AMD64 Technology

AMD64 Architecture Programmer's Manual

Volume 3: General-Purpose and System Instructions

Publication No.	Revision	Date
24594	3.34	October 2022

Advanced Micro Devices

Trademarks

Contents

Figures

Tables

Revision History

Date	Revision	Description
November 2022	3.34	Chapter 4: Added RMPQUERY instruction. Appendix E: Added CPUID Fn8000_0020 - Platform QoS Extended Features Added CPUID Fn8000_0022 - Extended PerformanceMonitoring and Debug Added CPUID Fn8000_0023 - Multi-Key Encrypted Memory Capabilities Added CPUID Fn8000_0026 - Extended CPU Topology. Added CPUID information for new features.
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 643. 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 593 to make instruction list comprehensive. Added a new Appendix E “Obtaining Processor Information Via the CPUID Instruction” on page 599 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 Pshaw instructions in Table A-28 on page 550.
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. Adde A.d opcodes for the instructions to the opcode maps in Appendix

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 omitted 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 552 and Table A-34 on page 554. Changed notational convention for indicating addressing modes in Table A-33 on page 552, Table A-35 on page 555, Table A-36 on page 557, and Table A-37 on page 558; 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 555.
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 multi-volume 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 "_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

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.

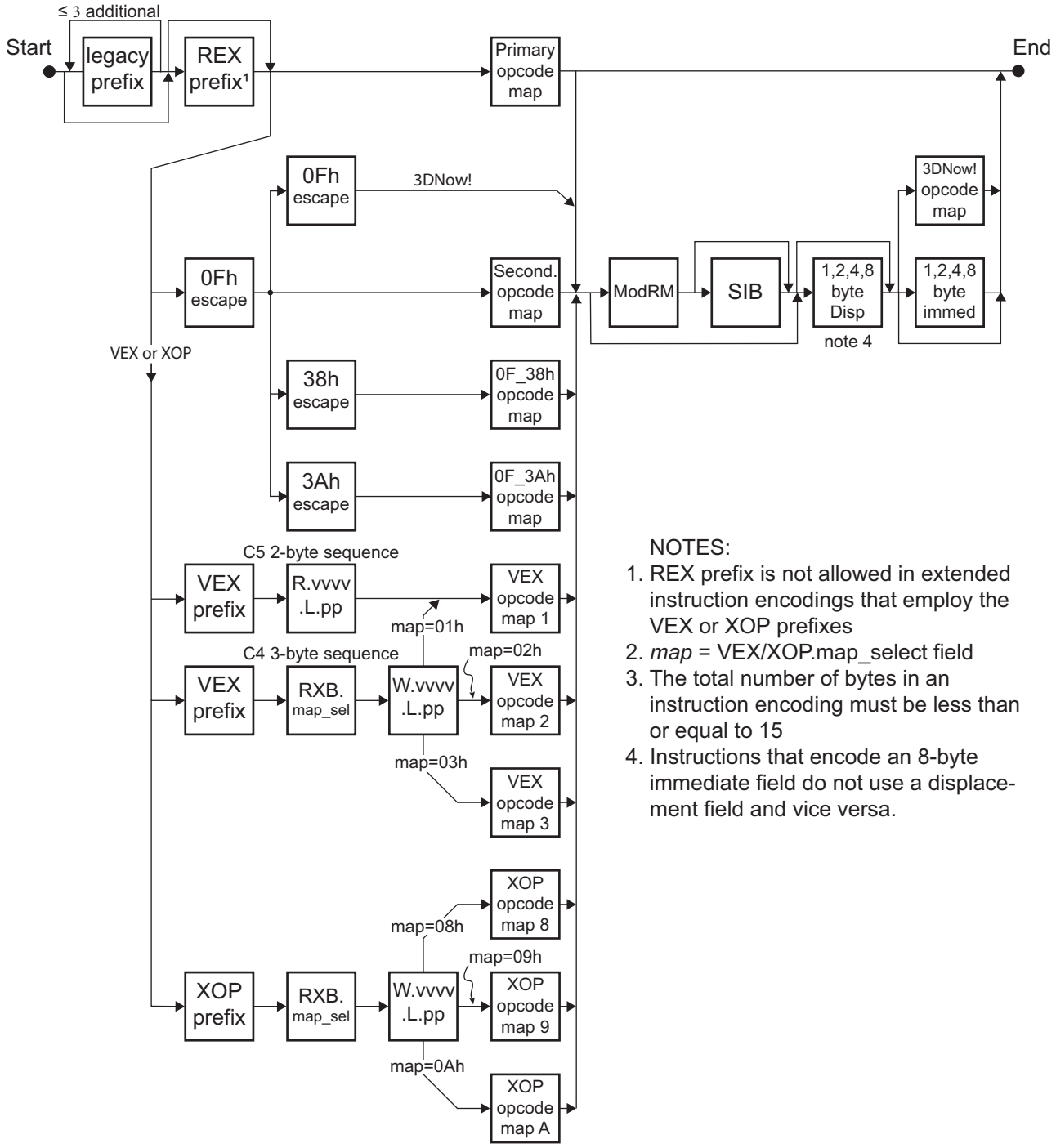


Figure 1-1. Instruction Encoding Syntax

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 “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 “0F_38h and 0F_3Ah Opcode Maps” of Appendix A.

1.1.1.5 ModRM and SIB Bytes

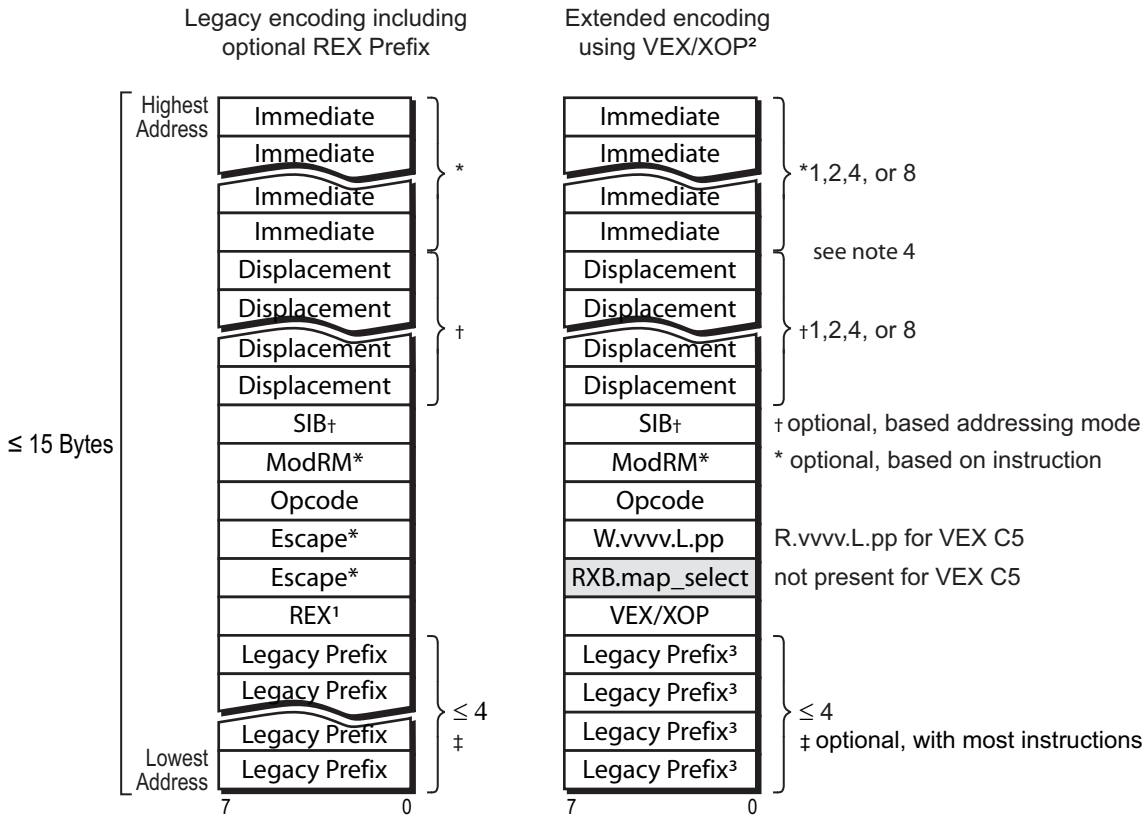
The opcode can be followed by a *mode-register-memory* (ModRM) byte, which further describes the operation and/or operands. The ModRM byte may also be followed by a *scale-index-base* (SIB) byte, which is used to specify indexed register-indirect forms of memory addressing. The ModRM and SIB bytes are 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-purpose 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 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	

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 in-line 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.

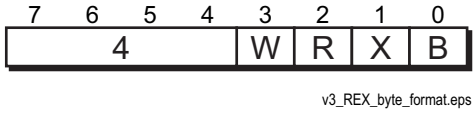


Figure 1-3. REX Prefix Format

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
J*r*CXZ	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 specif 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.

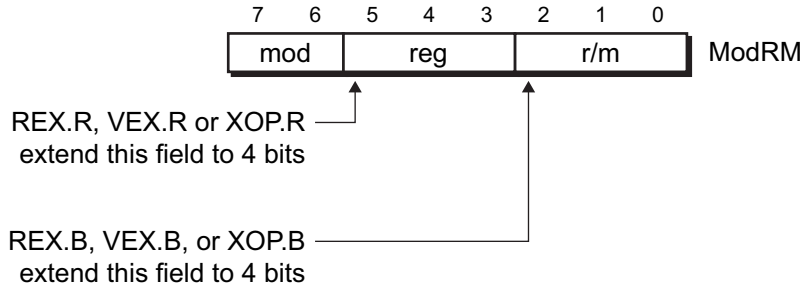


Figure 1-4. ModRM-Byte Format

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.

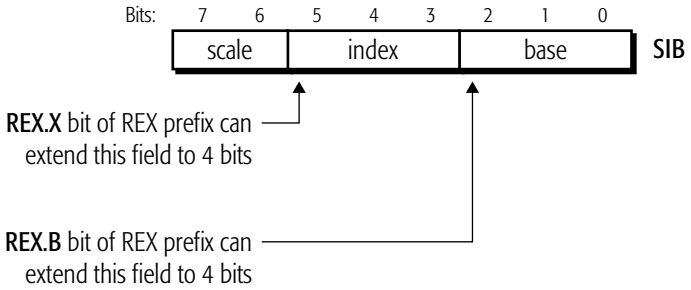


Figure 1-5. SIB Byte Format

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

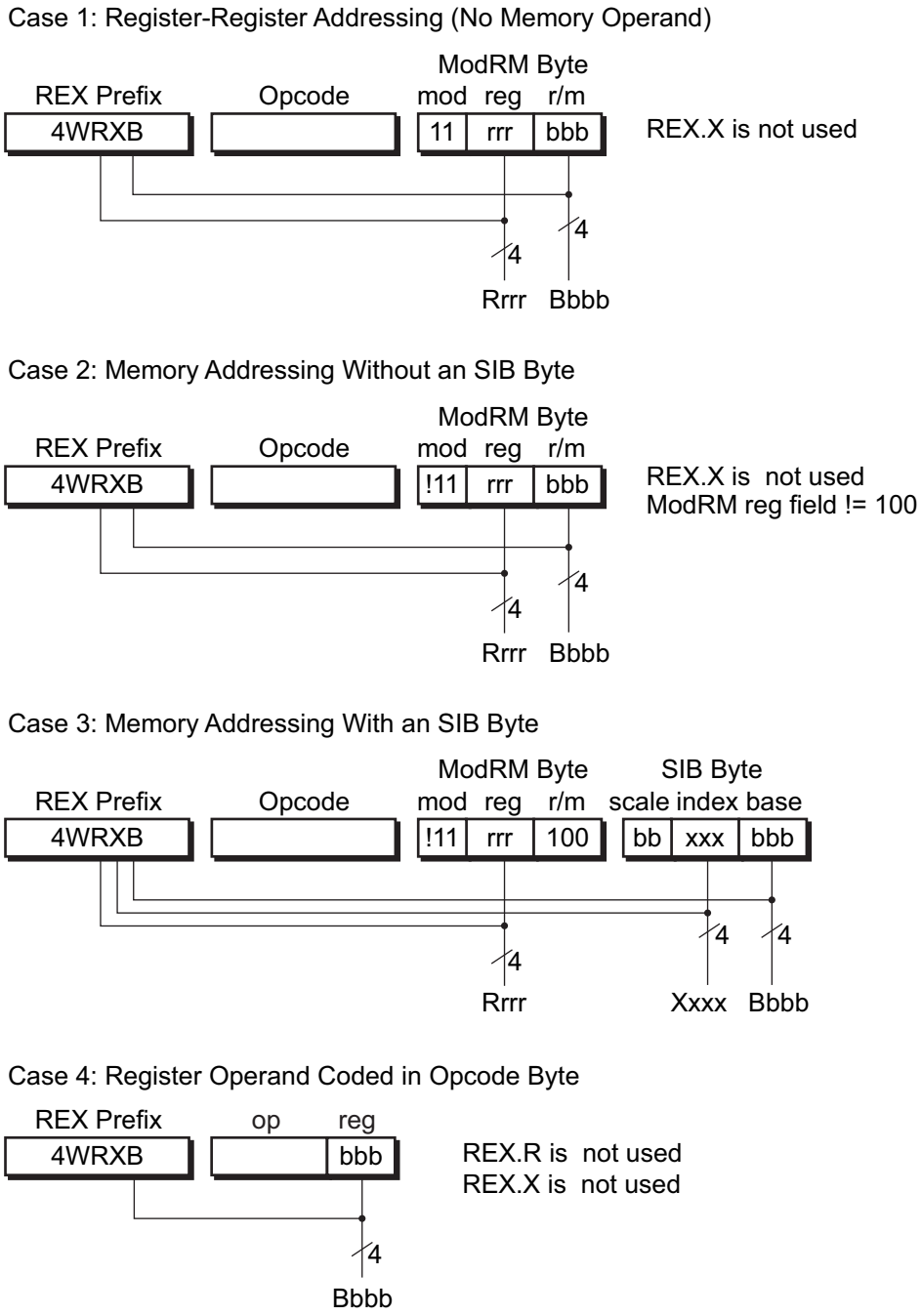


Figure 1-6. Encoding Examples Using REX R, X, and B Bits

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.

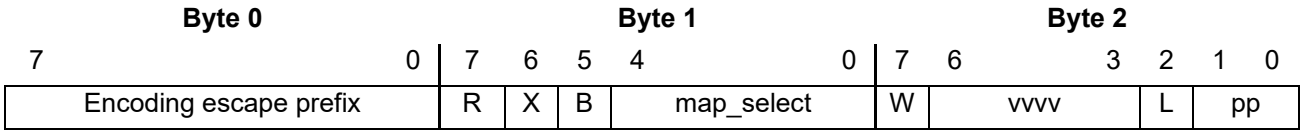


Figure 1-7. VEX/XOP Three-byte Escape Sequence Format

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.

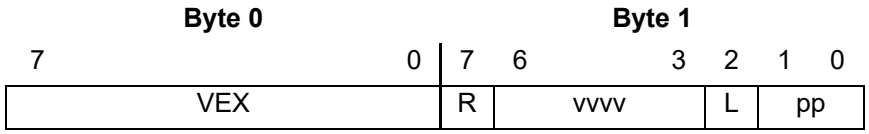


Figure 1-8. VEX Two-byte Escape Sequence Format

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.

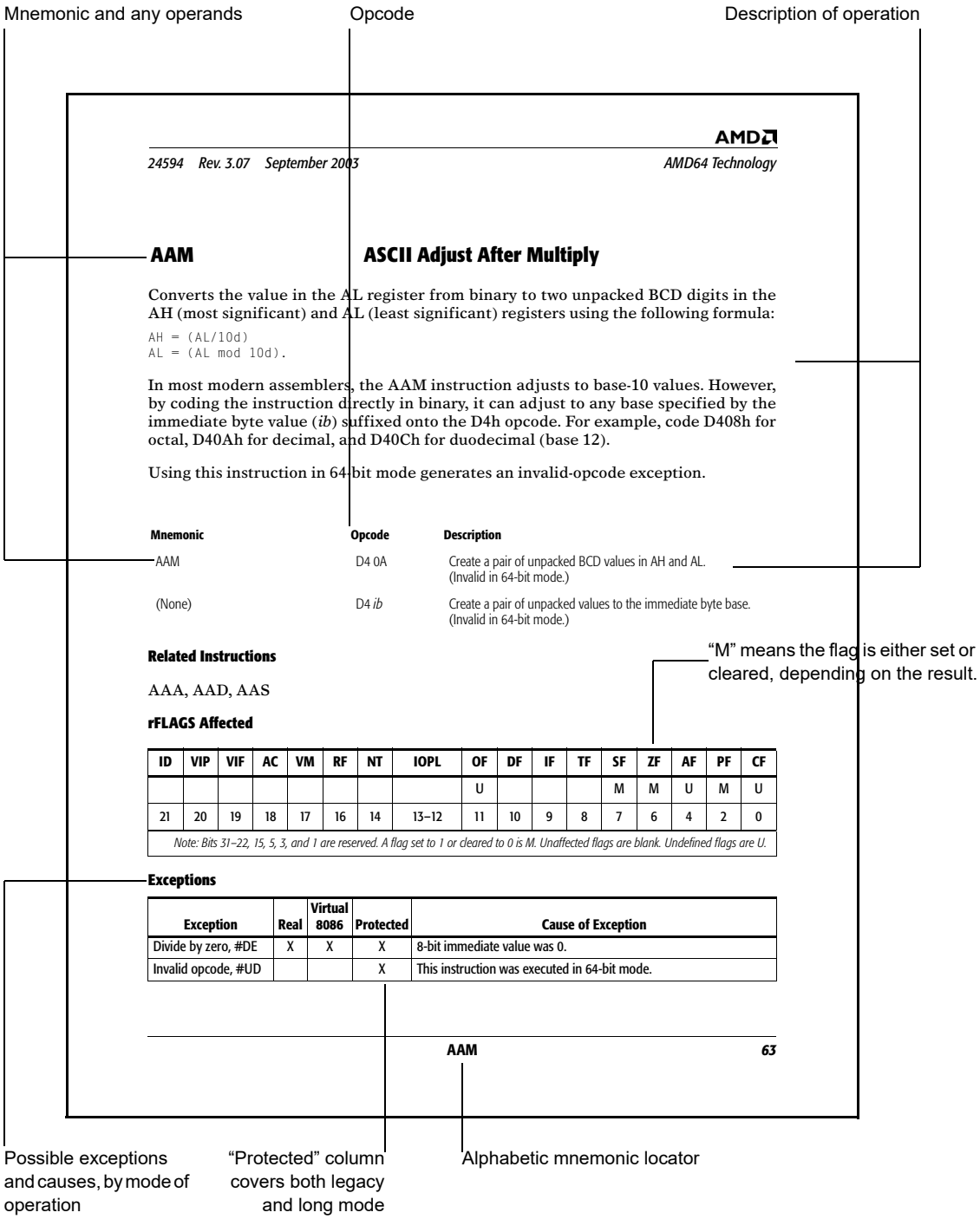


Figure 2-1. Format of Instruction-Detail Pages

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.

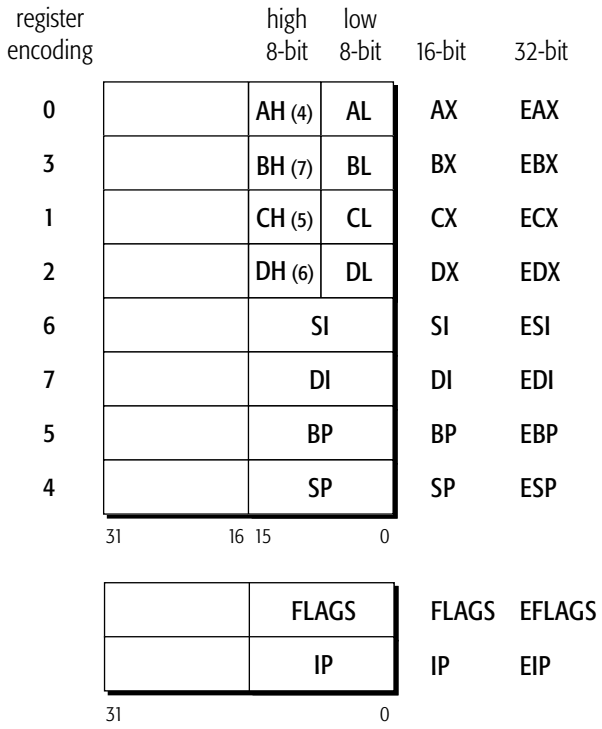


Figure 2-2. General Registers in Legacy and Compatibility Modes

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).

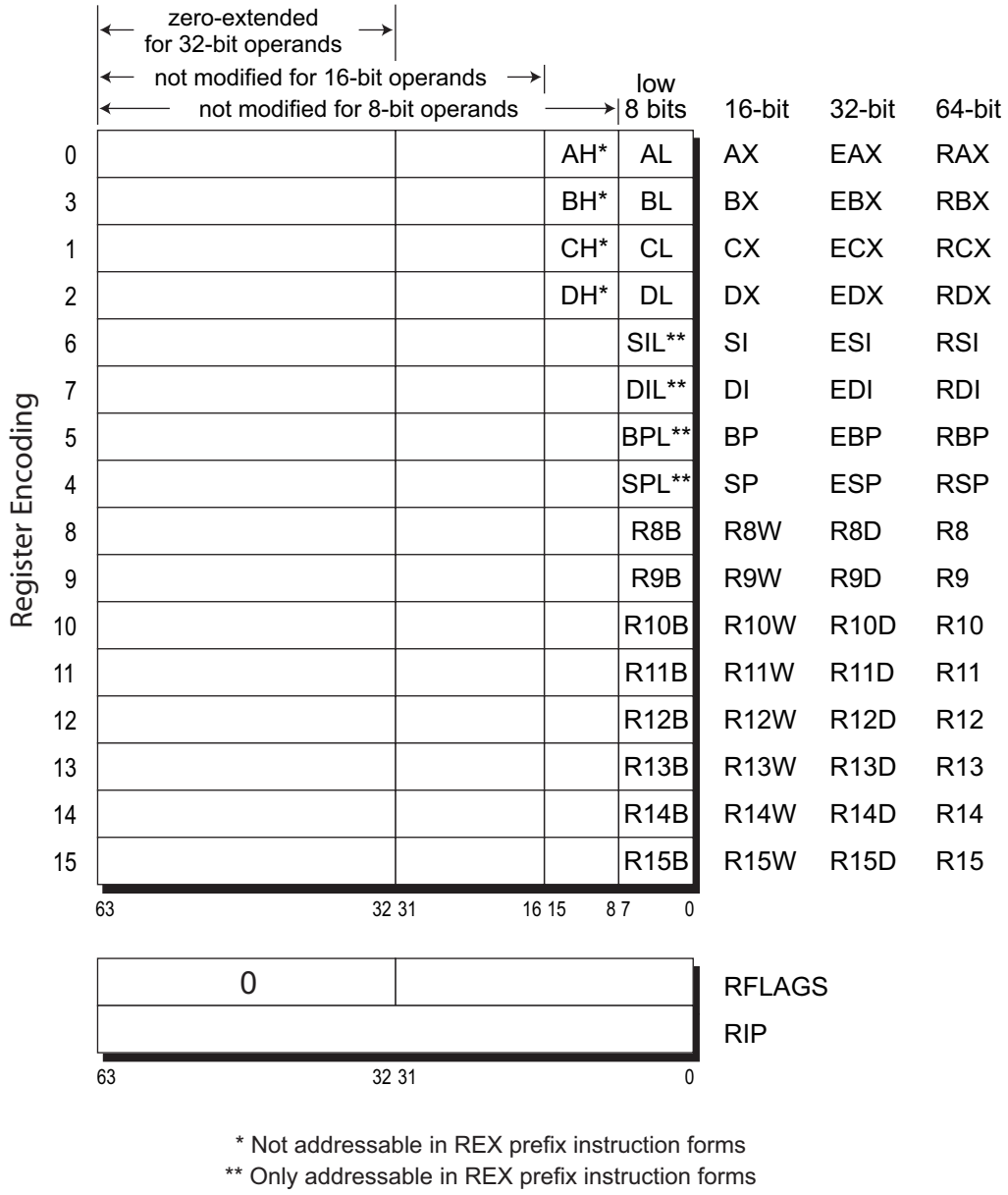


Figure 2-3. General Registers in 64-Bit Mode

For most instructions running in 64-bit mode, access to the extended GPRs requires a either a REX instruction modification prefix or extended encoding 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.

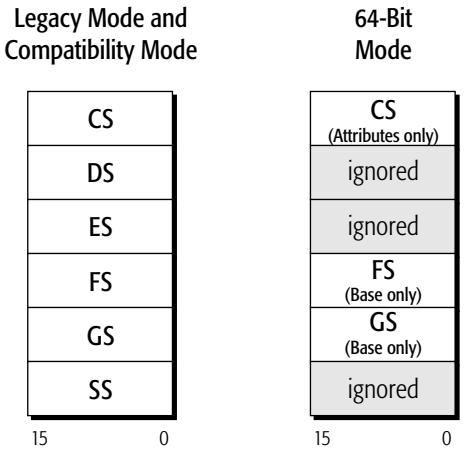


Figure 2-4. Segment Registers

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.

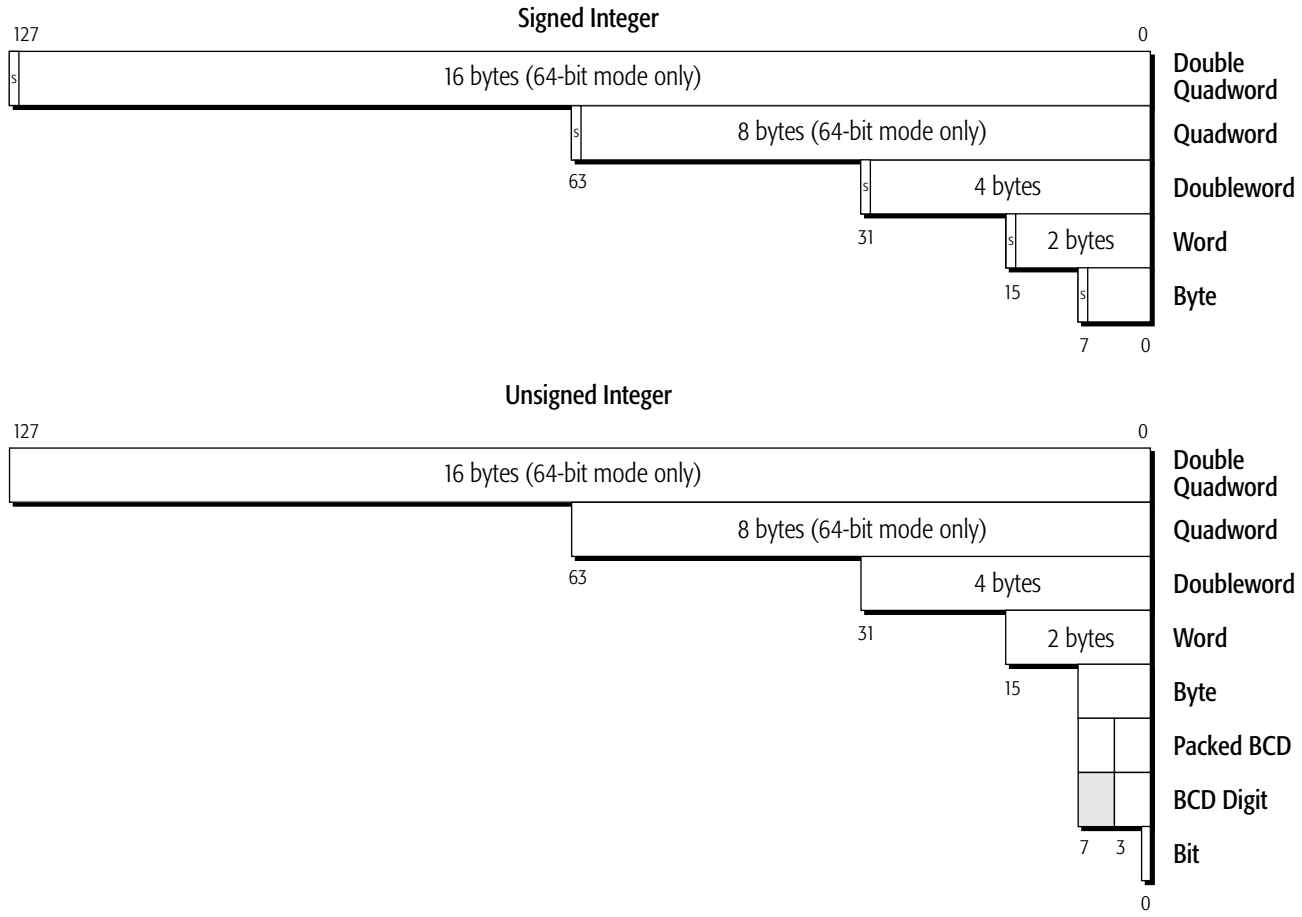


Figure 2-5. General-Purpose Data Types

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.

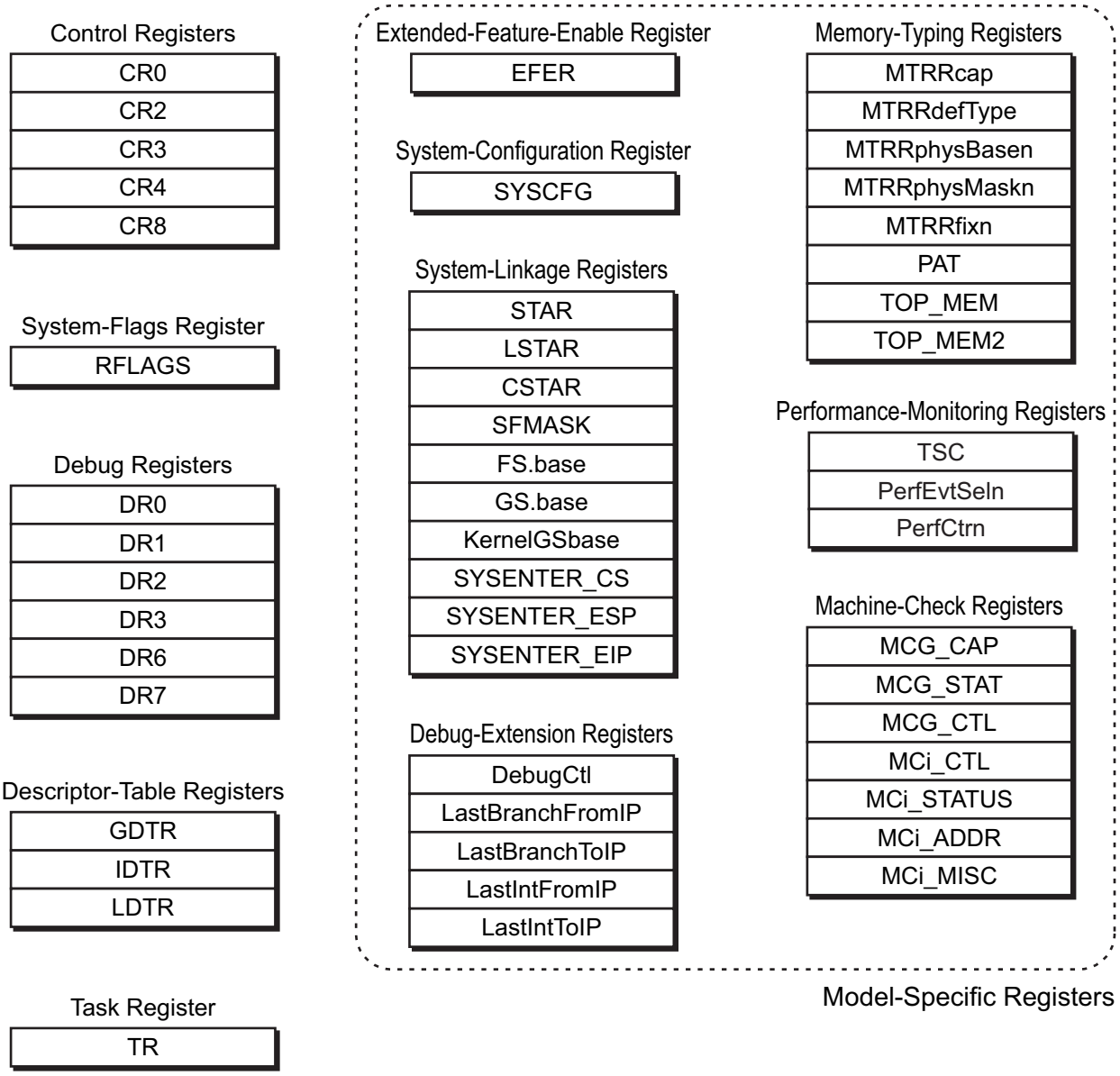


Figure 2-6. System Registers

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.

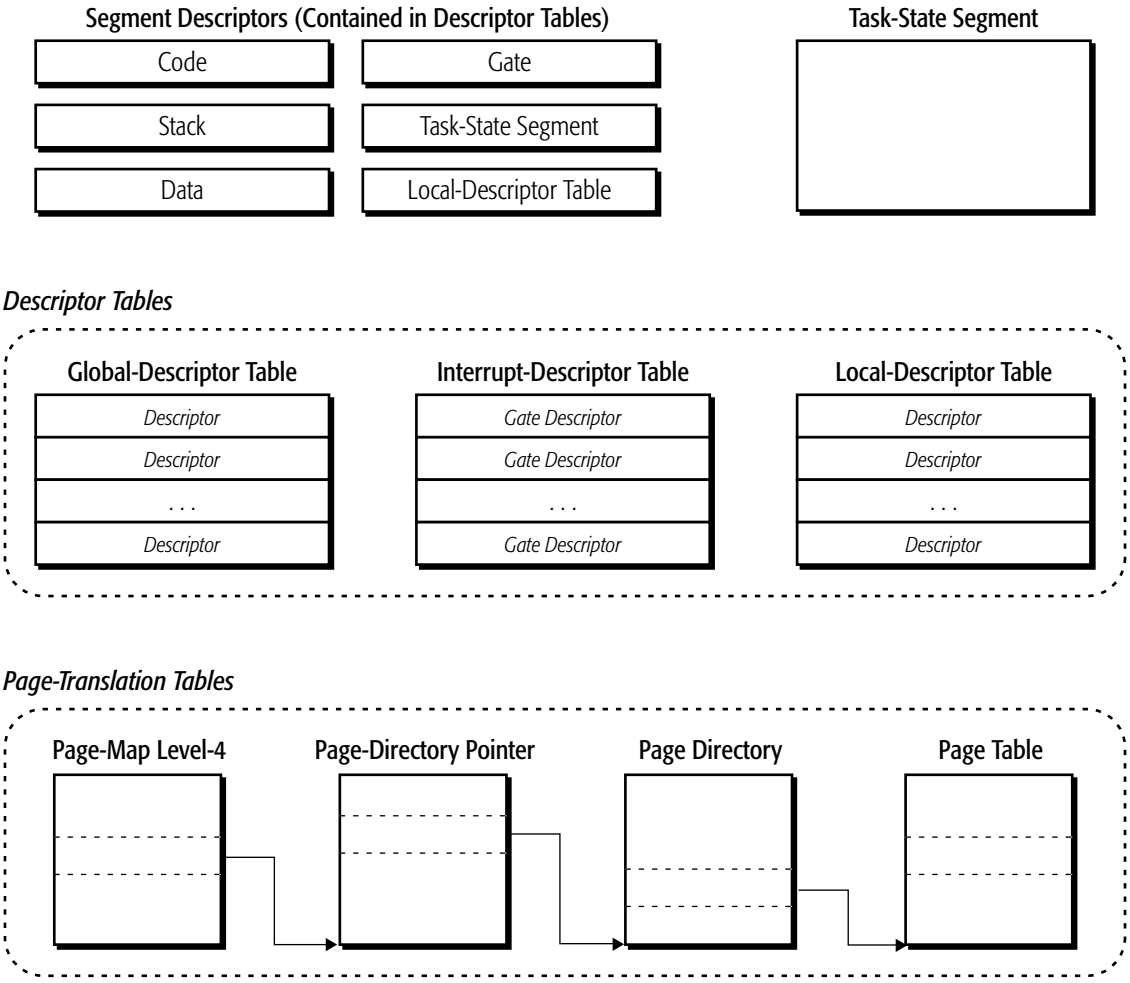


Figure 2-7. System Data Structures

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).

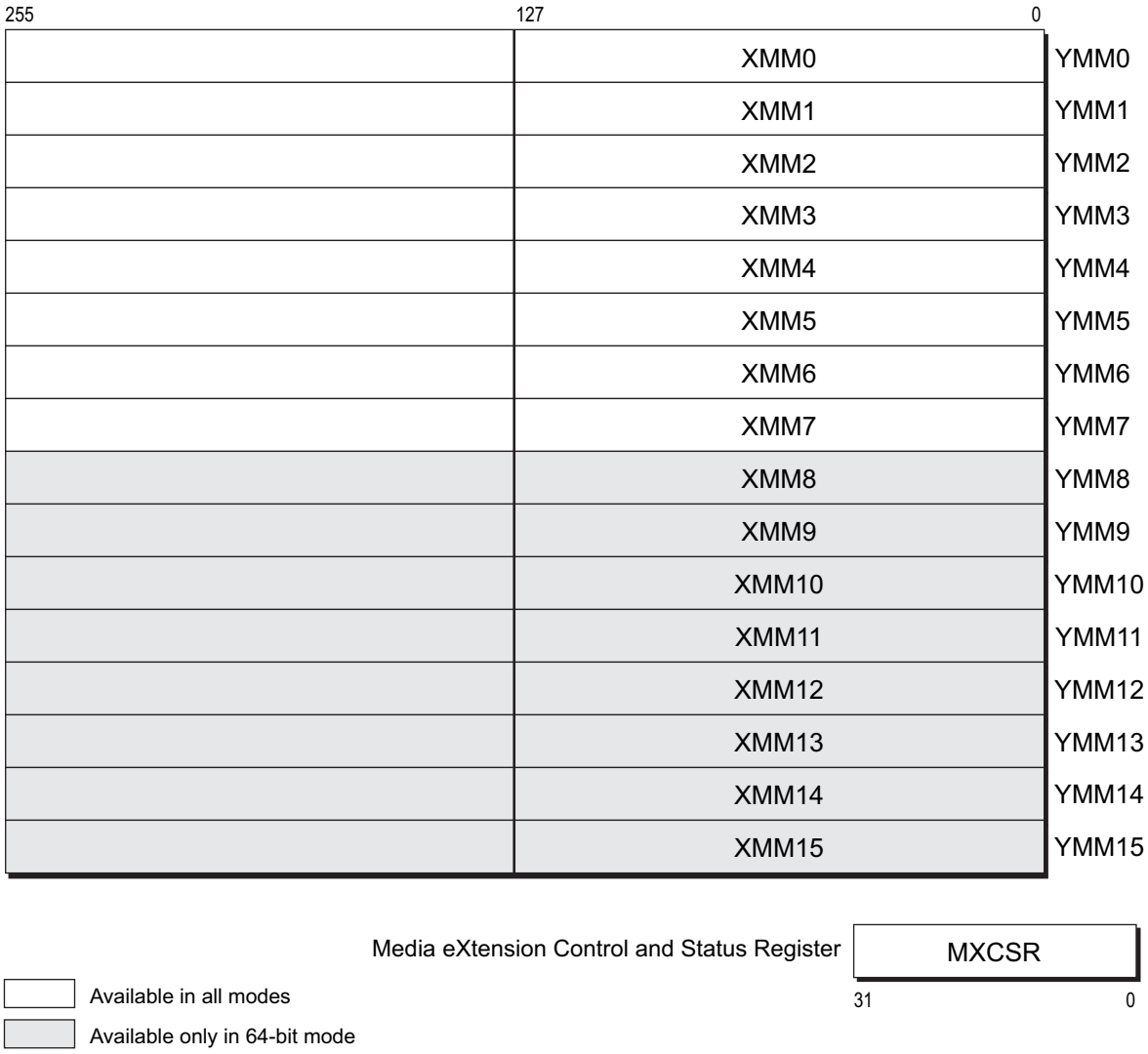


Figure 2-8. SSE Registers

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.

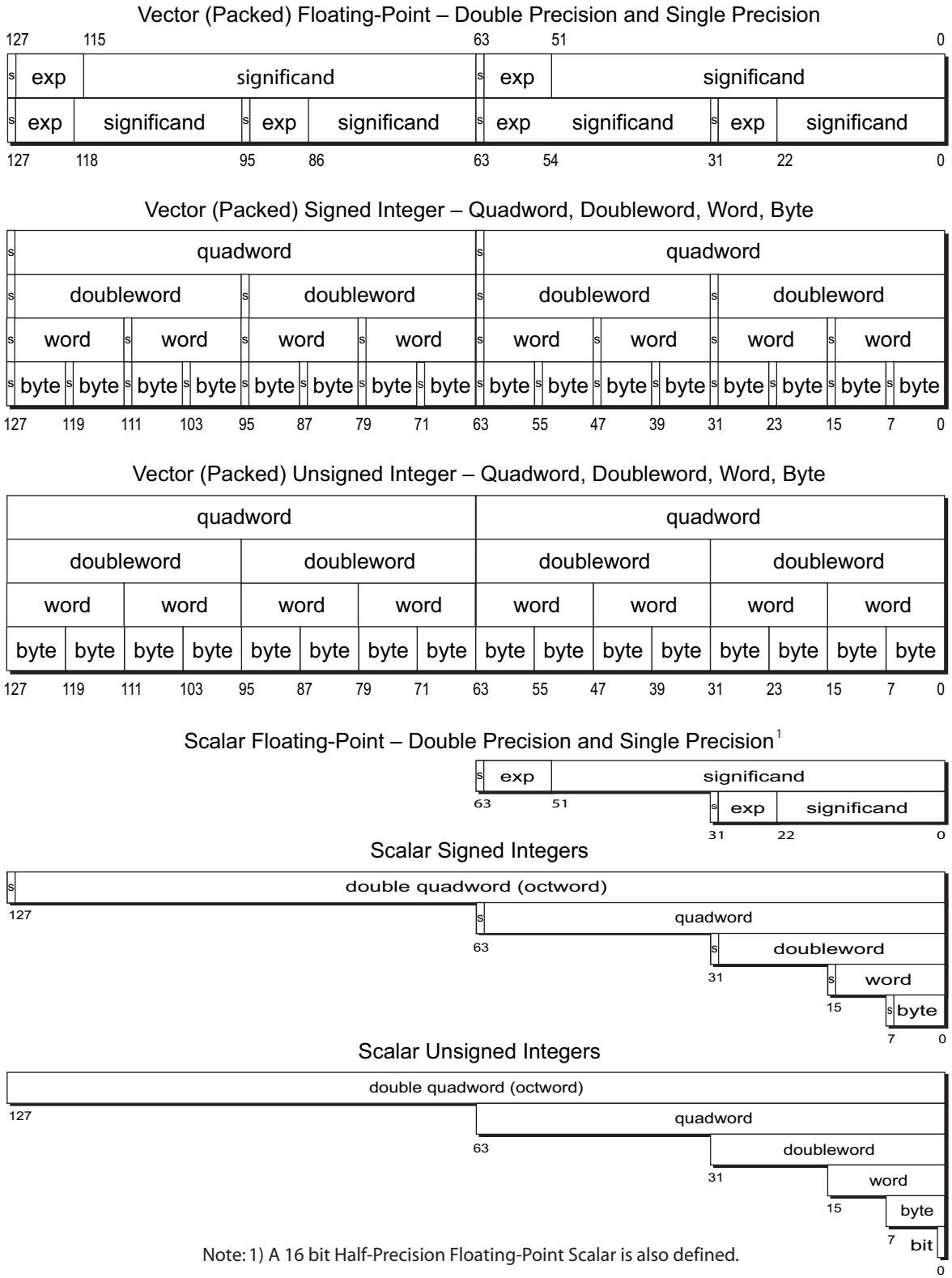


Figure 2-9. 128-Bit SSE Data Types

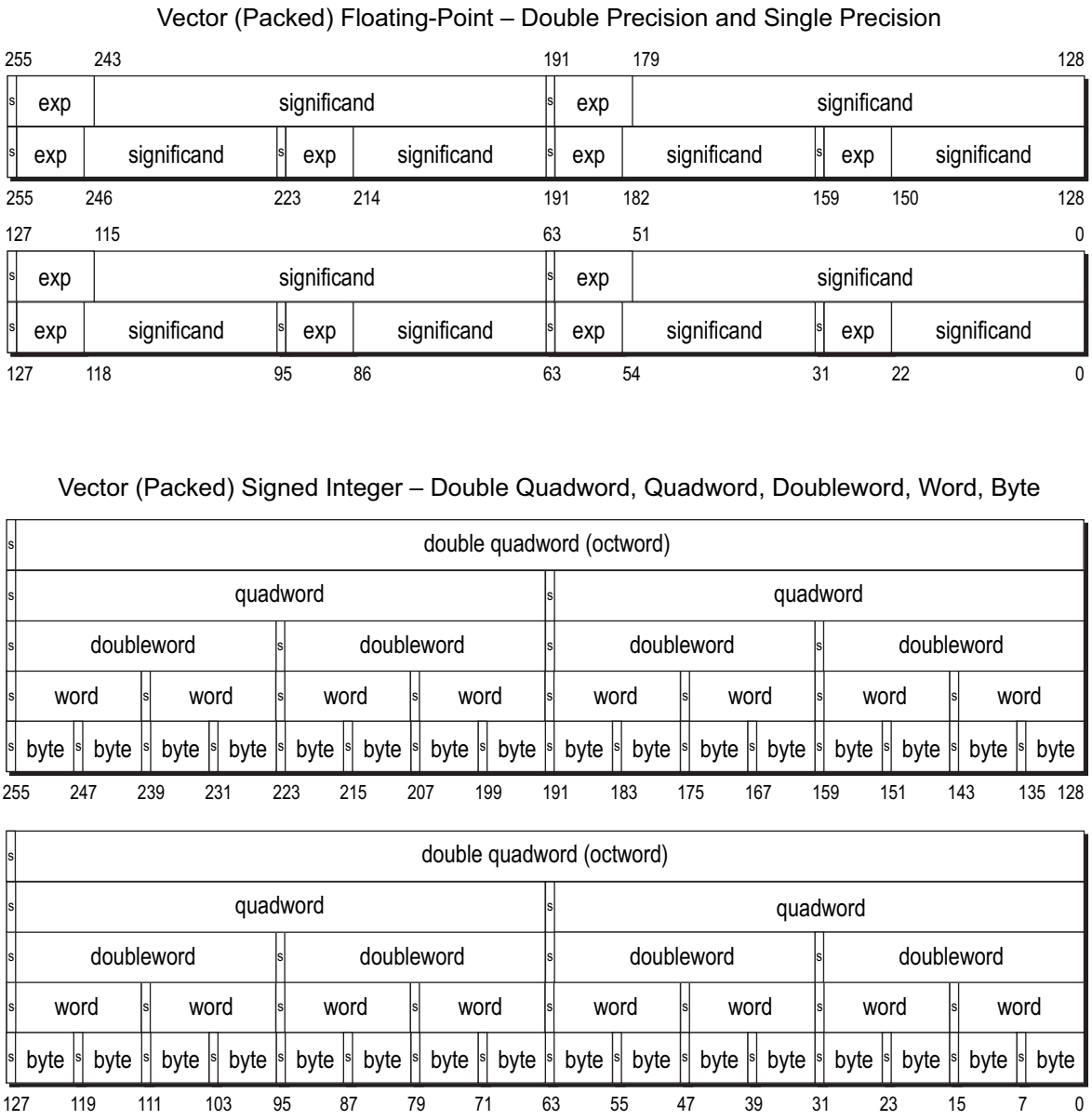
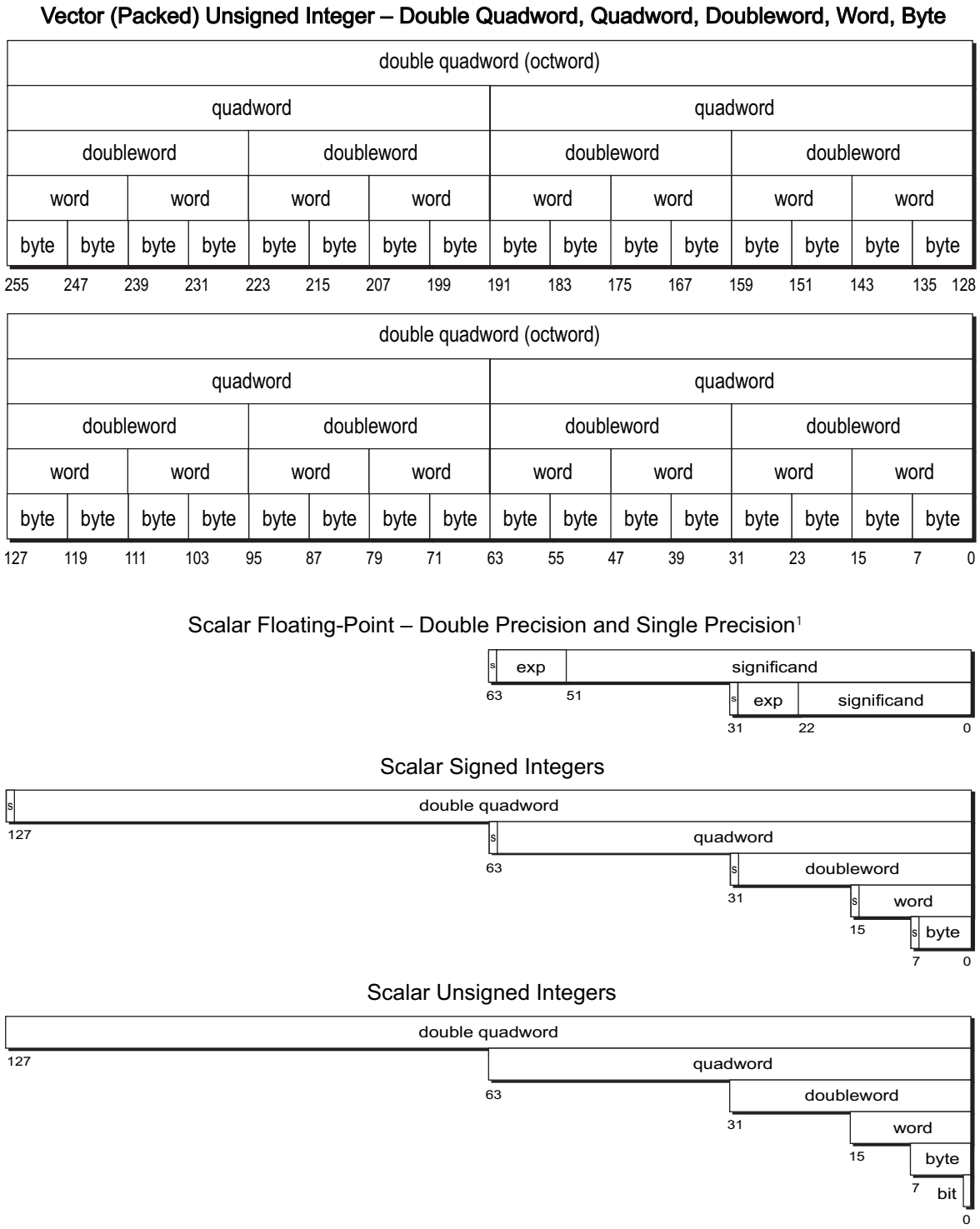


Figure 2-10. SSE 256-bit Data Types



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).

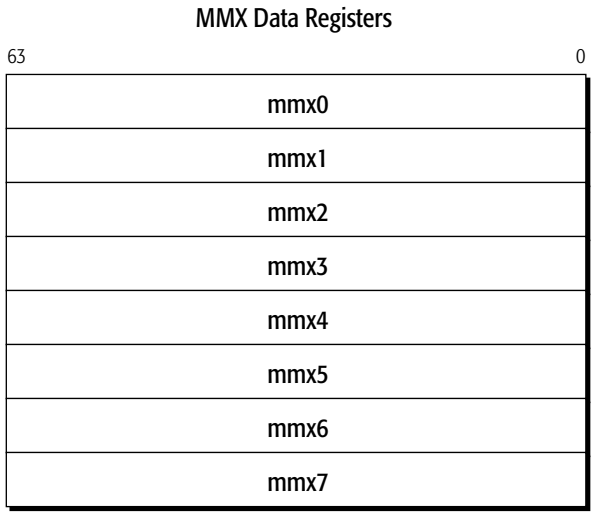


Figure 2-12. 64-Bit Media Registers

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.

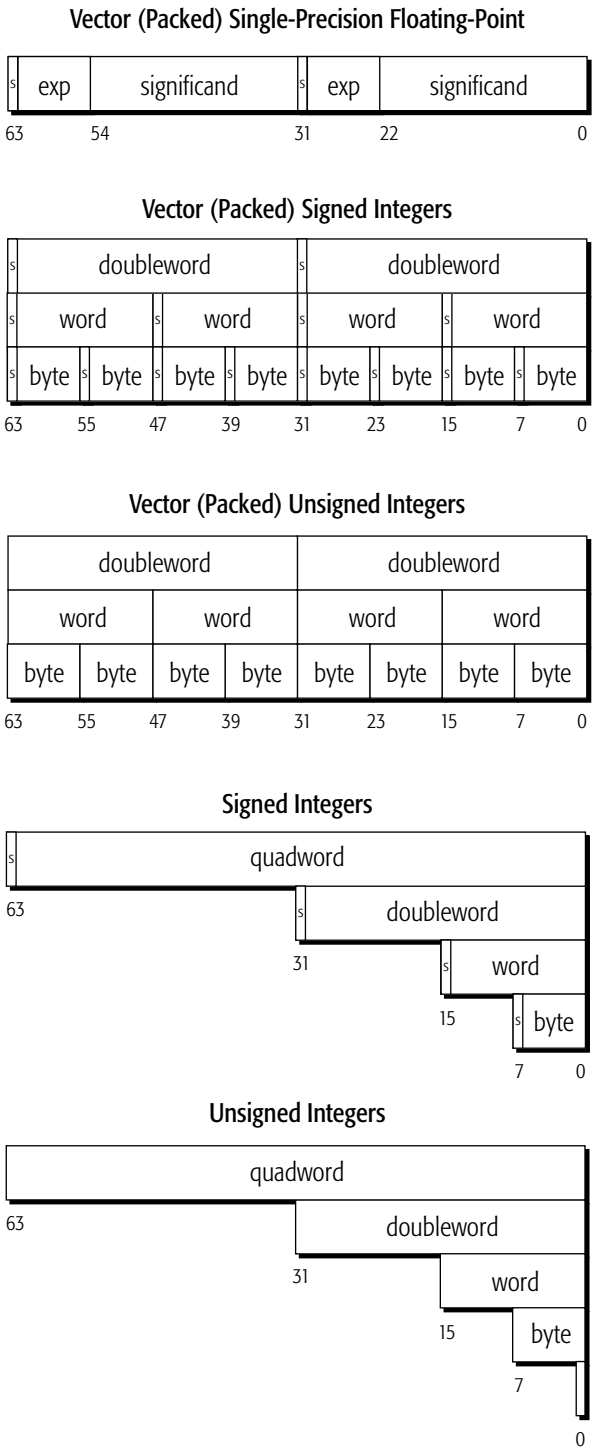


Figure 2-13. 64-Bit Media Data Types

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.

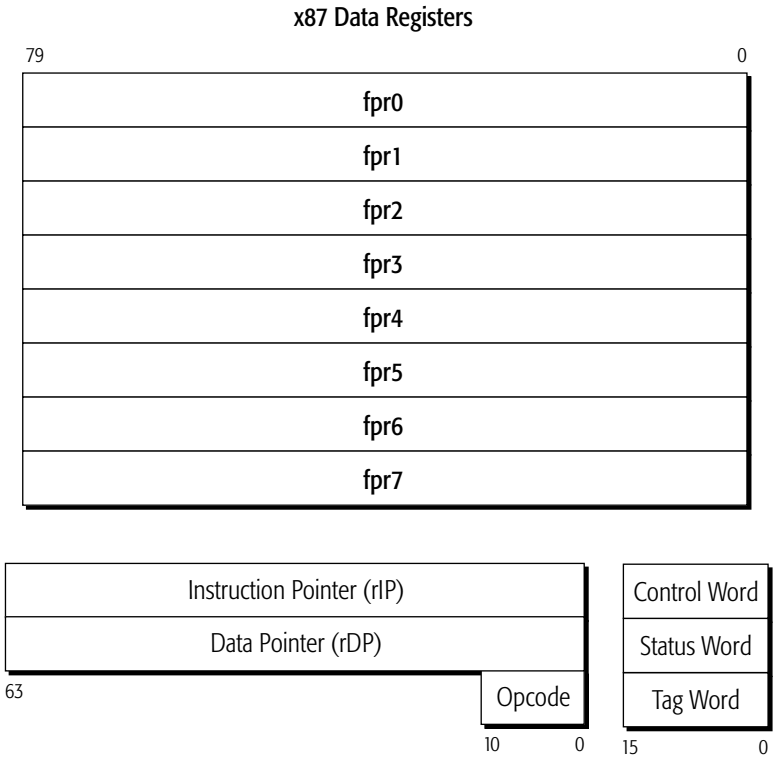


Figure 2-14. x87 Registers

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.

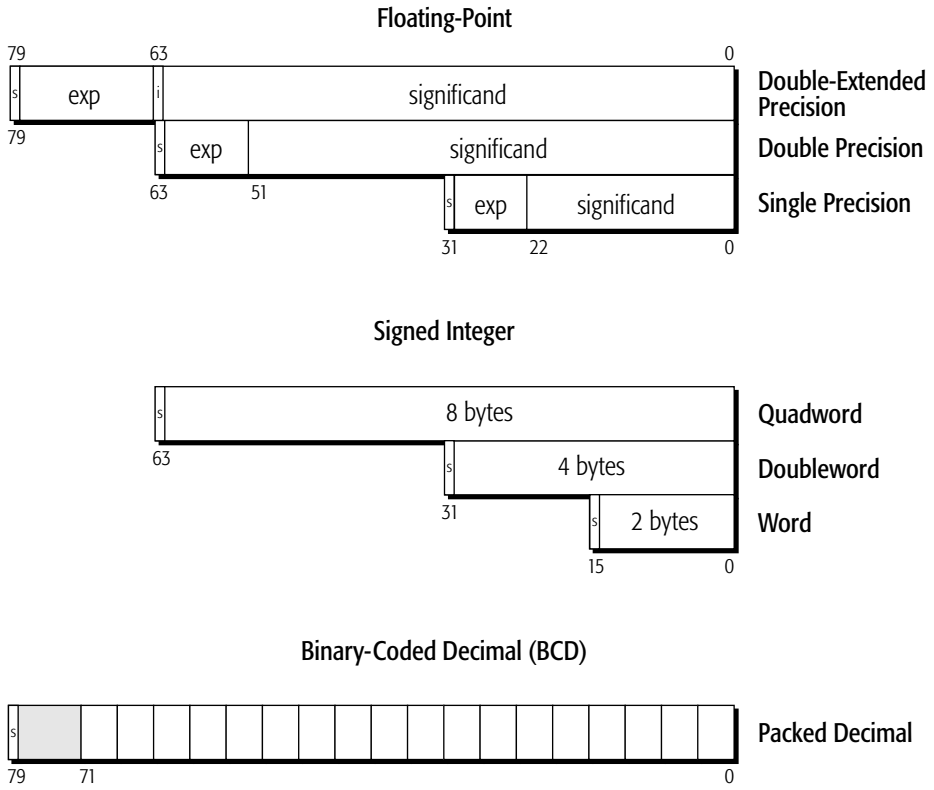


Figure 2-15. x87 Data Types

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.

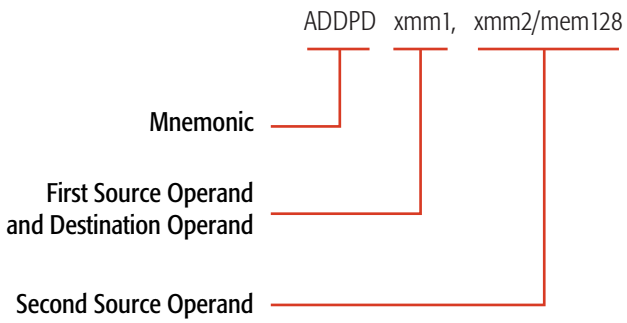


Figure 2-16. Syntax for Typical Two-Operand Instruction

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 591.

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 559.

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-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.

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 591.

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 591.

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 591.

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 591.

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 591.

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 591.

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 591.

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 591.

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 591.

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 591.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
BLSI *reg32*, *reg/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{dest}}$.0.00	F3 /3
BLSI *reg64, reg/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{dest}}$.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 591.

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 591.

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 591.

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 591.

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 591.

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 591.

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 591.

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.

When HWCR[CpuidUserDis]=0, software operating at any privilege level can execute the CPUID instruction to collect this information. When HWCR[CpuidUserDis]=1 and not in SMM, only privileged software can execute the CPUID instruction. CPUID Fn8000_0021_EAX[CpuidUserDis] (bit 17) indicates support for CPUID instruction disable for non-privileged software (CPL>0).

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 591. For a description of all defined feature numbers and return values, see Appendix E, "Obtaining Processor Information Via the CPUID Instruction," on page 597.

Related Instructions

None

rFLAGS Affected

None

Exceptions

Exception	Mode			Cause of Exception
	Real	Virtual 8086	Protected	
General Protection, #GP	X	X	X	HWCR[CpuidUserDis] = 1 and CPL was not 0

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 591.

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 196.

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 591.

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 198.

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 591.

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 591.

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 ≠ 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 591.

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 591.

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 591.

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 198.

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 591.

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 591.

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 240 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 591.

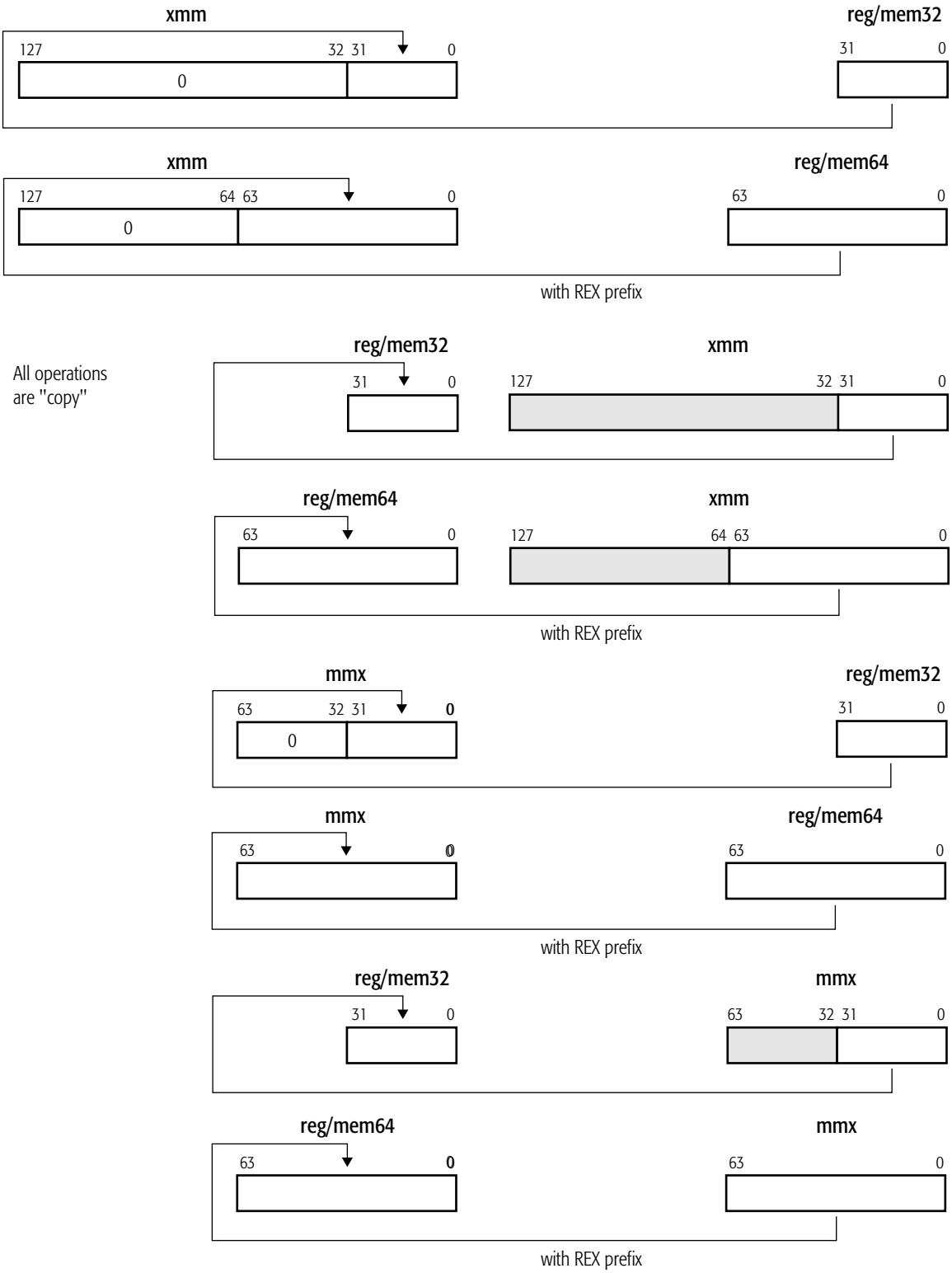


Figure 3-1. MOVD Instruction Operation

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

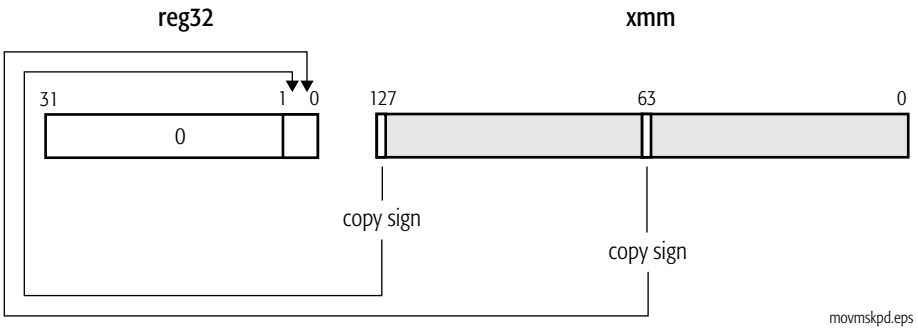
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode, #UD	X	X	X	MMX instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[MMX] or Fn0000_0001_EDX[MMX] = 0.
	X	X	X	SSE2 instructions are not supported, as indicated by CPUID Fn0000_0001_EDX[SSE2] = 0.
	X	X	X	The emulate bit (EM) 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 591.

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.

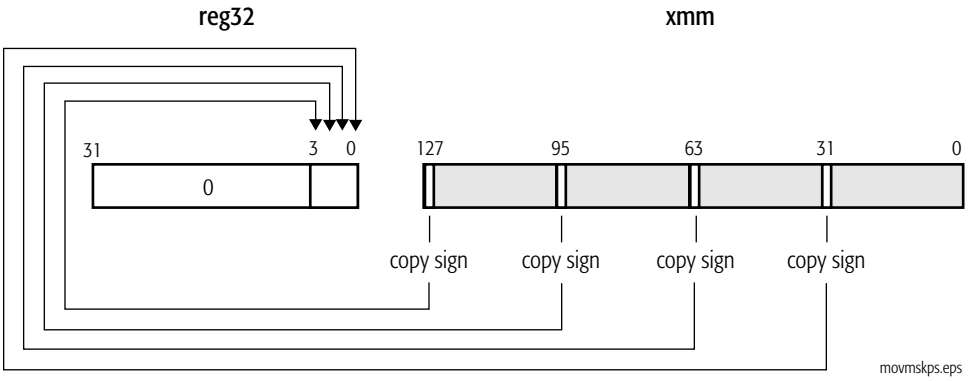
MOVMSKPS Extract Packed Single-Precision Floating-Point Sign Mask

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 591.

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 591.

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 591.

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

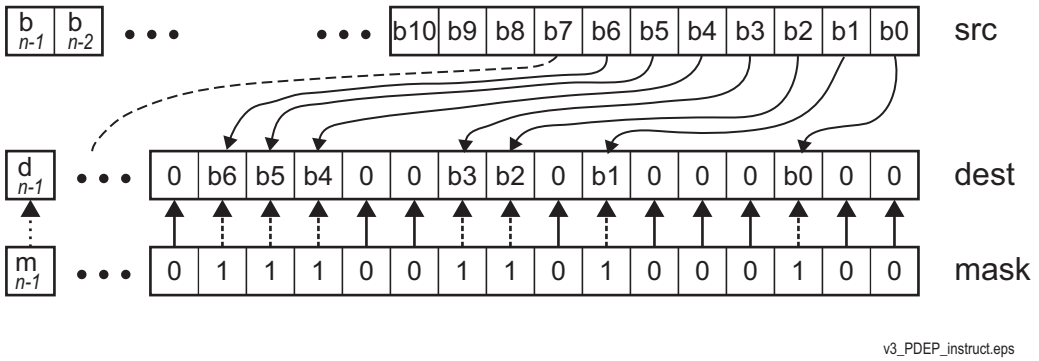
PDEP Parallel Deposit Bits

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 591.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
PDEP *reg32*, *reg32, reg/mem32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{\textit{src}}$.0.11	F5 /r
PDEP *reg64, reg64, reg/mem64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{\textit{src}}$.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.

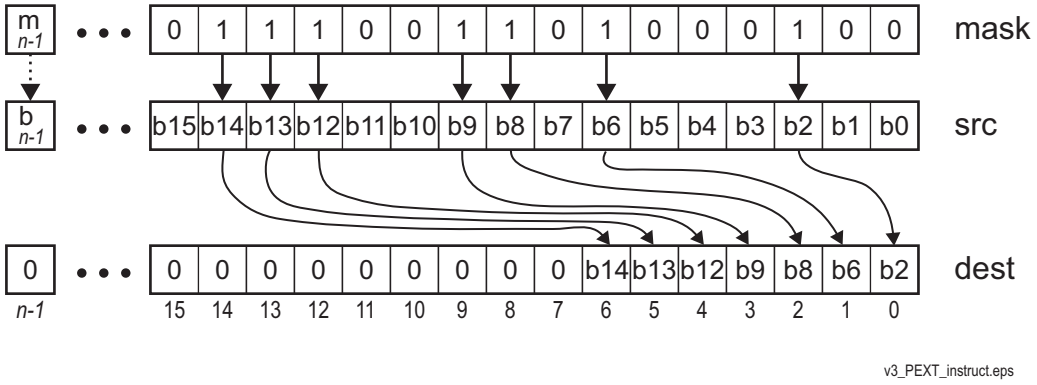
PEXT Parallel Extract Bits

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 591.

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 591.

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 591.

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 591.

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 591.

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 591.

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 591.

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 591.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
SARX *reg32, reg/mem32, reg32*	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*, section7.4 "Memory Types" on page 198.

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 591.

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 591.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
SHLX *reg32, reg/mem32, reg32*	C4	$\overline{\text{RXB}}$.02	0.$\overline{src2}$.0.01	F7 /r
SHLX *reg64, reg/mem64, reg64*	C4	$\overline{\text{RXB}}$.02	1.$\overline{src2}$.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 591.

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
SHRX *reg32*, *reg/mem32,* r*eg32*	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 591.

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 591.

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 591.

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 591.

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 591.

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]
RMPQUERY	8000_001F_EAX[RMPQUERY]	EAX[6]
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 591. 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 597.

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 597 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 up to 16 northbridge performance counters NB_PerfCtr0–15 and four L2 cache performance counters L2I_PerfCtr0–3.

The table below lists supported performance counters and the corresponding CPUID feature flags.

ECX	Supported Performance Counters	CPUID Feature Flag
0–3	Core performance counters 0–3	All processors
4–5	Core performance counters 4–5	Fn8000_0001_ECX[PerfCtrExtCore] = 1
6–9	Northbridge performance counters 0–3	Fn8000_0001_ECX[PerfCtrExtNB] = 1
10–15	L3 Cache performance counters 0–5	Fn8000_0001_ECX[PerfCtrExtLLC] = 1
16–27	Northbridge performance counters 4–15	Fn8000_0022_EBX[NumPerfCtrNB] > Northbridge performance counter number (4–15).
> 27	Reserved	

For more information on using the CPUID instruction, see the description of the CPUID instruction on page 165.

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.

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

RMPQUERY Read RMP Permissions

Reads an RMP permission mask for a guest page. The guest virtual address is specified in the RAX register. The target VMPL is specified in RDX[7:0]. RMP permissions for the specified VMPL are returned in RDX[63:8] and the RCX register as shown below.

RDX Bits	Field	Description
[63:17]	RESERVED	
[16]	VMSA	VMSA flag This field is always set to 0 unless the current VMPL is 0
[15:8]	TARGET_PERM_MASK	Target VMPL permission mask
[7:0]	TARGET_VMPL	Target VMPL

RCX Bits	Field	Description
[63:1]	RESERVED	
[0]	PAGE_SIZE	Page Size

The RMPQUERY instruction is used by an SNP-active guest to read RMP permissions of a lesser-privileged VMPL. The RMPQUERY instruction will attempt to access the specified page and can 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 and the target VMPL is numerically higher than the current VMPL, the RMPQUERY instruction will read RMP permissions from the RMP entry and return them in RDX and RCX registers.

Upon completion, a return code is stored in EAX. rFLAGS bits OF, ZF, AF, PF and SF are set based on return code.

RMPQUERY performs the same segmentation and paging checks as a 1-byte read at the current VMPL.

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[RMPQUERY] (bit 6) = 1.

Mnemonic	Opcode	Description
RMPQUERY	F3 0F 01 FD	Reads RMP permissions.

Action

```
GUEST_VA = RAX & ~0xFFF
TARGET_VMPL = RDX[7:0]

IF (!64BIT_MODE)           // This instruction is only valid in 64-bit mode
   EXCEPTION [#UD]

IF (!SNP_ACTIVE)
   EXCEPTION [#UD]

IF (CPL != 0)                       // This instruction is only allowed at CPL 0
   EXCEPTION [#GP(0)]

IF (TARGET_VMPL <= CURRENT_VMPL)    // Only permissions for a numerically
   EAX = FAIL_PERMISSION            // higher VMPL can be read
   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 must have an RMP entry
   #VMEXIT(NPF)

temp_RMP = READ_MEM_PA.o [RMP_ENTRY_PA]

IF (!temp_RMP.ASSIGNED || (temp_RMP.GUEST_PA != GUEST_PA) ||
    (temp_RMP.ASID != ASID) || (temp_RMP.PAGE_SIZE != nPT page size))
       #VMEXIT(NPF)

IF (!temp_RMP.VALIDATED)
   #VC(PAGE_NOT_VALIDATED)

RDX[63:16] = 0
RDX[15:8] = temp_RMP.PERMISSIONS[TARGET_VMPL]

IF (CURRENT_VMPL == 0)
   RDX[16] = temp_RMP.VMSA

RCX[63:1] = 0
RCX[0] = temp_RMP.PAGE_SIZE

EAX = SUCCESS
EXIT
```

Return Codes

Value	Name	Description
0	SUCCESS	Successful completion
2	FAIL_PERMISSION	Current ASID not 0 or RMP entry is Immutable

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 RMPQUERY instruction is not supported as indicated by CPUID Fn8000_001F_EAX[RMPQUERY](bit 6)=0
	X	X	X	This instruction is only recognized in 64-bit mode
			X	Guest is not SNP-Active
General Protection, #GP	X	X	X	CPL 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
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
2MB_RMP_ENTRY_PA = RMP_BASE + 0x4000 + (SYSTEM_PA & ~0x1FF000 / 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 (SYSTEM_PA[20:12] != 0)
      2MB_RMP = READ_MEM_PA.o [2MB_RMP_ENTRY_PA]
      IF (2MB_RMP.ASSIGNED && (2MB_RMP.PAGE_SIZE == 2MB))
         EAX = FAIL_OVERLAP
         EXIT
      ELSE IF (Another processor is modifying a page in 2MB region)
         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] = FULL_PERMISSIONS
         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 559 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 597 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 pending interrupts while flushing the caches. If an interrupt is pending, the processor stops flushing the caches, saves the instruction pointer and transfers control to the interrupt handler. Upon returning from the interrupt handler, the processor restarts the flush process from the beginning as lines may have been modified and cached while executing the interrupt handler. Interruptible WBINVD and WBNOINVD support is indicated by CPUID Fn8000_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.

Exception	Real	Virtual 8086	Protected	Cause of Exception
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-writable 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 initial 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 data type is indicated. Operand widths and data types are specified based on the source operands. For instructions where the result overwrites one of the source registers, the data width and data type 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 gray 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 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.

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 112 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 519 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 530							
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 519 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 516 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 519, 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 521. 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 523.

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 preceded 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 521 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	C			CMPSS Vss, Wss, Ib					
66	C			CMPPD Vpd, Wpd, Ib		PINSRW Vo, Ry/Mw, Ib	PEXTRW Gd, Uo, Ib	SHUFPD Vpd, Wpd, Ib	
F2	C			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	D							MOVQ2DQ Vo, Nq	
66	D	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	D	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	E							CVTDQ2PD Vpd, Wpj	
66	E	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	E							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	F								
66	F		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	F	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 preceded 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 521 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 526
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 preceded 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 521 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 preceded 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 521 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 preceded 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 521 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.

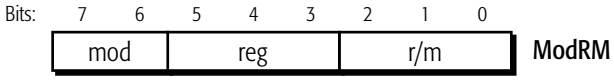


Figure A-1. ModRM-Byte Fields

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 511 and Table A-2 on page 512. 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 521 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 521 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 521 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 523 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 523 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)			RMPQUERY (F3...FD)	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 512.

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 528 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 512. 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 to A-23 below present the VEX opcode maps and Table A-24 on page 546 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 to A-30 below present the XOP opcode maps and Table A-31 on page 548 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 519, 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.

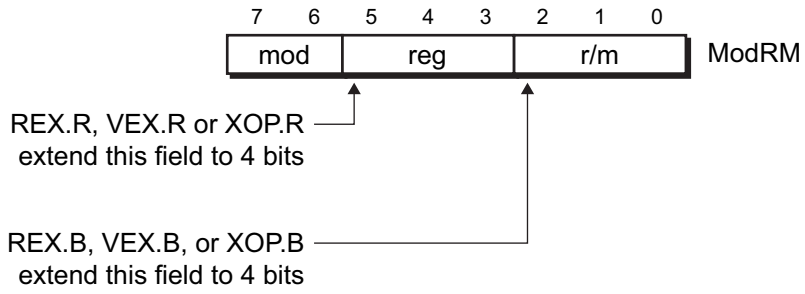


Figure A-2. ModRM-Byte Format

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 552 but applies only when the address-size is 16-bit. Table A-33 on page 550 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 553.

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 552 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 553 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 555 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.

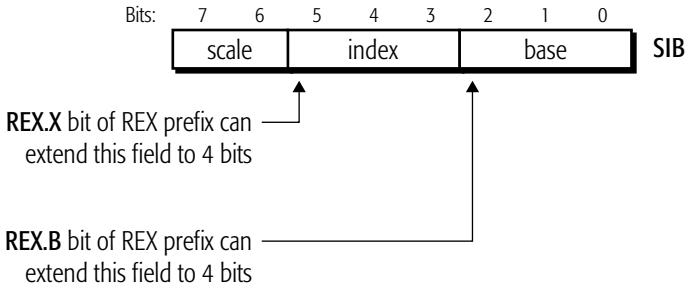


Figure A-3. SIB Byte Format

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 556 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 555 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 555 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 587 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 559.

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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 559, 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 559 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 585 and Table B-3 on page 586)	
		Various opcodes are invalid in long mode (see Table B-4 on page 586)	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 597.

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 593 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.

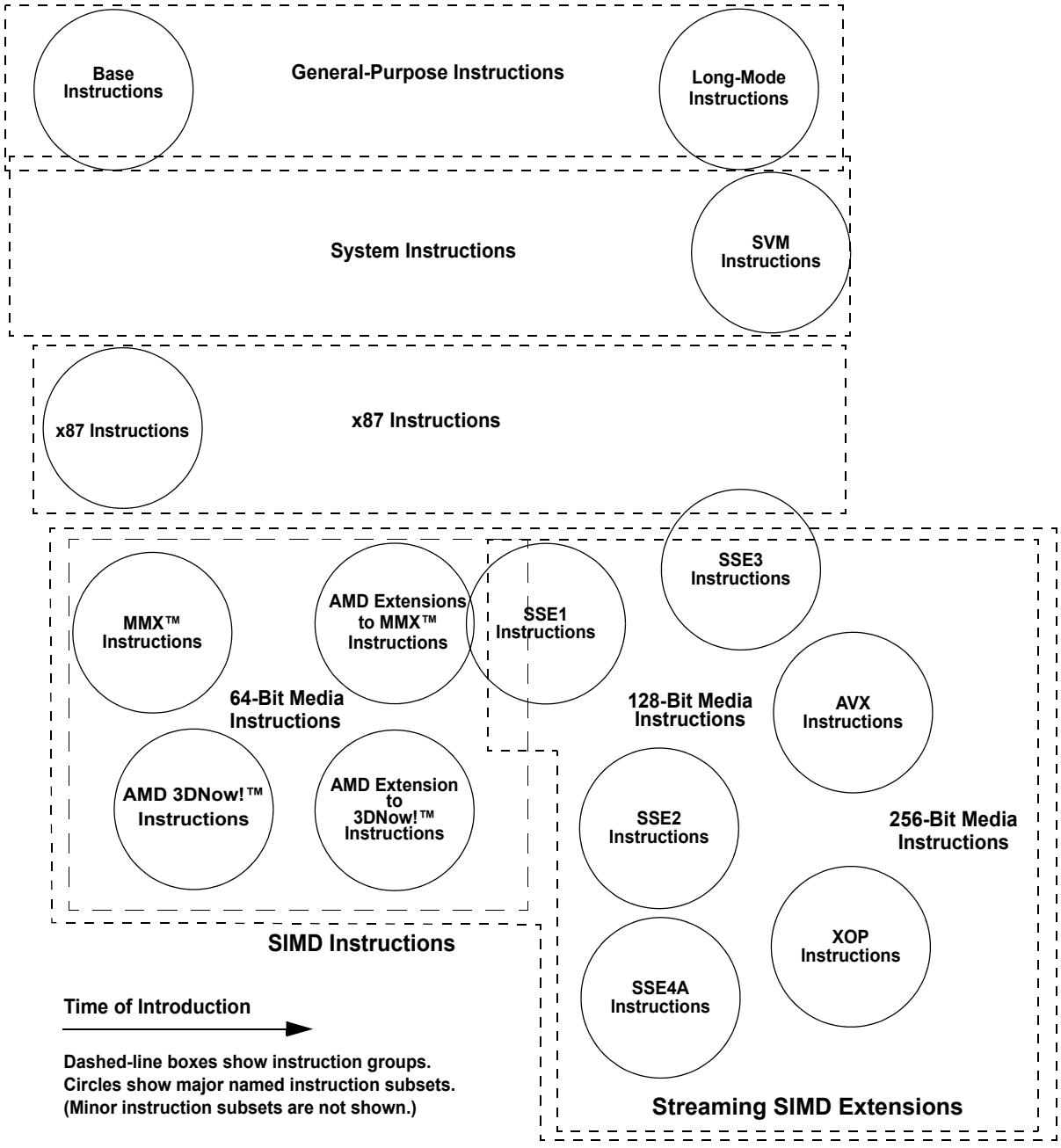


Figure D-1. AMD64 ISA Instruction Subsets

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]
SNP	RMPQUERY	8000_001F	EAX[6]
SSE	SSE1	standard	EDX[25]

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]
SSE2	SSE2	standard	EDX[26]
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 parameter, 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 omitted 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 referring 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.
25	AES	AES instruction support. See “AES Instructions” in APM4.

Bits	Field Name	Description
24	—	Reserved.
23	POPCNT	POPCNT instruction. See “POPCNT” in APM3.
22		MOVBE: MOVBE instruction support.
21	x2APIC	x2APIC support. See “x2APIC Mode” in APM2.
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 640.
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.
18	—	Reserved.

Bits	Field Name	Description
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	Advanced 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.

Bits	Field Name	Description
23	CLFLUSHOPT	CLFLUSHOPT instruction support.
22	RDPID	RDPID instruction and TSC_AUX MSR support.
21	—	Reserved.
20	SMAP	Supervisor mode access prevention.
19	ADX	ADCX, ADOX instruction support.
18	RDSEED	RDSEED instruction support.
17:16	—	Reserved.
15	PQE	Platform QOS Enforcement support. See *AMD64 Technology Platform Quality of Service Extensions*, #56375.
14:13	—	Reserved.
12	PQM	Platform QOS Monitoring support. See *AMD64 Technology Platform Quality of Service Extensions*, #56375.
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
31:17	—	Reserved.
16	LA57	5-Level paging support.
15:11	—	Reserved.
10	VPCMULQDQ	VPCLMULQDQ 256-bit instruction support.
9	VAES	VAES 256-bit instructions support.
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.

Bits	Field Name	Description
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.

Bits	Field Name	Description
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 Function Eh—Reserved

E.3.12 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 598 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 599 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.

Bits	Field Name	Description
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.
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 640.
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.

Bits	Field Name	Description
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.
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 618.

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 618.

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 618.

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 to 5-way associative.
5h	6 to 7-way associative.
6h	8 to 15-way associative.
7h	Permanently reserved
8h	16 to 31-way associative.
9h	Value for all fields should be determined from Fn8000_001D.
Ah	32 to 47-way associative.
Bh	48 to 63-way associative.
Ch	64 to 95-way associative.
Dh	96 to 127-way associative.
Eh	More than 128-way associative but not fully associative.
Fh	Fully associative.

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 620.

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 620.

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 620.
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 620.

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 obtain 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 Extension (MCAX) support. Indicates support for MCAX MSRs. MCA_CONFIG[Mcax] is present in all MCA banks.

2	HWA	Hardware assert support. 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
31:30	—	Reserved
29	BTC_NO	The processor is not affected by branch type confusion
28	PSFD	Predictive Store Forward Disable
27	—	Reserved

Bit	Field Name	Description
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

Bits	Field Name	Description

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 640.

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
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 processor 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
31:27	—	Reserved.
26	IbsVirt	IBS Virtualization. See “Instruction-Based Sampling Virtualization” in Volume 2.
25	VNMI	NMI Virtualization. See “NMI Virtualization” in Volume 2.
24	TlbiCtl	INVLPGB/TLBSYNC hypervisor enable in VMCB and TLBSYNC intercept support.
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	—	Reserved.
21	ROGPT	Read-Only Guest Page Table feature support. See “Nested Table Walk” in Volume 2.
20	SpecCtrl	SPEC_CTRL virtualization.
19	SSSCheck	SVM supervisor shadow stack restrictions. See “Supervisor Shadow Stack Restrictions” in Volume 2.

Bits	Field Name	Description
18	x2AVIC	Support for the AMD advanced virtual interrupt controller for x2APIC mode. See “Advanced Virtual Interrupt Controller” in Volume 2.
17	GMET	Guest Mode Execution Trap.
16	VGIF	Virtualize the Global Interrupt Flag. See “Nested Virtualization” in Volume 2.
15	VMSAVEvirt	VMSAVE and VMLOAD virtualization. See “Nested Virtualization” in Volume 2.
14	—	Reserved.
13	AVIC	Support for the AMD advanced virtual interrupt controller. See “Advanced Virtual Interrupt Controller” in Volume 2.
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” in Volume 2.
9:8	—	Reserved.
7	DecodeAssists	Decode assists. Indicates support for the decode assists. See “Decode Assists” in Volume 2.
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 620.
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 620.
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 620.
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 620.
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 data path is 256 bits wide.
1	MOVU	MOVU SSE instructions 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 data path 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
31:12		Reserved.
11	IbsL3MissFiltering	L3 Miss Filtering for IBS supported. See IBS Filtering in Volume 2.
10:9		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 profiling (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 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
31:30	—	Reserved.
29	NestedVirtSnpMsr	VIRT_RMPUPDATE MSR (C001_F001h) and VIRT_PSMASH MSR (C001_F002h) supported.

Bits	Field Name	Description
28	SvsmCommPageMSR	SVSM Communication Page MSR (C001_F000h) is supported.
27:26	—	Reserved.
25	SmtProtection	SMT Protection supported.
24	VmsaRegProt	VMSA Register Protection supported.
23:20	—	Reserved.
19	IbsVirtGuestCtl	IBS Virtualization supported for SEV-ES guests.
18	VirtualTomMsr	Virtual TOM MSR supported.
17	VmgexitParameter	VMGEXIT Parameter supported.
16	VTE	Virtual Transparent Encryption supported.
15	PreventHostIbs	Disallowing IBS use by the host supported.
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	TscAuxVirtualization	TSC AUX Virtualization supported.
8	SecureTsc	Secure TSC supported.
7	VmplSSS	VMPL Supervisor Shadow Stack supported.
6	RMPQUERY	RMPQUERY Instruction supported
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—Platform QoS Extended Features

CPUID Fn8000_0020_EBX Platform QoS Extended Feature Identifiers (ECX=0)

Bits	Field Name	Description
31:5	—	Reserved.
4	L3RR	L3 Range Reservation is supported. See “L3 Range Reservation” in Volume 2.
3:0	—	Reserved.

E.4.19 Function 8000_0021—Extended Feature Identification 2

CPUID Fn8000_0021_EAX Extended Feature 2

Bits	Field Name	Description
31:18	—	Reserved.
17	CpuidUserDis	CPUID disable for non-privileged software.
16:14	—	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	AutomaticIBRS	Automatic IBRS.
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.

E.4.20 Function 8000_0022—Extended Performance Monitoring and Debug

CPUID Fn8000_0022_EAX Reserved

Bits	Field Name	Description
31:3	—	Reserved.
2	LbrAndPmcFreeze	Freezing Core Performance Counters and LBR Stack on Core Performance Counter overflow supported.
1	LbrStack	Last Branch Record Stack supported.
0	PerfMonV2	Performance Monitoring Version 2 supported. When set, CPUID_Fn8000_0022_EBX reports the number of available performance counters.

CPUID Fn8000_0022_EBX Extended Performance Monitoring and Debug

Bits	Field Name	Description
31:16	—	Reserved.
15:10	NumPerfCtrNB	Number of Northbridge Performance Monitor Counters.
9:4	LbrStackSize	Number of Last Branch Record Stack entries.
3:0	NumPerfCtrCore	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.4.21 Function 8000_0023—Multi-Key Encrypted Memory Capabilities

CPUID Fn8000_0023_EAX Secure Multi-Key Encryption

Bits	Field Name	Description
31:1	—	Reserved.
0	MemHmk	Secure Host Multi-Key Memory (MEM-HMK) Encryption Mode Supported.

CPUID Fn8000_0023_EBX Secure Multi-Key Encryption

Bits	Field Name	Description
31:16	—	Reserved.
15:0	MaxMemHmkEncrKeyID	Number of simultaneously available host encryption key IDs in MEM-HMK encryption mode.

CPUID Fn8000_0023_E[C,D]X Reserved

The values returned in ECX and EDX are undefined and are reserved.

E.4.22 Function 8000_0024—Reserved

E.4.23 Function 8000_0025—Reserved

E.4.24 Function 8000_0026—Extended CPU Topology

CPUID Fn8000_0026 reports extended topology information for logical processors, including asymmetric and heterogenous topology descriptions. Individual logical processors may report different values in systems with asynchronous and heterogeneous topologies.

The topology level is selected by the value passed to the instruction in ECX. To discover the topology of a system, software should execute CPUID Fn8000_0026 with increasing ECX values, starting with a value of zero, until the returned hierarchy level type (CPUID Fn8000_0026_ECX[LevelType]) is equal to zero. It is not guaranteed that all topology level types are present in the system.

Software may use asymmetric and heterogenous indicators reported by CPUID Fn8000_0026_EAX[31:29] for each hierarchy level to efficiently determine system topology. If CPUID Fn8000_0026_EAX[31:29] is equal to zero at a given hierarchy level, all components at this level are symmetric and homogenous. If CPUID Fn8000_0026_EAX[31:29] is not equal to zero, software should use APIC ID, APIC ID mask (derived from CPUID Fn8000_0026_EAX[MaskWidth]), and the number of logical processors at a hierarchy level (CPUID Fn8000_0026[NumLogProc]) to determine asymmetric and heterogenous component properties at this hierarchy level.

CPUID Fn8000_0026_EAX_n[N:0] Extended CPU Topology

Bits	Field Name	Description
31	AsymmetricTopology	Set to 1 if all components at the current hierarchy level do not report the same number of logical processors (NumLogProc).
30	HeterogeneousCores	Set to 1 if all components at the current hierarchy level do not consist of the cores that report the same core type (CoreType).
29	EfficiencyRankingAvailable	Set to 1 if processor power efficiency ranking (PwrEfficiencyRanking) is available and varies between cores. Only valid for LevelType = 1h (Core).
28:5	—	Reserved.
4:0	MaskWidth	Number of bits to shift Extended APIC ID right to get a unique topology ID of the current hierarchy level.

CPUID Fn8000_0026_EBX_n[N:0] Extended CPU Topology

Bits	Field Name	Description
31:28	CoreType	Reports a value that may be used to distinguish between cores with different architectural and microarchitectural properties (for example, cores with different performance or power characteristics). Refer to the *Processor Programming Reference Manual* applicable to your product for a list of the available core types. Only valid for LevelType = 1h (Core).
27:24	NativeModelID	Reports a value that may be used to further differentiate implementation specific features. Native mode ID is used in conjunction with the family, model, and stepping identifiers. Refer to the *Processor Programming Reference Manual* applicable to your product for a list of Native Mode IDs. Only valid for LevelType = 1h (Core).
23:16	PwrEfficiencyRanking	Reports a static efficiency ranking between cores of a specific core type, where a lower value indicates comparatively lower power consumption and lower performance. Only valid for LevelType = 1h (Core).
15:0	NumLogProc	Number of logical processors at the current hierarchy level.

CPUID Fn8000_0026_ECX_n[N:0] Extended CPU Topology

Bits	Field Name	Description
31:16	—	Reserved
15:8	LevelType	Encoded hierarchy level type. **Value** — **Description** 0h — Reserved 1h — Core 2h — Complex 3h — Die 4h — Socket FFh-05h — Reserved

7:0	InputEcx	Input ECX[7:0].

CPUID Fn8000_0026_EDX_n[N:0] Extended CPU Topology

Bits	Field Name	Description
31:0	ExtendedApicId	Extended APIC ID of the logical processor.

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 pack-

age 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	
RMPQUERY									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

Numerics

A

B

C

D

E

F

I

J

M

N

O

P

R

S

T

V

X

Z