flags to reflect the results of the comparison, and pop the x87 register stack.
FUCOMP ST(*i*)	DD E8+*i*	Compare ST(0) to ST(*i*), set condition code flags to reflect the results of the comparison, and pop the x87 register stack.
FUCOMPP	DA E9	Compare ST(0) to ST(1), set condition code flags to reflect the results of the comparison, and pop the x87 register stack twice.

Related Instructions

FCOM, FCOMPP, FCOMI, FCOMIP, FICOM, FICOMP, FTST, FUCOMI, FUCOMIP, FXAM

rFLAGS Affected

None

x87 Condition Code

C3	C2	C1	C0	Compare Result
0	0	0	0	ST(0) > source
0	0	0	1	ST(0) < source
1	0	0	0	ST(0) = source
1	1	0	1	Operands were unordered

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.

FUCOMI FUCOMIP Floating-Point Unordered Compare and Set eFLAGS

Compares the contents of ST(0) with the contents of another floating-point register, and sets the zero flag (ZF), parity flag (PF), and carry flag (CF) as shown in the rFLAGS Affected table below.

Unlike FCOMI and FCOMIP, the FUCOMI and FUCOMIP instructions do not set the invalid-operation exception (IE) bit in the x87 status word for QNaNs.

After completing the comparison, FUCOMIP pops the x87 register stack.

Support for the FCOMI(P) instruction can be determined by executing either CPUID function 0000_0001h or CPUID function 8000_0001. Support is indicated when both the EDX[FPU] (bit 0) and EDX[CMOV] (bit 15) feature flags are set.

Mnemonic	Opcode	Description
FUCOMI ST(0),ST(*i*)	DB E8+*i*	Compare ST(0) to ST(*i*) and set eFLAGS to reflect the result of the comparison.
FUCOMIP ST(0),ST(*i*)	DF E8+*i*	Compare ST(0) to ST(*i*), set eFLAGS to reflect the result of the comparison, and pop the x87 register stack.

Related Instructions

FCOM, FCOMPP, FCOMI, FCOMIP, FICOM, FICOMP, FTST, FUCOM, FUCOMP, FUCOMPP, FXAM

rFLAGS Affected

ZF	PF	CF	Compare Result
0	0	0	ST(0) > source
0	0	1	ST(0) < source
1	0	0	ST(0) = source
1	1	1	Operands were unordered

x87 Condition Code

x87 Condition Code	Value	Description
C0		
C1	0	

x87 Condition Code	Value	Description
C2		
C3		
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The conditional move instructions are not supported, as indicated by EDX[FPU] and EDX[CMOV] returned by CPUID function 0000_0001h or 8000_0001h.
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.

FWAIT (WAIT) — Wait for Unmasked x87 Floating-Point Exceptions

Forces the processor to test for pending unmasked floating-point exceptions before proceeding.

If there is a pending floating-point exception and CR0.NE = 1, a numeric exception (#MF) is generated. If there is a pending floating-point exception and CR0.NE = 0, FWAIT asserts the FERR output signal, then waits for an external interrupt.

This instruction is useful for insuring that unmasked floating-point exceptions are handled before altering the results of a floating point instruction.

FWAIT and WAIT are synonyms for the same opcode.

Mnemonic	Opcode	Description
FWAIT	9B	Check for any pending floating-point exceptions.

Related Instructions

None

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	U	
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The monitor coprocessor bit (MP) and the task switch bit (TS) of the control register (CR0) were both set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

FXAM Floating-Point Examine

Examines the value in ST(0) and sets the C0, C2, and C3 condition code flags in the x87 status word as shown in the x87 Condition Code table below to indicate whether the value is a NaN, infinity, zero, empty, denormal, normal finite, or unsupported value. The instruction also sets the C1 flag to indicate the sign of the value in ST(0) (0 = positive, 1 = negative).

Mnemonic	Opcode	Description
FXAM	D9 E5	Characterize the number in the ST(0) register.

Related Instructions

FCOM, FCOMP, FCOMPP, FCOMI, FCOMIP, FICOM, FICOMP, FTST, FUCOM, FUCOMI, FUCOMIP, FUCOMP, FUCOMPP

rFLAGS Affected

None

x87 Condition Code

C3	C2	C1	C0	Meaning
0	0	0	0	+unsupported format
0	0	0	1	+NaN
0	0	1	0	–unsupported format
0	0	1	1	–NaN
0	1	0	0	+normal
0	1	0	1	+infinity
0	1	1	0	–normal
0	1	1	1	–infinity
1	0	0	0	+0
1	0	0	1	+empty
1	0	1	0	–0
1	0	1	1	–empty
1	1	0	0	+denormal
1	1	1	0	–denormal

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.

FXCH Floating-Point Exchange

Exchanges the value in ST(0) with the value in any other x87 register. If no operand is specified, the instruction exchanges the values in ST(0) and ST(1).

Use this instruction to move a value from an x87 register to ST(0) for subsequent processing by a floating-point instruction that can only operate on ST(0).

Mnemonic	Opcode	Description
FXCH	D9 C9	Exchange the contents of ST(0) and ST(1).
FXCH ST(*i*)	D9 C8+*i*	Exchange the contents of ST(0) and ST(*i*).

Related Instructions

FLD, FST, FSTP

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.

FXTRACT Floating-Point Extract Exponent and Significand

Extracts the exponent and significand portions of the floating-point value in ST(0), stores the exponent in ST(0), and then pushes the significand onto the x87 register stack. After this operation, the new ST(0) contains a real number with the sign and value of the original significand and an exponent of 3FFFh (biased value for true exponent of zero), and ST(1) contains a real number that is the value of the original value's true (unbiased) exponent.

The FXTRACT instruction is useful for converting a double-extended-precision number to its decimal representation.

If the zero-divide-exception mask (ZM) bit of the x87 control word is set to 1 and the source value is ±0, then the instruction stores ±zero in ST(0) and an exponent value of –∞ in register ST(1).

Mnemonic	Opcode	Description
FXTRACT	D9 F4	Extract the exponent and significand of ST(0), store the exponent in ST(0), and push the significand onto the x87 register stack.

Related Instructions

FABS, FPREM, FRNDINT, FCHS

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
	1	x87 stack overflow, if an x87 register stack fault was detected.
C2	U	
C3	U	
Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*		

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) is set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
	X	X	X	An x87 stack overflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Zero-divide exception (ZE)	X	X	X	The source operand was ±zero.

FYL2X Floating-Point y ∗ Log_2 (x)

Computes (ST(1) ∗ $\log_2$(ST(0))), stores the result in ST(1), and pops the x87 register stack. The value in ST(0) must be greater than zero.

If the zero-divide-exception mask (ZM) bit in the x87 control word is set to 1 and ST(0) contains ±zero, the instruction returns ∞ with the opposite sign of the value in register ST(1).

Mnemonic	Opcode	Description
FYL2X	D9 F1	Replace ST(1) with ST(1) ∗ $\log_2$(ST(0)), then pop the x87 register stack.

Related Instructions

FYL2XP1, F2XM1

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	No precision exception occurred.
	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN value or an unsupported format.
	X	X	X	The source operand in ST(0) was a negative finite value (not -zero).
	X	X	X	The source operand in ST(0) was +1 and the source operand in ST(1) was ±infinity.
	X	X	X	The source operand in ST(0) was -infinity.
	X	X	X	The source operand in ST(0) was ±zero or ±infinity and the source operand in ST(1) was ±zero.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Zero-divide exception (ZE)	X	X	X	The source operand in ST(0) was ±zero and the source operand in ST(1) was a finite value.
Overflow exception (OE)	X	X	X	A rounded result was too large to fit into the format of the destination operand.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

FYL2XP1 Floating-Point y ∗ Log_2 (x+1)

Computes (ST(1) ∗ $\log_2$(ST(0) + 1.0)), stores the result in ST(1), and pops the x87 register stack. The value in ST(0) must be in the range sqrt(1/2)–1 to sqrt(2)-1.

Mnemonic	Opcode	Description
FYL2XP1	D9 F9	Replace ST(1) with ST(1) ∗ $\log_2$(ST(0) + 1.0), then pop the x87 register stack.

Related Instructions

FYL2X, F2XM1

rFLAGS Affected

None

x87 Condition Code

x87 Condition Code	Value	Description
C0	U	
C1	0	x87 stack underflow, if an x87 register stack fault was detected.
	0	Result was rounded down, if a precision exception was detected.
	1	Result was rounded up, if a precision exception was detected.
C2	U	
C3	U	

Note: *A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Device not available, #NM	X	X	X	The emulate bit (EM) or the task switch bit (TS) of the control register (CR0) was set to 1.
x87 floating-point exception pending, #MF	X	X	X	An unmasked x87 floating-point exception was pending.
x87 Floating-Point Exception Generated, #MF				
Invalid-operation exception (IE)	X	X	X	A source operand was an SNaN or unsupported format.
	X	X	X	The source operand in ST(0) was ±0 and the source operand in ST(1) was ±infinity.
Invalid-operation exception (IE) with stack fault (SF)	X	X	X	An x87 stack underflow occurred.
Denormalized-operand exception (DE)	X	X	X	A source operand was a denormal value.
Overflow exception (OE)	X	X	X	A rounded result was too large to fit into the format of the destination operand.
Underflow exception (UE)	X	X	X	A rounded result was too small to fit into the format of the destination operand.
Precision exception (PE)	X	X	X	A result could not be represented exactly in the destination format.

Appendix A Recommended Substitutions for 3DNow!™ Instructions

Table A-1 lists the deprecated 3DNow!™ instructions and the recommended substitutions.

Table A-1. Substitutions for 3DNow!™ Instructions

64-Bit 3DNow!™ Instruction	128-Bit SSE Instruction	64-Bit MMX™ Instruction	Notes
FEMMS	N/A	EMMS (MMX)	
PAVGUSB	PAVGB	PAVGB	SSE and MMX™ instructions round according to the current rounding mode; 3DNow!™ instructions always round up.
PF2ID	CVTTPS2DQ		
PF2IW			CVTTPS2DQ may be used if 16-bit result is not necessary.
PFACC	HADDPS		
PFADD	ADDPS		
PFCMPEQ	CMPPS		
PFCMPGE	CMPPS		
PFCMPGT	CMPPS		
PFMAX	MAXPS		MAXPS may return -0.0.
PFMIN	MINPS		MINPS may return -0.0.
PFMUL	MULPS		
PFNACC	HSUBPS		
PFPNACC	ADDSUBPS		ADDSUBPS expects arguments in different positions from PFPNACC.
PFRCP			RCPSS may be used in conjunction with the Newton-Raphson algorithm.
PFRCPIT1			See PFRCP.
PFRCPIT2			See PFRCP.
PFRSQIT1			See PFRSQRT.
PFRSQRT			RSQRTSS may be used in conjunction with the Newton-Raphson algorithm.
PFSUB	SUBPS		
PFSUBR			SUBPS may be used.
PI2FD	CVTDQ2PS		SSE instructions round according to the current rounding mode; 3DNow! instructions always truncate.
PI2FW			
PMULHRW			PMULHW may be used if rounding is not necessary.
PSWAPD	PSHUFD		

Index

I

L

M

O

P

Q

R

S

T

U

V

W

X