

# IA-32 Intel® Architecture Software Developer's Manual

Volume 2B:  
Instruction Set Reference, N-Z

**NOTE:** The IA-32 Intel Architecture Software Developer's Manual consists of five volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-M*, Order Number 253666; *Instruction Set Reference N-Z*, Order Number 253667; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669. Refer to all five volumes when evaluating your design needs.

Order Number: 253667-020US  
June 2006

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS.

Intel may make changes to specifications and product descriptions at any time, without notice.

Developers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Improper use of reserved or undefined features or instructions may cause unpredictable behavior or failure in developer's software code when running on an Intel processor. Intel reserves these features or instructions for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from their unauthorized use.

The Intel® IA-32 architecture processors (e.g., Pentium® 4 and Pentium III processors) may contain design defects or errors known as errata. Current characterized errata are available on request.

Hyper-Threading Technology requires a computer system with an Intel® Pentium® 4 processor supporting Hyper-Threading Technology and an HT Technology enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. See <http://www.intel.com/techtrends/technologies/hyperthreading.htm> for more information including details on which processors support HT Technology.

Intel® Virtualization Technology requires a computer system with an enabled Intel® processor, BIOS, virtual machine monitor (VMM) and for some uses, certain platform software enabled for it. Functionality, performance or other benefits will vary depending on hardware and software configurations and may require a BIOS update. Software applications may not be compatible with all operating systems. Please check with your application vendor.

Intel® Extended Memory 64 Technology (Intel® EM64T) requires a computer system with a processor, chipset, BIOS, OS, device drivers and applications enabled for Intel EM64T. **Processor will not operate (including 32-bit operation) without an Intel EM64T-enabled BIOS.** Performance will vary depending on your hardware and software configurations. **Intel EM64T-enabled OS, BIOS, device drivers and applications may not be available.** Check with your vendor for more information.

Intel, Pentium, Intel Xeon, Intel NetBurst, Intel Core Solo, Intel Core Duo, Intel Pentium D, Itanium, MMX, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

\*Other names and brands may be claimed as the property of others.

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

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

Intel Corporation  
P.O. Box 5937  
Denver, CO 80217-9808

or call 1-800-548-4725  
or visit Intel's website at <http://www.intel.com>

Copyright © 1997-2006 Intel Corporation

# 4

## **Instruction Set Reference, N-Z**



# CHAPTER 4

## INSTRUCTION SET REFERENCE, N-Z

### 4.1 INSTRUCTIONS (N-Z)

Chapter 4 continues the alphabetical discussion of IA-32 instructions (N-Z). See also: Chapter 3, “Instruction Set Reference, A-M,” in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 2A*.

## NEG—Two's Complement Negation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F6 /3	NEG <i>r/m8</i>	Valid	Valid	Two's complement negate <i>r/m8</i> .
REX + F6 /3	NEG <i>r/m8*</i>	Valid	N.E.	Two's complement negate <i>r/m8</i> .
F7 /3	NEG <i>r/m16</i>	Valid	Valid	Two's complement negate <i>r/m16</i> .
F7 /3	NEG <i>r/m32</i>	Valid	Valid	Two's complement negate <i>r/m32</i> .
REX.W + F7 /3	NEG <i>r/m64</i>	Valid	N.E.	Two's complement negate <i>r/m64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.

### Description

Replaces the value of operand (the destination operand) with its two's complement. (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using an REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```
IF DEST = 0
  THEN CF ← 0;
  ELSE CF ← 1;
FI;
DEST ← [– (DEST)]
```

### Flags Affected

The CF flag set to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

### Protected Mode Exceptions

- #GP(0) If the destination is located in a non-writable segment.
- If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## NOP—No Operation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
90	NOP	Valid	Valid	One byte no-operation instruction.
0F 1F /0	NOP r/m16	Valid	Valid	Multi-byte no-operation instruction.
0F 1F /0	NOP r/m32	Valid	Valid	Multi-byte no-operation instruction.

### Description

This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register.

The multi-byte form of NOP is available on processors with model encoding:

- CPUID.01H.EAX[Bytes 11:8] = 0110B or 1111B

The multi-byte NOP instruction does not alter the content of a register and will not issue a memory operation. The instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

The multi-byte NOP instruction performs no operation on supported processors and generates undefined opcode exception on processors that do not support the multi-byte NOP instruction.

The memory operand form of the instruction allows software to create a byte sequence of “no operation” as one instruction. For situations where multiple-byte NOPs are needed, the recommended operations (32-bit mode and 64-bit mode) are:

**Table 4-1. Recommended Multi-Byte Sequence of NOP Instruction**

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H



**Flags Affected**

None.

**Exceptions (All Operating Modes)**

None.

## NOT—One's Complement Negation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F6 /2	NOT <i>r/m8</i>	Valid	Valid	Reverse each bit of <i>r/m8</i> .
REX + F6 /2	NOT <i>r/m8</i> *	Valid	N.E.	Reverse each bit of <i>r/m8</i> .
F7 /2	NOT <i>r/m16</i>	Valid	Valid	Reverse each bit of <i>r/m16</i> .
F7 /2	NOT <i>r/m32</i>	Valid	Valid	Reverse each bit of <i>r/m32</i> .
REX.W + F7 /2	NOT <i>r/m64</i>	Valid	N.E.	Reverse each bit of <i>r/m64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.

### Description

Performs a bitwise NOT operation (each 1 is set to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using an REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

DEST ← NOT DEST;

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

- #GP                    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS                    If a memory operand effective address is outside the SS segment limit.

**Virtual-8086 Mode Exceptions**

- #GP(0)                If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0)                If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code)      If a page fault occurs.
- #AC(0)                If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

- #SS(0)                If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0)                If the memory address is in a non-canonical form.
- #PF(fault-code)      If a page fault occurs.
- #AC(0)                If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## OR—Logical Inclusive OR

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0C <i>ib</i>	OR AL, <i>imm8</i>	Valid	Valid	AL OR <i>imm8</i> .
0D <i>iw</i>	OR AX, <i>imm16</i>	Valid	Valid	AX OR <i>imm16</i> .
0D <i>id</i>	OR EAX, <i>imm32</i>	Valid	Valid	EAX OR <i>imm32</i> .
REX.W + 0D <i>id</i>	OR RAX, <i>imm32</i>	Valid	N.E.	RAX OR <i>imm32</i> ( <i>sign-extended</i> ).
80 /1 <i>ib</i>	OR <i>r/m8</i> , <i>imm8</i>	Valid	Valid	<i>r/m8</i> OR <i>imm8</i> .
REX + 80 /1 <i>ib</i>	OR <i>r/m8*</i> , <i>imm8</i>	Valid	N.E.	<i>r/m8</i> OR <i>imm8</i> .
81 /1 <i>iw</i>	OR <i>r/m16</i> , <i>imm16</i>	Valid	Valid	<i>r/m16</i> OR <i>imm16</i> .
81 /1 <i>id</i>	OR <i>r/m32</i> , <i>imm32</i>	Valid	Valid	<i>r/m32</i> OR <i>imm32</i> .
REX.W + 81 /1 <i>id</i>	OR <i>r/m64</i> , <i>imm32</i>	Valid	N.E.	<i>r/m64</i> OR <i>imm32</i> ( <i>sign-extended</i> ).
83 /1 <i>ib</i>	OR <i>r/m16</i> , <i>imm8</i>	Valid	Valid	<i>r/m16</i> OR <i>imm8</i> ( <i>sign-extended</i> ).
83 /1 <i>ib</i>	OR <i>r/m32</i> , <i>imm8</i>	Valid	Valid	<i>r/m32</i> OR <i>imm8</i> ( <i>sign-extended</i> ).
REX.W + 83 /1 <i>ib</i>	OR <i>r/m64</i> , <i>imm8</i>	Valid	N.E.	<i>r/m64</i> OR <i>imm8</i> ( <i>sign-extended</i> ).
08 /r	OR <i>r/m8</i> , <i>r8</i>	Valid	Valid	<i>r/m8</i> OR <i>r8</i> .
REX + 08 /r	OR <i>r/m8*</i> , <i>r8*</i>	Valid	N.E.	<i>r/m8</i> OR <i>r8</i> .
09 /r	OR <i>r/m16</i> , <i>r16</i>	Valid	Valid	<i>r/m16</i> OR <i>r16</i> .
09 /r	OR <i>r/m32</i> , <i>r32</i>	Valid	Valid	<i>r/m32</i> OR <i>r32</i> .
REX.W + 09 /r	OR <i>r/m64</i> , <i>r64</i>	Valid	N.E.	<i>r/m64</i> OR <i>r64</i> .
0A /r	OR <i>r8</i> , <i>r/m8</i>	Valid	Valid	<i>r8</i> OR <i>r/m8</i> .
REX + 0A /r	OR <i>r8*</i> , <i>r/m8*</i>	Valid	N.E.	<i>r8</i> OR <i>r/m8</i> .
0B /r	OR <i>r16</i> , <i>r/m16</i>	Valid	Valid	<i>r16</i> OR <i>r/m16</i> .
0B /r	OR <i>r32</i> , <i>r/m32</i>	Valid	Valid	<i>r32</i> OR <i>r/m32</i> .
REX.W + 0B /r	OR <i>r64</i> , <i>r/m64</i>	Valid	N.E.	<i>r64</i> OR <i>r/m64</i> .

**NOTES:**

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.

**Description**

Performs a bitwise inclusive OR operation between the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using an REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

DEST ← DEST OR SRC;

## Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

## Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## INSTRUCTION SET REFERENCE, N-Z

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 56 /r	ORPD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Bitwise OR of <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a bitwise logical OR of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[127:0] ← DEST[127:0] BitwiseOR SRC[127:0];

### Intel C/C++ Compiler Intrinsic Equivalent

ORPD            \_\_m128d \_mm\_or\_pd(\_\_m128d a, \_\_m128d b)

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.

## INSTRUCTION SET REFERENCE, N-Z

### Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.



## ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 56 /r	ORPS <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Bitwise OR of <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a bitwise logical OR of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[127:0] ← DEST[127:0] BitwiseOR SRC[127:0];

### Intel C/C++ Compiler Intrinsic Equivalent

ORPS            \_\_m128 \_mm\_or\_ps(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#GP(0)            For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.

If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)            For an illegal address in the SS segment.

#PF(fault-code)    For a page fault.

#NM                If CR0.TS[bit 3] = 1.

#UD                If CR0.EM[bit 2] = 1.

If CR4.OSFXSR[bit 9] = 0.

If CPUID.01H:EDX.SSE[bit 25] = 0.

## INSTRUCTION SET REFERENCE, N-Z

### Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE[bit 25] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.  If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE[bit 25] = 0.

## OUT—Output to Port

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
E6 <i>ib</i>	OUT <i>imm8</i> , AL	Valid	Valid	Output byte in AL to I/O port address <i>imm8</i> .
E7 <i>ib</i>	OUT <i>imm8</i> , AX	Valid	Valid	Output word in AX to I/O port address <i>imm8</i> .
E7 <i>ib</i>	OUT <i>imm8</i> , EAX	Valid	Valid	Output doubleword in EAX to I/O port address <i>imm8</i> .
EE	OUT DX, AL	Valid	Valid	Output byte in AL to I/O port address in DX.
EF	OUT DX, AX	Valid	Valid	Output word in AX to I/O port address in DX.
EF	OUT DX, EAX	Valid	Valid	Output doubleword in EAX to I/O port address in DX.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Description

Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand). The source operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively); the destination operand can be a byte-immediate or the DX register. Using a byte immediate allows I/O port addresses 0 to 255 to be accessed; using the DX register as a source operand allows I/O ports from 0 to 65,535 to be accessed.

The size of the I/O port being accessed is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 13, "Input/Output," in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## INSTRUCTION SET REFERENCE, N-Z

### IA-32 Architecture Compatibility

After executing an OUT instruction, the Pentium processor insures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin; the other IA-32 processors do not.

### Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
  THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
    IF (Any I/O Permission Bit for I/O port being accessed = 1)
      THEN (* I/O operation is not allowed *)
        #GP(0);
      ELSE (* I/O operation is allowed *)
        DEST ← SRC; (* Writes to selected I/O port *)
    FI;
  ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
    DEST ← SRC; (* Writes to selected I/O port *)
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)            If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

#GP(0)            If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.

#PF(fault-code)    If a page fault occurs.

### Compatibility Mode Exceptions

Same as protected mode exceptions.

### 64-Bit Mode Exceptions

Same as protected mode exceptions.

## OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
6E	OUTS DX, <i>m8</i>	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTS DX, <i>m16</i>	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTS DX, <i>m32</i>	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6E	OUTSB	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTSW	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTSD	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

\*\* In 64-bit mode, only 64-bit (RSI) and 32-bit (ESI) address sizes are supported. In non-64-bit mode, only 32-bit (ESI) and 16-bit (SI) address sizes are supported.

### Description

Copies data from the source operand (second operand) to the I/O port specified with the destination operand (first operand). The source operand is a memory location, the address of which is read from either the DS:SI, DS:ESI or the RSI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The DS segment may be overridden with a segment override prefix.) The destination operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the OUTS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand should be a symbol that indicates the size of the I/O port and the source address, and the destination operand must be DX. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the DS:(E)SI or RSI registers, which must be loaded correctly before the OUTS instruction is executed.

## INSTRUCTION SET REFERENCE, N-Z

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the OUTS instructions. Here also DS:(E)SI is assumed to be the source operand and DX is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: OUTSB (byte), OUTSW (word), or OUTSD (doubleword).

After the byte, word, or doubleword is transferred from the memory location to the I/O port, the SI/ESI/RSI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the SI/ESI/RSI register is decremented.) The SI/ESI/RSI register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See “REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix. This instruction is only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 13, “Input/Output,” in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, the default operand size is 32 bits; operand size is not promoted by the use of REX.W. In 64-bit mode, the default address size is 64 bits, and 64-bit address is specified using RSI by default. 32-bit address using ESI is supported using the prefix 67H, but 16-bit address is not supported in 64-bit mode.

### IA-32 Architecture Compatibility

After executing an OUTS, OUTSB, OUTSW, or OUTSD instruction, the Pentium processor insures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin; the other IA-32 processors do not. For the Pentium 4, Intel® Xeon®, and P6 processor family, upon execution of an OUTS, OUTSB, OUTSW, or OUTSD instruction, the processor will not execute the next instruction until the data phase of the transaction is complete.

**Operation**

```

IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
  THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
    IF (Any I/O Permission Bit for I/O port being accessed = 1)
      THEN (* I/O operation is not allowed *)
        #GP(0);
      ELSE (* I/O operation is allowed *)
        DEST ← SRC; (* Writes to I/O port *)
    FI;
  ELSE (Real Mode or Protected Mode or 64-Bit Mode with CPL ≤ IOPL *)
    DEST ← SRC; (* Writes to I/O port *)
FI;

```

Byte transfer:

```

IF 64-bit mode
  Then
    IF 64-Bit Address Size
      THEN
        IF DF = 0
          THEN RSI ← RSI + 1;
          ELSE RSI ← RSI - 1;
        FI;
      ELSE (* 32-Bit Address Size *)
        IF DF = 0
          THEN ESI ← ESI + 1;
          ELSE ESI ← ESI - 1;
        FI;
      FI;
    ELSE
      IF DF = 0
        THEN (E)SI ← (E)SI + 1;
        ELSE (E)SI ← (E)SI - 1;
      FI;
    FI;

```

Word transfer:

```

IF 64-bit mode
  Then
    IF 64-Bit Address Size
      THEN
        IF DF = 0
          THEN RSI ← RSI + 2;
          ELSE RSI ← RSI - 2;
        FI;
      ELSE (* 32-Bit Address Size *)
        IF DF = 0
          THEN ESI ← ESI + 2;
          ELSE ESI ← ESI - 2;
        FI;
      FI;

```

## INSTRUCTION SET REFERENCE, N-Z

```

    FI;
    FI;
ELSE
    IF DF = 0
        THEN (E)SI ← (E)SI + 2;
        ELSE (E)SI ← (E)SI - 2;
    FI;
FI;
Doubleword transfer:
IF 64-bit mode
    Then
        IF 64-Bit Address Size
            THEN
                IF DF = 0
                    THEN RSI ← RSI + 4;
                    ELSE RSI ← RSI - 4;
                FI;
            ELSE (* 32-Bit Address Size *)
                IF DF = 0
                    THEN ESI ← ESI + 4;
                    ELSE ESI ← ESI - 4;
                FI;
        FI;
    ELSE
        IF DF = 0
            THEN (E)SI ← (E)SI + 4;
            ELSE (E)SI ← (E)SI - 4;
        FI;
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
- If a memory operand effective address is outside the limit of the CS, DS, ES, FS, or GS segment.
- If the segment register contains a NULL segment selector.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



**Real-Address Mode Exceptions**

- #GP                    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS                    If a memory operand effective address is outside the SS segment limit.

**Virtual-8086 Mode Exceptions**

- #GP(0)                If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
- #PF(fault-code)      If a page fault occurs.
- #AC(0)                If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

- #SS(0)                If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0)                If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
- If the memory address is in a non-canonical form.
- #PF(fault-code)      If a page fault occurs.
- #AC(0)                If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 63 /r	PACKSSWB <i>mm1</i> , <i>mm2/m64</i>	Valid	Valid	Converts 4 packed signed word integers from <i>mm1</i> and from <i>mm2/m64</i> into 8 packed signed byte integers in <i>mm1</i> using signed saturation.
66 0F 63 /r	PACKSSWB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Converts 8 packed signed word integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.
0F 6B /r	PACKSSDW <i>mm1</i> , <i>mm2/m64</i>	Valid	Valid	Converts 2 packed signed doubleword integers from <i>mm1</i> and from <i>mm2/m64</i> into 4 packed signed word integers in <i>mm1</i> using signed saturation.
66 0F 6B /r	PACKSSDW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Converts 4 packed signed doubleword integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.

### Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-1 for an example of the packing operation.

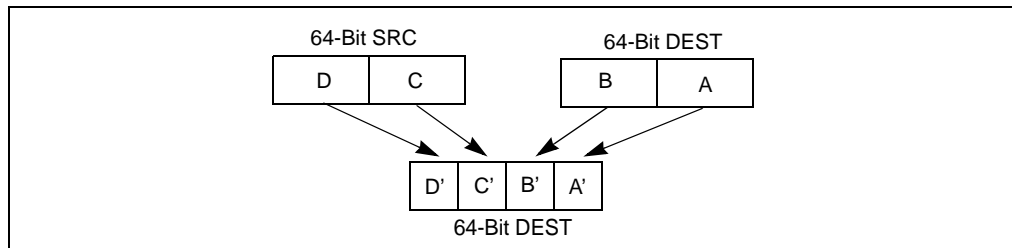


Figure 4-1. Operation of the PACKSSDW Instruction Using 64-bit Operands

The PACKSSWB instruction converts 4 or 8 signed word integers from the destination operand (first operand) and 4 or 8 signed word integers from the source operand (second operand) into 8 or 16 signed byte integers and stores the result in the destination operand. If a signed word integer value is beyond the range of a signed byte integer (that is, greater than 7FH for a positive integer or greater than 80H for a negative integer), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination.

The PACKSSDW instruction packs 2 or 4 signed doublewords from the destination operand (first operand) and 2 or 4 signed doublewords from the source operand (second operand) into 4 or 8 signed words in the destination operand (see Figure 4-1). If a signed doubleword integer value is beyond the range of a signed word (that is, greater than 7FFFH for a positive integer or greater than 8000H for a negative integer), the saturated signed word integer value of 7FFFH or 8000H, respectively, is stored into the destination.

The PACKSSWB and PACKSSDW instructions operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

PACKSSWB instruction with 64-bit operands:

```
DEST[7:0] ← SaturateSignedWordToSignedByte DEST[15:0];
DEST[15:8] ← SaturateSignedWordToSignedByte DEST[31:16];
DEST[23:16] ← SaturateSignedWordToSignedByte DEST[47:32];
DEST[31:24] ← SaturateSignedWordToSignedByte DEST[63:48];
DEST[39:32] ← SaturateSignedWordToSignedByte SRC[15:0];
DEST[47:40] ← SaturateSignedWordToSignedByte SRC[31:16];
DEST[55:48] ← SaturateSignedWordToSignedByte SRC[47:32];
DEST[63:56] ← SaturateSignedWordToSignedByte SRC[63:48];
```

PACKSSDW instruction with 64-bit operands:

```
DEST[15:0] ← SaturateSignedDoublewordToSignedWord DEST[31:0];
DEST[31:16] ← SaturateSignedDoublewordToSignedWord DEST[63:32];
DEST[47:32] ← SaturateSignedDoublewordToSignedWord SRC[31:0];
DEST[63:48] ← SaturateSignedDoublewordToSignedWord SRC[63:32];
```

PACKSSWB instruction with 128-bit operands:

```
DEST[7:0] ← SaturateSignedWordToSignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (DEST[127:112]);
```

## INSTRUCTION SET REFERENCE, N-Z

DEST[71:64] ← SaturateSignedWordToSignedByte (SRC[15:0]);  
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC[31:16]);  
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC[47:32]);  
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC[63:48]);  
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC[79:64]);  
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC[95:80]);  
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC[111:96]);  
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC[127:112]);

PACKSSDW instruction with 128-bit operands:

DEST[15:0] ← SaturateSignedDwordToSignedWord (DEST[31:0]);  
DEST[31:16] ← SaturateSignedDwordToSignedWord (DEST[63:32]);  
DEST[47:32] ← SaturateSignedDwordToSignedWord (DEST[95:64]);  
DEST[63:48] ← SaturateSignedDwordToSignedWord (DEST[127:96]);  
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC[31:0]);  
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC[63:32]);  
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC[95:64]);  
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC[127:96]);

### Intel C/C++ Compiler Intrinsic Equivalents

PACKSSWB `__m64 _mm_packs_pi16(__m64 m1, __m64 m2)`

PACKSSDW `__m64 _mm_packs_pi32(__m64 m1, __m64 m2)`

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.  (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PACKUSWB—Pack with Unsigned Saturation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 67 /r	PACKUSWB <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Converts 4 signed word integers from <i>mm</i> and 4 signed word integers from <i>mm/m64</i> into 8 unsigned byte integers in <i>mm</i> using unsigned saturation.
66 0F 67 /r	PACKUSWB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Converts 8 signed word integers from <i>xmm1</i> and 8 signed word integers from <i>xmm2/m128</i> into 16 unsigned byte integers in <i>xmm1</i> using unsigned saturation.

### Description

Converts 4 or 8 signed word integers from the destination operand (first operand) and 4 or 8 signed word integers from the source operand (second operand) into 8 or 16 unsigned byte integers and stores the result in the destination operand. (See Figure 4-1 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

The PACKUSWB instruction operates on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PACKUSWB instruction with 64-bit operands:

```
DEST[7:0] ← SaturateSignedWordToUnsignedByte DEST[15:0];
DEST[15:8] ← SaturateSignedWordToUnsignedByte DEST[31:16];
DEST[23:16] ← SaturateSignedWordToUnsignedByte DEST[47:32];
DEST[31:24] ← SaturateSignedWordToUnsignedByte DEST[63:48];
DEST[39:32] ← SaturateSignedWordToUnsignedByte SRC[15:0];
DEST[47:40] ← SaturateSignedWordToUnsignedByte SRC[31:16];
DEST[55:48] ← SaturateSignedWordToUnsignedByte SRC[47:32];
DEST[63:56] ← SaturateSignedWordToUnsignedByte SRC[63:48];
```

PACKUSWB instruction with 128-bit operands:

```
DEST[7:0] ← SaturateSignedWordToUnsignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToUnsignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToUnsignedByte (DEST[47:32]);
```

DEST[31:24] ← SaturateSignedWordToUnsignedByte (DEST[63:48]);  
 DEST[39:32] ← SaturateSignedWordToUnsignedByte (DEST[79:64]);  
 DEST[47:40] ← SaturateSignedWordToUnsignedByte (DEST[95:80]);  
 DEST[55:48] ← SaturateSignedWordToUnsignedByte (DEST[111:96]);  
 DEST[63:56] ← SaturateSignedWordToUnsignedByte (DEST[127:112]);  
 DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC[15:0]);  
 DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC[31:16]);  
 DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC[47:32]);  
 DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC[63:48]);  
 DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC[79:64]);  
 DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC[95:80]);  
 DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC[111:96]);  
 DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC[127:112]);

### Intel C/C++ Compiler Intrinsic Equivalent

PACKUSWB \_\_m64 \_\_mm\_packs\_pu16(\_\_m64 m1, \_\_m64 m2)

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #UD If CR0.EM[bit 2] = 1.  
  
128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
- #NM If CR0.TS[bit 3] = 1.
- #MF (64-bit operations only) If there is a pending x87 FPU exception.
- #PF(fault-code) If a page fault occurs.
- #AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## INSTRUCTION SET REFERENCE, N-Z

### Real-Address Mode Exceptions

- #GP(0) (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  
If any part of the operand lies outside of the effective address space from 0 to FFFFH.
- #UD If CR0.EM[bit 2] = 1.  
128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
- #NM If CR0.TS[bit 3] = 1.
- #MF (64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

- #PF(fault-code) For a page fault.
- #AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.



**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PADDB/PADDW/PADDD—Add Packed Integers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F FC /r	PADDB <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Add packed byte integers from <i>mm/m64</i> and <i>mm</i> .
66 0F FC /r	PADDB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Add packed byte integers from <i>xmm2/m128</i> and <i>xmm1</i> .
0F FD /r	PADDW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Add packed word integers from <i>mm/m64</i> and <i>mm</i> .
66 0F FD /r	PADDW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Add packed word integers from <i>xmm2/m128</i> and <i>xmm1</i> .
0F FE /r	PADDD <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Add packed doubleword integers from <i>mm/m64</i> and <i>mm</i> .
66 0F FE /r	PADDD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Add packed doubleword integers from <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs an SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of an SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PADDB instruction adds packed byte integers. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW instruction adds packed word integers. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand.

The PADDD instruction adds packed doubleword integers. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand.

Note that the PADDB, PADDW, and PADDD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

PADDB instruction with 64-bit operands:

DEST[7:0] ← DEST[7:0] + SRC[7:0];  
 (\* Repeat add operation for 2nd through 7th byte \*)  
 DEST[63:56] ← DEST[63:56] + SRC[63:56];

PADDB instruction with 128-bit operands:

DEST[7:0] ← DEST[7:0] + SRC[7:0];  
 (\* Repeat add operation for 2nd through 14th byte \*)  
 DEST[127:120] ← DEST[111:120] + SRC[127:120];

PADDW instruction with 64-bit operands:

DEST[15:0] ← DEST[15:0] + SRC[15:0];  
 (\* Repeat add operation for 2nd and 3th word \*)  
 DEST[63:48] ← DEST[63:48] + SRC[63:48];

PADDW instruction with 128-bit operands:

DEST[15:0] ← DEST[15:0] + SRC[15:0];  
 (\* Repeat add operation for 2nd through 7th word \*)  
 DEST[127:112] ← DEST[127:112] + SRC[127:112];

PADDD instruction with 64-bit operands:

DEST[31:0] ← DEST[31:0] + SRC[31:0];  
 DEST[63:32] ← DEST[63:32] + SRC[63:32];

PADDD instruction with 128-bit operands:

DEST[31:0] ← DEST[31:0] + SRC[31:0];  
 (\* Repeat add operation for 2nd and 3th doubleword \*)  
 DEST[127:96] ← DEST[127:96] + SRC[127:96];

## Intel C/C++ Compiler Intrinsic Equivalents

PADDB	__m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDB	__m128i _mm_add_epi8 (__m128ia, __m128ib )
PADDW	__m64 _mm_addw_pi16(__m64 m1, __m64 m2)
PADDW	__m128i _mm_add_epi16 ( __m128i a, __m128i b)
PADDD	__m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDD	__m128i _mm_add_epi32 ( __m128i a, __m128i b)

## Flags Affected

None.

## INSTRUCTION SET REFERENCE, N-Z

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PADDQ—Add Packed Quadword Integers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F D4 /r	PADDQ <i>mm1</i> , <i>mm2/m64</i>	Valid	Valid	Add quadword integer <i>mm2/m64</i> to <i>mm1</i> .
66 0F D4 /r	PADDQ <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Add packed quadword integers <i>xmm2/m128</i> to <i>xmm1</i> .

### Description

Adds the first operand (destination operand) to the second operand (source operand) and stores the result in the destination operand. The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location, or it can be two packed quadword integers stored in an XMM register or an 128-bit memory location. The destination operand can be a quadword integer stored in an MMX technology register or two packed quadword integers stored in an XMM register. When packed quadword operands are used, an SIMD add is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the PADDQ instruction can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PADDQ instruction with 64-Bit operands:

$$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] + \text{SRC}[63:0];$$

PADDQ instruction with 128-Bit operands:

$$\begin{aligned} \text{DEST}[63:0] &\leftarrow \text{DEST}[63:0] + \text{SRC}[63:0]; \\ \text{DEST}[127:64] &\leftarrow \text{DEST}[127:64] + \text{SRC}[127:64]; \end{aligned}$$

### Intel C/C++ Compiler Intrinsic Equivalents

PADDQ            `__m64 _mm_add_si64 (__m64 a, __m64 b)`

PADDQ            `__m128i _mm_add_epi64 (__m128i a, __m128i b)`

### Flags Affected

None.

**Numeric Exceptions**

None.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

## INSTRUCTION SET REFERENCE, N-Z

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F EC /r	PADDSB <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Add packed signed byte integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F EC /r	PADDSB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Add packed signed byte integers from <i>xmm2/m128</i> and <i>xmm1</i> and saturate the results.
0F ED /r	PADDSW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Add packed signed word integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F ED /r	PADDSW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Add packed signed word integers from <i>xmm2/m128</i> and <i>xmm1</i> and saturate the results.

### Description

Performs an SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of an SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PADDSB instruction adds packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The PADDSW instruction adds packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## INSTRUCTION SET REFERENCE, N-Z

### Operation

PADDSB instruction with 64-bit operands:

DEST[7:0] ← SaturateToSignedByte(DEST[7:0] + SRC(7:0));  
(\* Repeat add operation for 2nd through 7th bytes \*)  
DEST[63:56] ← SaturateToSignedByte(DEST[63:56] + SRC[63:56]);

PADDSB instruction with 128-bit operands:

DEST[7:0] ← SaturateToSignedByte (DEST[7:0] + SRC[7:0]);  
(\* Repeat add operation for 2nd through 14th bytes \*)  
DEST[127:120] ← SaturateToSignedByte (DEST[111:120] + SRC[127:120]);

PADDSW instruction with 64-bit operands

DEST[15:0] ← SaturateToSignedWord(DEST[15:0] + SRC[15:0]);  
(\* Repeat add operation for 2nd and 7th words \*)  
DEST[63:48] ← SaturateToSignedWord(DEST[63:48] + SRC[63:48]);

PADDSW instruction with 128-bit operands

DEST[15:0] ← SaturateToSignedWord (DEST[15:0] + SRC[15:0]);  
(\* Repeat add operation for 2nd through 7th words \*)  
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] + SRC[127:112]);

### Intel C/C++ Compiler Intrinsic Equivalents

PADDSB     \_\_m64 \_\_mm\_adds\_pi8(\_\_m64 m1, \_\_m64 m2)  
PADDSB     \_\_m128i \_\_mm\_adds\_epi8 ( \_\_m128i a, \_\_m128i b)  
PADDSW     \_\_m64 \_\_mm\_adds\_pi16(\_\_m64 m1, \_\_m64 m2)  
PADDSW     \_\_m128i \_\_mm\_adds\_epi16 ( \_\_m128i a, \_\_m128i b)

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)     If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
  
(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  
  
#SS(0)     If a memory operand effective address is outside the SS segment limit.  
  
#UD         If CR0.EM[bit 2] = 1.  
  
128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0.  
Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.  
  
#NM         If CR0.TS[bit 3] = 1.

#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.  (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

## INSTRUCTION SET REFERENCE, N-Z

#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F DC /r	PADDUSB <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Add packed unsigned byte integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F DC /r	PADDUSB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Add packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> saturate the results.
0F DD /r	PADDUSW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Add packed unsigned word integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 0F DD /r	PADDUSW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Add packed unsigned word integers from <i>xmm2/m128</i> to <i>xmm1</i> and saturate the results.

### Description

Performs an SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of an SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PADDUSB instruction adds packed unsigned byte integers. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

The PADDUSW instruction adds packed unsigned word integers. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## INSTRUCTION SET REFERENCE, N-Z

### Operation

PADDUSB instruction with 64-bit operands:

$DEST[7:0] \leftarrow \text{SaturateToUnsignedByte}(DEST[7:0] + SRC(7:0));$   
(\* Repeat add operation for 2nd through 7th bytes \*)  
 $DEST[63:56] \leftarrow \text{SaturateToUnsignedByte}(DEST[63:56] + SRC[63:56])$

PADDUSB instruction with 128-bit operands:

$DEST[7:0] \leftarrow \text{SaturateToUnsignedByte}(DEST[7:0] + SRC[7:0]);$   
(\* Repeat add operation for 2nd through 14th bytes \*)  
 $DEST[127:120] \leftarrow \text{SaturateToUnsignedByte}(DEST[127:120] + SRC[127:120]);$

PADDUSW instruction with 64-bit operands:

$DEST[15:0] \leftarrow \text{SaturateToUnsignedWord}(DEST[15:0] + SRC[15:0]);$   
(\* Repeat add operation for 2nd and 3rd words \*)  
 $DEST[63:48] \leftarrow \text{SaturateToUnsignedWord}(DEST[63:48] + SRC[63:48]);$

PADDUSW instruction with 128-bit operands:

$DEST[15:0] \leftarrow \text{SaturateToUnsignedWord}(DEST[15:0] + SRC[15:0]);$   
(\* Repeat add operation for 2nd through 7th words \*)  
 $DEST[127:112] \leftarrow \text{SaturateToUnsignedWord}(DEST[127:112] + SRC[127:112]);$

### Intel C/C++ Compiler Intrinsic Equivalents

PADDUSB     \_\_m64 \_mm\_adds\_pu8(\_\_m64 m1, \_\_m64 m2)  
PADDUSW     \_\_m64 \_mm\_adds\_pu16(\_\_m64 m1, \_\_m64 m2)  
PADDUSB     \_\_m128i \_mm\_adds\_epu8 (\_\_m128i a, \_\_m128i b)  
PADDUSW     \_\_m128i \_mm\_adds\_epu16 (\_\_m128i a, \_\_m128i b)

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

- #GP(0)           If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
  
(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
- #SS(0)           If a memory operand effective address is outside the SS segment limit.

#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## PAND—Logical AND

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F DB /r	PAND <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Bitwise AND <i>mm/m64</i> and <i>mm</i> .
66 0F DB /r	PAND <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Bitwise AND of <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a bitwise logical AND operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST ← (DEST AND SRC);

### Intel C/C++ Compiler Intrinsic Equivalent

PAND            \_\_m64 \_mm\_and\_si64 ( \_\_m64 m1, \_\_m64 m2)

PAND            \_\_m128i \_mm\_and\_si128 ( \_\_m128i a, \_\_m128i b)

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

- #GP(0)            If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
- #SS(0)            If a memory operand effective address is outside the SS segment limit.

## INSTRUCTION SET REFERENCE, N-Z

#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**PANDN—Logical AND NOT**

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F DF /r	PANDN <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Bitwise AND NOT of <i>mm/m64</i> and <i>mm</i> .
66 0F DF /r	PANDN <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Bitwise AND NOT of <i>xmm2/m128</i> and <i>xmm1</i> .

**Description**

Performs a bitwise logical NOT of the destination operand (first operand), then performs a bitwise logical AND of the source operand (second operand) and the inverted destination operand. The result is stored in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1; otherwise, it is set to 0.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Operation**

DEST ← ((NOT DEST) AND SRC);

**Intel C/C++ Compiler Intrinsic Equivalent**

PANDN `__m64 _mm_andnot_si64 (__m64 m1, __m64 m2)`

PANDN `_m128i _mm_andnot_si128 (__m128i a, __m128i b)`

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Protected Mode Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PAUSE—Spin Loop Hint

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 90	PAUSE	Valid	Valid	Gives hint to processor that improves performance of spin-wait loops.

### Description

Improves the performance of spin-wait loops. When executing a “spin-wait loop,” a Pentium 4 or Intel Xeon processor suffers a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.

An additional function of the PAUSE instruction is to reduce the power consumed by a Pentium 4 processor while executing a spin loop. The Pentium 4 processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spin-wait loop greatly reduces the processor’s power consumption.

This instruction was introduced in the Pentium 4 processors, but is backward compatible with all IA-32 processors. In earlier IA-32 processors, the PAUSE instruction operates like a NOP instruction. The Pentium 4 and Intel Xeon processors implement the PAUSE instruction as a pre-defined delay. The delay is finite and can be zero for some processors. This instruction does not change the architectural state of the processor (that is, it performs essentially a delaying no-op operation).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

Execute\_Next\_Instruction(Delay);

### Numeric Exceptions

None.

### Exceptions (All Operating Modes)

None.

## PAVGB/PAVGW—Average Packed Integers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F E0 /r	PAVGB <i>mm1</i> , <i>mm2/m64</i>	Valid	Valid	Average packed unsigned byte integers from <i>mm2/m64</i> and <i>mm1</i> with rounding.
66 0F E0, /r	PAVGB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Average packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> with rounding.
0F E3 /r	PAVGW <i>mm1</i> , <i>mm2/m64</i>	Valid	Valid	Average packed unsigned word integers from <i>mm2/m64</i> and <i>mm1</i> with rounding.
66 0F E3 /r	PAVGW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Average packed unsigned word integers from <i>xmm2/m128</i> and <i>xmm1</i> with rounding.

### Description

Performs an SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

The PAVGB instruction operates on packed unsigned bytes and the PAVGW instruction operates on packed unsigned words.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PAVGB instruction with 64-bit operands:

$$\text{SRC}[7:0] \leftarrow (\text{SRC}[7:0] + \text{DEST}[7:0] + 1) \gg 1; \text{ (* Temp sum before shifting is 9 bits *)}$$

(\* Repeat operation performed for bytes 2 through 6 \*)

$$\text{SRC}[63:56] \leftarrow (\text{SRC}[63:56] + \text{DEST}[63:56] + 1) \gg 1;$$

PAVGW instruction with 64-bit operands:

$$\text{SRC}[15:0] \leftarrow (\text{SRC}[15:0] + \text{DEST}[15:0] + 1) \gg 1; \text{ (* Temp sum before shifting is 17 bits *)}$$

(\* Repeat operation performed for words 2 and 3 \*)

$$\text{SRC}[63:48] \leftarrow (\text{SRC}[63:48] + \text{DEST}[63:48] + 1) \gg 1;$$

PAVGB instruction with 128-bit operands:

$$\text{SRC}[7:0] \leftarrow (\text{SRC}[7:0] + \text{DEST}[7:0] + 1) \gg 1; \text{ (* Temp sum before shifting is 9 bits *)}$$

(\* Repeat operation performed for bytes 2 through 14 \*)

$$\text{SRC}[63:56] \leftarrow (\text{SRC}[63:56] + \text{DEST}[63:56] + 1) \gg 1;$$

PAVGW instruction with 128-bit operands:



$SRC[15:0] \leftarrow (SRC[15:0] + DEST[15:0] + 1) \gg 1$ ; (\* Temp sum before shifting is 17 bits \*)  
 (\* Repeat operation performed for words 2 through 6 \*)  
 $SRC[127:48] \leftarrow (SRC[127:112] + DEST[127:112] + 1) \gg 1$ ;

### Intel C/C++ Compiler Intrinsic Equivalent

PAVGB        \_\_m64\_mm\_avg\_pu8 (\_\_m64 a, \_\_m64 b)  
 PAVGW        \_\_m64\_mm\_avg\_pu16 (\_\_m64 a, \_\_m64 b)  
 PAVGB        \_\_m128i\_mm\_avg\_epu8 (\_\_m128i a, \_\_m128i b)  
 PAVGW        \_\_m128i\_mm\_avg\_epu16 (\_\_m128i a, \_\_m128i b)

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

**#GP(0)**        If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
                   (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  
**#SS(0)**        If a memory operand effective address is outside the SS segment limit.  
**#UD**            If CR0.EM[bit 2] = 1.  
                   (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  
                   (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.  
**#NM**            If CR0.TS[bit 3] = 1.  
**#MF**            (64-bit operations only) If there is a pending x87 FPU exception.  
**#PF(fault-code)** If a page fault occurs.  
**#AC(0)**        (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

**#GP(0)**        (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  
                   If any part of the operand lies outside of the effective address space from 0 to FFFFH.

## INSTRUCTION SET REFERENCE, N-Z

#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 74 /r	PCMPEQB <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Compare packed bytes in <i>mm/m64</i> and <i>mm</i> for equality.
66 0F 74 /r	PCMPEQB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Compare packed bytes in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
0F 75 /r	PCMPEQW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Compare packed words in <i>mm/m64</i> and <i>mm</i> for equality.
66 0F 75 /r	PCMPEQW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Compare packed words in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
0F 76 /r	PCMPEQD <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Compare packed doublewords in <i>mm/m64</i> and <i>mm</i> for equality.
66 0F 76 /r	PCMPEQD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Compare packed doublewords in <i>xmm2/m128</i> and <i>xmm1</i> for equality.

### Description

Performs an SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

The PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the PCMPEQW instruction compares the corresponding words in the destination and source operands; and the PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PCMPEQB instruction with 64-bit operands:

IF DEST[7:0] = SRC[7:0]

THEN DEST[7:0] ← FFH;

ELSE DEST[7:0] ← 0; FI;

(\* Continue comparison of 2nd through 7th bytes in DEST and SRC \*)

IF DEST[63:56] = SRC[63:56]

THEN DEST[63:56] ← FFH;

ELSE DEST[63:56] ← 0; FI;

## INSTRUCTION SET REFERENCE, N-Z

PCMPEQB instruction with 128-bit operands:

```
IF DEST[7:0] = SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
IF DEST[63:56] = SRC[63:56]
    THEN DEST[63:56] ← FFH;
    ELSE DEST[63:56] ← 0; FI;
```

PCMPEQW instruction with 64-bit operands:

```
IF DEST[15:0] = SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd words in DEST and SRC *)
IF DEST[63:48] = SRC[63:48]
    THEN DEST[63:48] ← FFFFH;
    ELSE DEST[63:48] ← 0; FI;
```

PCMPEQW instruction with 128-bit operands:

```
IF DEST[15:0] = SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd through 7th words in DEST and SRC *)
IF DEST[63:48] = SRC[63:48]
    THEN DEST[63:48] ← FFFFH;
    ELSE DEST[63:48] ← 0; FI;
```

PCMPEQD instruction with 64-bit operands:

```
IF DEST[31:0] = SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
IF DEST[63:32] = SRC[63:32]
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 0; FI;
```

PCMPEQD instruction with 128-bit operands:

```
IF DEST[31:0] = SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
IF DEST[63:32] = SRC[63:32]
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 0; FI;
```

### Intel C/C++ Compiler Intrinsic Equivalents

```
PCMPEQB    __m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)
PCMPEQW    __m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD    __m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)
```

PCMPEQB    \_\_m128i \_\_mm\_cmpeq\_epi8 ( \_\_m128i a, \_\_m128i b)  
 PCMPEQW    \_\_m128i \_\_mm\_cmpeq\_epi16 ( \_\_m128i a, \_\_m128i b)  
 PCMPEQD    \_\_m128i \_\_mm\_cmpeq\_epi32 ( \_\_m128i a, \_\_m128i b)

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)       If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
               (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)       If a memory operand effective address is outside the SS segment limit.

#UD           If CR0.EM[bit 2] = 1.  
               128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.

#NM           If CR0.TS[bit 3] = 1.

#MF           (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code)   If a page fault occurs.

#AC(0)       (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP(0)       (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  
               If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD           If CR0.EM[bit 2] = 1.  
               128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.

#NM           If CR0.TS[bit 3] = 1.

#MF           (64-bit operations only) If there is a pending x87 FPU exception.

## INSTRUCTION SET REFERENCE, N-Z

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#UD If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 64 /r	PCMPGTB <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Compare packed signed byte integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 0F 64 /r	PCMPGTB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Compare packed signed byte integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
0F 65 /r	PCMPGTW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Compare packed signed word integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 0F 65 /r	PCMPGTW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Compare packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
0F 66 /r	PCMPGTD <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Compare packed signed doubleword integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 0F 66 /r	PCMPGTD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Compare packed signed doubleword integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.

### Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding data element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## INSTRUCTION SET REFERENCE, N-Z

### Operation

PCMPGTB instruction with 64-bit operands:

```
IF DEST[7:0] > SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
IF DEST[63:56] > SRC[63:56]
    THEN DEST[63:56] ← FFH;
    ELSE DEST[63:56] ← 0; FI;
```

PCMPGTB instruction with 128-bit operands:

```
IF DEST[7:0] > SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
IF DEST[63:56] > SRC[63:56]
    THEN DEST[63:56] ← FFH;
    ELSE DEST[63:56] ← 0; FI;
```

PCMPGTW instruction with 64-bit operands:

```
IF DEST[15:0] > SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd words in DEST and SRC *)
IF DEST[63:48] > SRC[63:48]
    THEN DEST[63:48] ← FFFFH;
    ELSE DEST[63:48] ← 0; FI;
```

PCMPGTW instruction with 128-bit operands:

```
IF DEST[15:0] > SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd through 7th words in DEST and SRC *)
IF DEST[63:48] > SRC[63:48]
    THEN DEST[63:48] ← FFFFH;
    ELSE DEST[63:48] ← 0; FI;
```

PCMPGTD instruction with 64-bit operands:

```
IF DEST[31:0] > SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
IF DEST[63:32] > SRC[63:32]
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 0; FI;
```

PCMPGTD instruction with 128-bit operands:

```
IF DEST[31:0] > SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
```



```
IF DEST[63:32] > SRC[63:32]
THEN DEST[63:32] ← FFFFFFFFH;
ELSE DEST[63:32] ← 0; FI;
```

### Intel C/C++ Compiler Intrinsic Equivalents

```
PCMPGTB    __m64 _mm_cmpgt_pi8 ( __m64 m1, __m64 m2)
PCMPGTW    __m64 _mm_pcmpgt_pi16 ( __m64 m1, __m64 m2)
DCMPGTD    __m64 _mm_pcmpgt_pi32 ( __m64 m1, __m64 m2)
PCMPGTB    __m128i _mm_cmpgt_epi8 ( __m128i a, __m128i b)
PCMPGTW    __m128i _mm_cmpgt_epi16 ( __m128i a, __m128i b)
DCMPGTD    __m128i _mm_cmpgt_epi32 ( __m128i a, __m128i b)
```

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## INSTRUCTION SET REFERENCE, N-Z

### Real-Address Mode Exceptions

- #GP(0) (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  
If any part of the operand lies outside of the effective address space from 0 to FFFFH.
- #UD If CR0.EM[bit 2] = 1.  
128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
- #NM If CR0.TS[bit 3] = 1.
- #MF (64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

- #PF(fault-code) For a page fault.
- #AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.  
(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
- #UD If CR0.EM[bit 2] = 1.  
(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  
(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
- #NM If CR0.TS[bit 3] = 1.
- #MF (64-bit operations only) If there is a pending x87 FPU exception.

## INSTRUCTION SET REFERENCE, N-Z

#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**PEXTRW—Extract Word**

<b>Opcode</b>	<b>Instruction</b>	<b>64-Bit Mode</b>	<b>Compat/ Leg Mode</b>	<b>Description</b>
0F C5 /r ib	PEXTRW <i>r32</i> , <i>mm</i> , <i>imm8</i>	Valid	Valid	Extract the word specified by <i>imm8</i> from <i>mm</i> and move it to <i>r32</i> , bits 15-0. Zero-extend the result.
REX.W + 0F C5 /r ib	PEXTRW <i>r64</i> , <i>mm</i> , <i>imm8</i>	Valid	N.E.	Extract the word specified by <i>imm8</i> from <i>mm</i> and move it to <i>r64</i> , bits 15-0. Zero-extend the result.
66 0F C5 /r ib	PEXTRW <i>r32</i> , <i>xmm</i> , <i>imm8</i>	Valid	Valid	Extract the word specified by <i>imm8</i> from <i>xmm</i> and move it to <i>r32</i> , bits 15-0. Zero-extend the result.
REX.W + 66 0F C5 /r ib	PEXTRW <i>r64</i> , <i>xmm</i> , <i>imm8</i>	Valid	N.E.	Extract the word specified by <i>imm8</i> from <i>xmm</i> and move it to <i>r64</i> , bits 15-0. Zero-extend the result.

**Description**

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand is the low word of a general-purpose register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The high word of the destination operand is cleared (set to all 0s).

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64-bit general purpose registers.

**Operation**

```

IF (64-Bit Mode and REX.W used and 64-bit register selected)
THEN
    FOR (PEXTRW instruction with 64-bit source operand)
        { SEL ← COUNT AND 3H;
          TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
          r64[15:0] ← TEMP[15:0];
          r64[63:16] ← ZERO_FILL; };
    FOR (PEXTRW instruction with 128-bit source operand)
        { SEL ← COUNT AND 7H;
          TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
          r64[15:0] ← TEMP[15:0];
          r64[63:16] ← ZERO_FILL; }
ELSE
    FOR (PEXTRW instruction with 64-bit source operand)
        { SEL ← COUNT AND 3H;
          TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
          r32[15:0] ← TEMP[15:0];
          r32[31:16] ← ZERO_FILL; };
    FOR (PEXTRW instruction with 128-bit source operand)
        { SEL ← COUNT AND 7H;
          TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
          r32[15:0] ← TEMP[15:0];
          r32[31:16] ← ZERO_FILL; };
FI;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

PEXTRW    int_mm_extract_pi16 (__m64 a, int n)
PEXTRW    int_mm_extract_epi16 (__m128i a, int imm)

```

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Protected Mode Exceptions**

```

#GP(0)    If a memory operand effective address is outside the CS, DS, ES, FS, or
           GS segment limit.

#SS(0)    If a memory operand effective address is outside the SS segment
           limit.

```

## INSTRUCTION SET REFERENCE, N-Z

#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

## PINSRW—Insert Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F C4 /r ib	PINSRW <i>mm</i> , <i>r32/m16</i> , <i>imm8</i>	Valid	Valid	Insert the low word from <i>r32</i> or from <i>m16</i> into <i>mm</i> at the word position specified by <i>imm8</i>
REX.W + 0F C4 /r ib	PINSRW <i>mm</i> , <i>r64/m16</i> , <i>imm8</i>	Valid	N.E.	Insert the low word from <i>r64</i> or from <i>m16</i> into <i>mm</i> at the word position specified by <i>imm8</i>
66 0F C4 /r ib	PINSRW <i>xmm</i> , <i>r32/m16</i> , <i>imm8</i>	Valid	Valid	Move the low word of <i>r32</i> or from <i>m16</i> into <i>xmm</i> at the word position specified by <i>imm8</i> .
REX.W + 66 0F C4 /r ib	PINSRW <i>xmm</i> , <i>r64/m16</i> , <i>imm8</i>	Valid	N.E.	Move the low word of <i>r64</i> or from <i>m16</i> into <i>xmm</i> at the word position specified by <i>imm8</i> .

### Description

Copies a word from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

### Operation

PINSRW instruction with 64-bit source operand:

SEL ← COUNT AND 3H;

CASE (Determine word position) OF

SEL ← 0: MASK ← 000000000000FFFFH;

SEL ← 1: MASK ← 00000000FFFF0000H;

SEL ← 2: MASK ← 0000FFFF00000000H;

SEL ← 3: MASK ← FFFF000000000000H;

DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL \* 16)) AND MASK);

PINSRW instruction with 128-bit source operand:

SEL ← COUNT AND 7H;

## INSTRUCTION SET REFERENCE, N-Z

CASE (Determine word position) OF

SEL ← 0: MASK ← 00000000000000000000000000000000FFFFH;

SEL ← 1: MASK ← 00000000000000000000000000000000FFFF0000H;

SEL ← 2: MASK ← 00000000000000000000000000000000FFFF00000000H;

SEL ← 3: MASK ← 00000000000000000000000000000000FFFF000000000000H;

SEL ← 4: MASK ← 00000000000000000000000000000000FFFF0000000000000000H;

SEL ← 5: MASK ← 00000000FFFF0000000000000000000000000000H;

SEL ← 6: MASK ← 0000FFFF00000000000000000000000000000000H;

SEL ← 7: MASK ← FFFF00000000000000000000000000000000000000H;

DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL \* 16)) AND MASK);

### Intel C/C++ Compiler Intrinsic Equivalent

PINSRW `__m64 _mm_insert_pi16` (`__m64 a`, `int d`, `int n`)

PINSRW `__m128i _mm_insert_epi16` (`__m128i a`, `int b`, `int imm`)

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to FFFFH.



#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PMADDWD—Multiply and Add Packed Integers

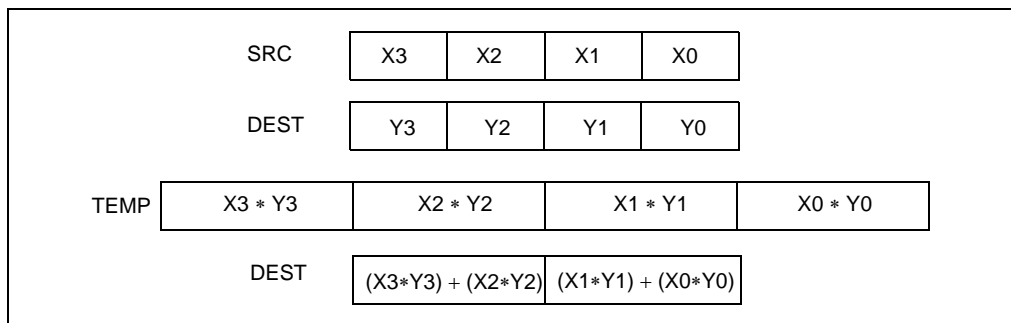
Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F F5 /r	PMADDWD <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Multiply the packed words in <i>mm</i> by the packed words in <i>mm/m64</i> , add adjacent doubleword results, and store in <i>mm</i> .
66 0F F5 /r	PMADDWD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Multiply the packed word integers in <i>xmm1</i> by the packed word integers in <i>xmm2/m128</i> , add adjacent doubleword results, and store in <i>xmm1</i> .

### Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-2 shows this operation when using 64-bit operands.) The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

The PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).



**Figure 4-2. PMADDWD Execution Model Using 64-bit Operands**

## Operation

PMADDWD instruction with 64-bit operands:

$$\begin{aligned} \text{DEST}[31:0] &\leftarrow (\text{DEST}[15:0] * \text{SRC}[15:0]) + (\text{DEST}[31:16] * \text{SRC}[31:16]); \\ \text{DEST}[63:32] &\leftarrow (\text{DEST}[47:32] * \text{SRC}[47:32]) + (\text{DEST}[63:48] * \text{SRC}[63:48]); \end{aligned}$$

PMADDWD instruction with 128-bit operands:

$$\begin{aligned} \text{DEST}[31:0] &\leftarrow (\text{DEST}[15:0] * \text{SRC}[15:0]) + (\text{DEST}[31:16] * \text{SRC}[31:16]); \\ \text{DEST}[63:32] &\leftarrow (\text{DEST}[47:32] * \text{SRC}[47:32]) + (\text{DEST}[63:48] * \text{SRC}[63:48]); \\ \text{DEST}[95:64] &\leftarrow (\text{DEST}[79:64] * \text{SRC}[79:64]) + (\text{DEST}[95:80] * \text{SRC}[95:80]); \\ \text{DEST}[127:96] &\leftarrow (\text{DEST}[111:96] * \text{SRC}[111:96]) + (\text{DEST}[127:112] * \text{SRC}[127:112]); \end{aligned}$$

## Intel C/C++ Compiler Intrinsic Equivalent

PMADDWD `__m64 __mm_madd_pi16(__m64 m1, __m64 m2)`

PMADDWD `__m128i __mm_madd_epi16 (__m128i a, __m128i b)`

## Flags Affected

None.

## Numeric Exceptions

None.

## Protected Mode Exceptions

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
- #SS(0) If a memory operand effective address is outside the SS segment limit.

## INSTRUCTION SET REFERENCE, N-Z

#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PMAXSW—Maximum of Packed Signed Word Integers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F EE /r	PMAXSW <i>mm1</i> , <i>mm2/m64</i>	Valid	Valid	Compare signed word integers in <i>mm2/m64</i> and <i>mm1</i> and return maximum values.
66 0F EE /r	PMAXSW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Compare signed word integers in <i>xmm2/m128</i> and <i>xmm1</i> and return maximum values.

### Description

Performs an SIMD compare of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of word integers to the destination operand. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PMAXSW instruction for 64-bit operands:

```
IF DEST[15:0] > SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
IF DEST[63:48] > SRC[63:48] THEN
    DEST[63:48] ← DEST[63:48];
ELSE
    DEST[63:48] ← SRC[63:48]; FI;
```

PMAXSW instruction for 128-bit operands:

```
IF DEST[15:0] > SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:112] > SRC[127:112] THEN
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; FI;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

PMAXSW      `__m64 __mm_max_pi16(__m64 a, __m64 b)`

PMAXSW      `__m128i __mm_max_epi16 (__m128i a, __m128i b)`

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Protected Mode Exceptions**

#GP(0)            If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)            If a memory operand effective address is outside the SS segment limit.

#UD                If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM                If CR0.TS[bit 3] = 1.

#MF                (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code)    If a page fault occurs.

#AC(0)            (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP(0)            (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD                If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM                If CR0.TS[bit 3] = 1.

#MF                (64-bit operations only) If there is a pending x87 FPU exception.

## INSTRUCTION SET REFERENCE, N-Z

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#UD If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## PMAXUB—Maximum of Packed Unsigned Byte Integers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F DE /r	PMAXUB <i>mm1</i> , <i>mm2/m64</i>	Valid	Valid	Compare unsigned byte integers in <i>mm2/m64</i> and <i>mm1</i> and returns maximum values.
66 0F DE /r	PMAXUB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Compare unsigned byte integers in <i>xmm2/m128</i> and <i>xmm1</i> and returns maximum values.

### Description

Performs an SIMD compare of the packed unsigned byte integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of byte integers to the destination operand. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PMAXUB instruction for 64-bit operands:

```
IF DEST[7:0] > SRC[17:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
IF DEST[63:56] > SRC[63:56] THEN
    DEST[63:56] ← DEST[63:56];
ELSE
    DEST[63:56] ← SRC[63:56]; FI;
```

PMAXUB instruction for 128-bit operands:

```
IF DEST[7:0] > SRC[17:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] > SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
```

## INSTRUCTION SET REFERENCE, N-Z

### Intel C/C++ Compiler Intrinsic Equivalent

PMAXUB     \_\_m64 \_mm\_max\_pu8(\_\_m64 a, \_\_m64 b)

PMAXUB     \_\_m128i \_mm\_max\_epu8 ( \_\_m128i a, \_\_m128i b)

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

- #PF(fault-code) For a page fault.
- #AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.  
(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
- #UD If CR0.EM[bit 2] = 1.  
(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  
(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
- #NM If CR0.TS[bit 3] = 1.
- #MF (64-bit operations only) If there is a pending x87 FPU exception.
- #PF(fault-code) If a page fault occurs.
- #AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PMINSW—Minimum of Packed Signed Word Integers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F EA /r	PMINSW <i>mm1</i> , <i>mm2/m64</i>	Valid	Valid	Compare signed word integers in <i>mm2/m64</i> and <i>mm1</i> and return minimum values.
66 0F EA /r	PMINSW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Compare signed word integers in <i>xmm2/m128</i> and <i>xmm1</i> and return minimum values.

### Description

Performs an SIMD compare of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of word integers to the destination operand. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PMINSW instruction for 64-bit operands:

```
IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
IF DEST[63:48] < SRC[63:48] THEN
    DEST[63:48] ← DEST[63:48];
ELSE
    DEST[63:48] ← SRC[63:48]; FI;
```

PMINSW instruction for 128-bit operands:

```
IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:112] < SRC/m64[127:112] THEN
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; FI;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

PMINSW      \_\_m64 \_\_mm\_min\_pi16 ( \_\_m64 a, \_\_m64 b)

PMINSW      \_\_m128i \_\_mm\_min\_epi16 ( \_\_m128i a, \_\_m128i b)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Protected Mode Exceptions**

#GP(0)           If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)           If a memory operand effective address is outside the SS segment limit.

#UD              If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM              If CR0.TS[bit 3] = 1.

#MF              (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code)   If a page fault occurs.

#AC(0)           (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP(0)           (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD              If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM              If CR0.TS[bit 3] = 1.

#MF              (64-bit operations only) If there is a pending x87 FPU exception.

## INSTRUCTION SET REFERENCE, N-Z

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#UD If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PMINUB—Minimum of Packed Unsigned Byte Integers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F DA /r	PMINUB <i>mm1</i> , <i>mm2/m64</i>	Valid	Valid	Compare unsigned byte integers in <i>mm2/m64</i> and <i>mm1</i> and returns minimum values.
66 0F DA /r	PMINUB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Compare unsigned byte integers in <i>xmm2/m128</i> and <i>xmm1</i> and returns minimum values.

### Description

Performs an SIMD compare of the packed unsigned byte integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of byte integers to the destination operand. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PMINUB instruction for 64-bit operands:

```
IF DEST[7:0] < SRC[17:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
IF DEST[63:56] < SRC[63:56] THEN
    DEST[63:56] ← DEST[63:56];
ELSE
    DEST[63:56] ← SRC[63:56]; FI;
```

PMINUB instruction for 128-bit operands:

```
IF DEST[7:0] < SRC[17:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
```

## INSTRUCTION SET REFERENCE, N-Z

### Intel C/C++ Compiler Intrinsic Equivalent

PMINUB        \_\_m64 \_m\_min\_pu8 (\_\_m64 a, \_\_m64 b)

PMINUB        \_\_m128i \_mm\_min\_epu8 (\_\_m128i a, \_\_m128i b)

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.



**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#UD If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PMOVMSKB—Move Byte Mask

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F D7 /r	PMOVMSKB <i>r32, mm</i>	Valid	Valid	Move a byte mask of <i>mm</i> to <i>r32</i> .
REX.W + 0F D7 /r	PMOVMSKB <i>r64, mm</i>	Valid	N.E.	Move a byte mask of <i>mm</i> to the lower 32-bits of <i>r64</i> and zero-fill the upper 32-bits.
66 0F D7 /r	PMOVMSKB <i>r32, xmm</i>	Valid	Valid	Move a byte mask of <i>xmm</i> to <i>r32</i> .
REX.W + 66 0F D7 /r	PMOVMSKB <i>r64, xmm</i>	Valid	N.E.	Move a byte mask of <i>xmm</i> to the lower 32-bits of <i>r64</i> and zero-fill the upper 32-bits.

### Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand). The source operand is an MMX technology register or an XMM register; the destination operand is a general-purpose register. When operating on 64-bit operands, the byte mask is 8 bits; when operating on 128-bit operands, the byte mask is 16-bits.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

### Operation

PMOVMSKB instruction with 64-bit source operand and r32:

```
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r32[7] ← SRC[63];
r32[31:8] ← ZERO_FILL;
```

PMOVMSKB instruction with 128-bit source operand and r32:

```
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r32[15] ← SRC[127];
r32[31:16] ← ZERO_FILL;
```

PMOVMSKB instruction with 64-bit source operand and r64:

```
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r64[7] ← SRC[63];
```

r64[63:8] ← ZERO\_FILL;

PMOVMSKB instruction with 128-bit source operand and r64:

r64[0] ← SRC[7];

r64[1] ← SRC[15];

(\* Repeat operation for bytes 2 through 14 \*)

r64[15] ← SRC[127];

r64[63:16] ← ZERO\_FILL;

### Intel C/C++ Compiler Intrinsic Equivalent

PMOVMSKB int\_mm\_movemask\_pi8(\_\_m64 a)

PMOVMSKB int\_mm\_movemask\_epi8 ( \_\_m128i a)

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#UD If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

### Real-Address Mode Exceptions

Same exceptions as in Protected Mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in Protected Mode.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

Same exceptions as in Protected Mode.

## PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F E4 /r	PMULHUW <i>mm1</i> , <i>mm2/m64</i>	Valid	Valid	Multiply the packed unsigned word integers in <i>mm1</i> register and <i>mm2/m64</i> , and store the high 16 bits of the results in <i>mm1</i> .
66 0F E4 /r	PMULHUW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Multiply the packed unsigned word integers in <i>xmm1</i> and <i>xmm2/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .

### Description

Performs an SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-3 shows this operation when using 64-bit operands.) The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

n 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

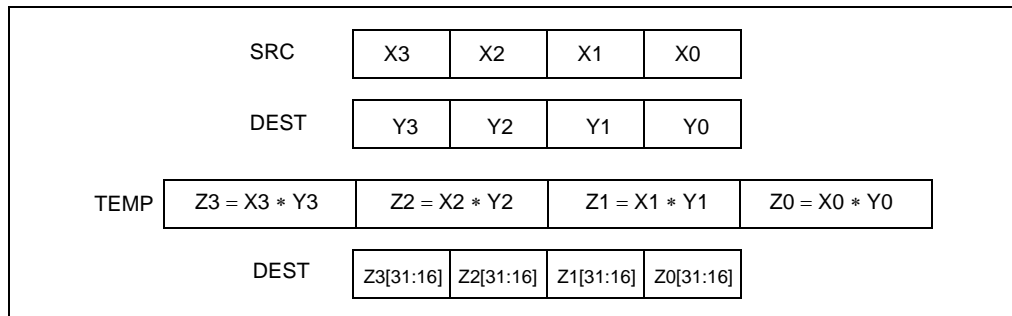


Figure 4-3. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

### Operation

PMULHUW instruction with 64-bit operands:

- TEMP0[31:0] ← DEST[15:0] \* SRC[15:0]; (\* Unsigned multiplication \*)
- TEMP1[31:0] ← DEST[31:16] \* SRC[31:16];
- TEMP2[31:0] ← DEST[47:32] \* SRC[47:32];
- TEMP3[31:0] ← DEST[63:48] \* SRC[63:48];

DEST[15:0] ← TEMP0[31:16];  
 DEST[31:16] ← TEMP1[31:16];  
 DEST[47:32] ← TEMP2[31:16];  
 DEST[63:48] ← TEMP3[31:16];

PMULHUW instruction with 128-bit operands:

TEMP0[31:0] ← DEST[15:0] \* SRC[15:0]; (\* Unsigned multiplication \*)  
 TEMP1[31:0] ← DEST[31:16] \* SRC[31:16];  
 TEMP2[31:0] ← DEST[47:32] \* SRC[47:32];  
 TEMP3[31:0] ← DEST[63:48] \* SRC[63:48];  
 TEMP4[31:0] ← DEST[79:64] \* SRC[79:64];  
 TEMP5[31:0] ← DEST[95:80] \* SRC[95:80];  
 TEMP6[31:0] ← DEST[111:96] \* SRC[111:96];  
 TEMP7[31:0] ← DEST[127:112] \* SRC[127:112];  
 DEST[15:0] ← TEMP0[31:16];  
 DEST[31:16] ← TEMP1[31:16];  
 DEST[47:32] ← TEMP2[31:16];  
 DEST[63:48] ← TEMP3[31:16];  
 DEST[79:64] ← TEMP4[31:16];  
 DEST[95:80] ← TEMP5[31:16];  
 DEST[111:96] ← TEMP6[31:16];  
 DEST[127:112] ← TEMP7[31:16];

### Intel C/C++ Compiler Intrinsic Equivalent

PMULHUW `__m64 __mm_mulhi_pu16(__m64 a, __m64 b)`  
 PMULHUW `__m128i __mm_mulhi_epu16 (__m128i a, __m128i b)`

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
 (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If CR0.EM[bit 2] = 1.  
 (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  
 (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

## INSTRUCTION SET REFERENCE, N-Z

#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F E5 /r	PMULHW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Multiply the packed signed word integers in <i>mm1</i> register and <i>mm2/m64</i> , and store the high 16 bits of the results in <i>mm1</i> .
66 0F E5 /r	PMULHW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Multiply the packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .

### Description

Performs an SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-3 shows this operation when using 64-bit operands.) The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

n 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PMULHW instruction with 64-bit operands:

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
```

PMULHW instruction with 128-bit operands:

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
```



```

DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
DEST[79:64] ← TEMP4[31:16];
DEST[95:80] ← TEMP5[31:16];
DEST[111:96] ← TEMP6[31:16];
DEST[127:112] ← TEMP7[31:16];

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

PMULHW    __m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)
PMULHW    __m128i _mm_mulhi_epi16 (__m128i a, __m128i b)

```

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## INSTRUCTION SET REFERENCE, N-Z

### Real-Address Mode Exceptions

- #GP(0) (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  
If any part of the operand lies outside of the effective address space from 0 to FFFFH.
- #UD If CR0.EM[bit 2] = 1.  
128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
- #NM If CR0.TS[bit 3] = 1.
- #MF (64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

- #PF(fault-code) For a page fault.
- #AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.  
(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
- #UD If CR0.EM[bit 2] = 1.  
(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  
(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
- #NM If CR0.TS[bit 3] = 1.
- #MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F D5 /r	PMULLW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Multiply the packed signed word integers in <i>mm1</i> register and <i>mm2/m64</i> , and store the low 16 bits of the results in <i>mm1</i> .
66 0F D5 /r	PMULLW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Multiply the packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> , and store the low 16 bits of the results in <i>xmm1</i> .

### Description

Performs an SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-3 shows this operation when using 64-bit operands.) The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

n 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

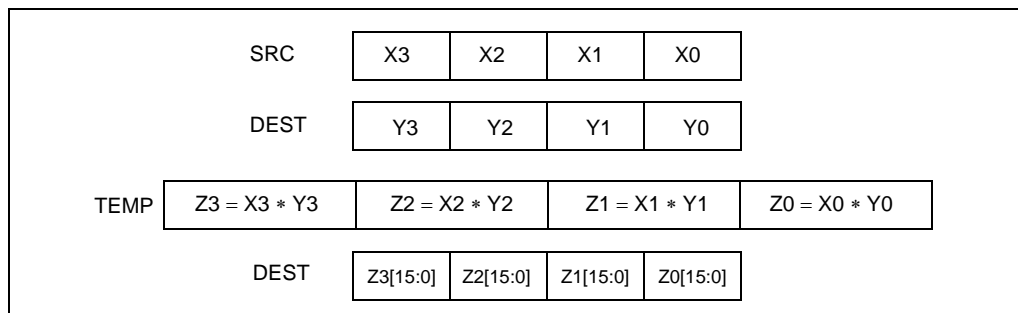


Figure 4-4. PMULLU Instruction Operation Using 64-bit Operands

### Operation

PMULLW instruction with 64-bit operands:

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[15:0];

```

DEST[31:16] ← TEMP1[15:0];  
 DEST[47:32] ← TEMP2[15:0];  
 DEST[63:48] ← TEMP3[15:0];

PMULLW instruction with 64-bit operands:

TEMP0[31:0] ← DEST[15:0] \* SRC[15:0]; (\* Signed multiplication \*)  
 TEMP1[31:0] ← DEST[31:16] \* SRC[31:16];  
 TEMP2[31:0] ← DEST[47:32] \* SRC[47:32];  
 TEMP3[31:0] ← DEST[63:48] \* SRC[63:48];  
 TEMP4[31:0] ← DEST[79:64] \* SRC[79:64];  
 TEMP5[31:0] ← DEST[95:80] \* SRC[95:80];  
 TEMP6[31:0] ← DEST[111:96] \* SRC[111:96];  
 TEMP7[31:0] ← DEST[127:112] \* SRC[127:112];  
 DEST[15:0] ← TEMP0[15:0];  
 DEST[31:16] ← TEMP1[15:0];  
 DEST[47:32] ← TEMP2[15:0];  
 DEST[63:48] ← TEMP3[15:0];  
 DEST[79:64] ← TEMP4[15:0];  
 DEST[95:80] ← TEMP5[15:0];  
 DEST[111:96] ← TEMP6[15:0];  
 DEST[127:112] ← TEMP7[15:0];

### Intel C/C++ Compiler Intrinsic Equivalent

PMULLW `__m64 __mm_mullo_pi16(__m64 m1, __m64 m2)`

PMULLW `__m128i __mm_mullo_epi16 (__m128i a, __m128i b)`

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If CR0.EM[bit 2] = 1.

128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one

## INSTRUCTION SET REFERENCE, N-Z

that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.

- #NM If CR0.TS[bit 3] = 1.
- #MF (64-bit operations only) If there is a pending x87 FPU exception.
- #PF(fault-code) If a page fault occurs.
- #AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

- #GP(0) (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  
  
If any part of the operand lies outside of the effective address space from 0 to FFFFH.
- #UD If CR0.EM[bit 2] = 1.  
  
128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
- #NM If CR0.TS[bit 3] = 1.
- #MF (64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

- #PF(fault-code) For a page fault.
- #AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F F4 /r	PMULUDQ <i>mm1</i> , <i>mm2/m64</i>	Valid	Valid	Multiply unsigned doubleword integer in <i>mm1</i> by unsigned doubleword integer in <i>mm2/m64</i> , and store the quadword result in <i>mm1</i> .
66 0F F4 /r	PMULUDQ <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Multiply packed unsigned doubleword integers in <i>xmm1</i> by packed unsigned doubleword integers in <i>xmm2/m128</i> , and store the quadword results in <i>xmm1</i> .

### Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand. The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location, or it can be two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or an 128-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register or two packed doubleword integers stored in the first and third doublewords of an XMM register. The result is an unsigned quadword integer stored in the destination an MMX technology register or two packed unsigned quadword integers stored in an XMM register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation; for 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.

n 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PMULUDQ instruction with 64-Bit operands:

$$\text{DEST}[63:0] \leftarrow \text{DEST}[31:0] * \text{SRC}[31:0];$$

PMULUDQ instruction with 128-Bit operands:

$$\text{DEST}[63:0] \leftarrow \text{DEST}[31:0] * \text{SRC}[31:0];$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[95:64] * \text{SRC}[95:64];$$

### Intel C/C++ Compiler Intrinsic Equivalent

PMULUDQ     \_\_m64 \_mm\_mul\_su32 (\_\_m64 a, \_\_m64 b)

PMULUDQ     \_\_m128i \_mm\_mul\_epu32 (\_\_m128i a, \_\_m128i b)



**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

## INSTRUCTION SET REFERENCE, N-Z

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.  (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is

## POP—Pop a Value from the Stack

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
8F /0	POP <i>r/m16</i>	Valid	Valid	Pop top of stack into <i>m16</i> ; increment stack pointer.
8F /0	POP <i>r/m32</i>	N.E.	Valid	Pop top of stack into <i>m32</i> ; increment stack pointer.
8F /0	POP <i>r/m64</i>	Valid	N.E.	Pop top of stack into <i>m64</i> ; increment stack pointer. Cannot encode 32-bit operand size.
58+ <i>rw</i>	POP <i>r16</i>	Valid	Valid	Pop top of stack into <i>r16</i> ; increment stack pointer.
58+ <i>rd</i>	POP <i>r32</i>	N.E.	Valid	Pop top of stack into <i>r32</i> ; increment stack pointer.
58+ <i>rd</i>	POP <i>r64</i>	Valid	N.E.	Pop top of stack into <i>r64</i> ; increment stack pointer. Cannot encode 32-bit operand size.
1F	POP DS	Invalid	Valid	Pop top of stack into DS; increment stack pointer.
07	POP ES	Invalid	Valid	Pop top of stack into ES; increment stack pointer.
17	POP SS	Invalid	Valid	Pop top of stack into SS; increment stack pointer.
0F A1	POP FS	Valid	Valid	Pop top of stack into FS; increment stack pointer by 16 bits.
0F A1	POP FS	N.E.	Valid	Pop top of stack into FS; increment stack pointer by 32 bits.
0F A1	POP FS	Valid	N.E.	Pop top of stack into FS; increment stack pointer by 64 bits.
0F A9	POP GS	Valid	Valid	Pop top of stack into GS; increment stack pointer by 16 bits.
0F A9	POP GS	N.E.	Valid	Pop top of stack into GS; increment stack pointer by 32 bits.
0F A9	POP GS	Valid	N.E.	Pop top of stack into GS; increment stack pointer by 64 bits.

### Description

Loads the value from the top of the stack to the location specified with the destination operand (or explicit opcode) and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

The address-size attribute of the stack segment determines the stack pointer size (16, 32, 64 bits) and the operand-size attribute of the current code segment determines the amount the stack pointer is incremented (2, 4, 8 bytes).

For example, if the address- and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is incremented by 4; if they are 16, the 16-bit SP register is incremented by 2. (The B flag in the stack segment's segment descriptor determines the stack's address-size attribute,

## INSTRUCTION SET REFERENCE, N-Z

and the D flag in the current code segment's segment descriptor, along with prefixes, determines the operand-size attribute and also the address-size attribute of the destination operand.)

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automatically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the "Operation" section below).

A NULL value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is NULL.

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register. For the case of a 16-bit stack where ESP wraps to 0H as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

A POP SS instruction inhibits all interrupts, including the NMI interrupt, until after execution of the next instruction. This action allows sequential execution of POP SS and MOV ESP, EBP instructions without the danger of having an invalid stack during an interrupt<sup>1</sup>. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

In 64-bit mode, using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). When in 64-bit mode, POPs using 32-bit operands are not encodable and POPs to DS, ES, SS are not valid. See the summary chart at the beginning of this section for encoding data and limits.

---

1. If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a POP SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that POP the SS register, only the first instruction in the sequence is guaranteed to delay an interrupt.

In the following sequence, interrupts may be recognized before POP ESP executes:

```
POP SS
POP SS
POP ESP
```

**Operation**

```

IF StackAddrSize = 32
  THEN
    IF OperandSize = 32
      THEN
        DEST ← SS:ESP; (* Copy a doubleword *)
        ESP ← ESP + 4;
      ELSE (* OperandSize = 16*)
        DEST ← SS:ESP; (* Copy a word *)
        ESP ← ESP + 2;
    FI;
  ELSE IF StackAddrSize = 64
    THEN
      IF OperandSize = 64
        THEN
          DEST ← SS:RSP; (* Copy quadword *)
          RSP ← RSP + 8;
        ELSE (* OperandSize = 16*)
          DEST ← SS:RSP; (* Copy a word *)
          RSP ← RSP + 2;
        FI;
      FI;
    ELSE StackAddrSize = 16
      THEN
        IF OperandSize = 16
          THEN
            DEST ← SS:SP; (* Copy a word *)
            SP ← SP + 2;
          ELSE (* OperandSize = 32 *)
            DEST ← SS:SP; (* Copy a doubleword *)
            SP ← SP + 4;
          FI;
        FI;
      FI;
    FI;
  FI;

```

Loading a segment register while in protected mode results in special actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

**64-BIT\_MODE**

```

IF FS, or GS is loaded with non-NULL selector;
  THEN
    IF segment selector index is outside descriptor table limits
      OR segment is not a data or readable code segment
      OR ((segment is a data or nonconforming code segment)
        AND (both RPL and CPL > DPL))
      THEN #GP(selector);

```

## INSTRUCTION SET REFERENCE, N-Z

```
        IF segment not marked present
            THEN #NP(selector);
    ELSE
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
    FI;
FI;
IF FS, or GS is loaded with a NULL selector;
    THEN
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
FI;

PREOTECTED MODE OR COMPATIBILITY MODE;
IF SS is loaded;
    THEN
        IF segment selector is NULL
            THEN #GP(0);
        FI;
        IF segment selector index is outside descriptor table limits
            or segment selector's RPL ≠ CPL
            or segment is not a writable data segment
            or DPL ≠ CPL
            THEN #GP(selector);
        FI;
        IF segment not marked present
            THEN #SS(selector);
        ELSE
            SS ← segment selector;
            SS ← segment descriptor;
        FI;
FI;

IF DS, ES, FS, or GS is loaded with non-NULL selector;
    THEN
        IF segment selector index is outside descriptor table limits
            or segment is not a data or readable code segment
            or ((segment is a data or nonconforming code segment)
                and (both RPL and CPL > DPL))
            THEN #GP(selector);
        FI;
        IF segment not marked present
            THEN #NP(selector);
        ELSE
            SegmentRegister ← segment selector;
            SegmentRegister ← segment descriptor;
```

```

FI;
FI;
IF DS, ES, FS, or GS is loaded with a NULL selector
  THEN
    SegmentRegister ← segment selector;
    SegmentRegister ← segment descriptor;
FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	<p>If attempt is made to load SS register with NULL segment selector.</p> <p>If the destination operand is in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#GP(selector)	<p>If segment selector index is outside descriptor table limits.</p> <p>If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.</p> <p>If the SS register is being loaded and the segment pointed to is a non-writable data segment.</p> <p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.</p> <p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.</p>
#SS(0)	<p>If the current top of stack is not within the stack segment.</p> <p>If a memory operand effective address is outside the SS segment limit.</p>
#SS(selector)	<p>If the SS register is being loaded and the segment pointed to is marked not present.</p>
#NP	<p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.</p>
#PF(fault-code)	<p>If a page fault occurs.</p>
#AC(0)	<p>If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.</p>

## INSTRUCTION SET REFERENCE, N-Z

### Real-Address Mode Exceptions

#GP                      If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

### Virtual-8086 Mode Exceptions

#GP(0)                  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#PF(fault-code)        If a page fault occurs.

#AC(0)                  If an unaligned memory reference is made while alignment checking is enabled.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#GP(0)                  If the memory address is in a non-canonical form.

#SS(U)                  If the stack address is in a non-canonical form.

#GP(selector)          If the descriptor is outside the descriptor table limit.

If the FS or GS register is being loaded and the segment pointed to is not a data or readable code segment.

If the FS or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.

#AC(0)                  If an unaligned memory reference is made while alignment checking is enabled.

#PF(fault-code)        If a page fault occurs.

#NP                      If the FS or GS register is being loaded and the segment pointed to is marked not present.



## POPA/POPAD—Pop All General-Purpose Registers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
61	POPA	Invalid	Valid	Pop DI, SI, BP, BX, DX, CX, and AX.
61	POPAD	Invalid	Valid	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX.

### Description

Pops doublewords (POPAD) or words (POPA) from the stack into the general-purpose registers. The registers are loaded in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (if the operand-size attribute is 32) and DI, SI, BP, BX, DX, CX, and AX (if the operand-size attribute is 16). (These instructions reverse the operation of the PUSH/POPA instructions.) The value on the stack for the ESP or SP register is ignored. Instead, the ESP or SP register is incremented after each register is loaded.

The POPA (pop all) and POPAD (pop all double) mnemonics reference the same opcode. The POPA instruction is intended for use when the operand-size attribute is 16 and the POPAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPA is used and to 32 when POPAD is used (using the operand-size override prefix [66H] if necessary). Others may treat these mnemonics as synonyms (POPA/POPAD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used. (The D flag in the current code segment's segment descriptor determines the operand-size attribute.)

This instruction executes as described in non-64-bit modes. It is not valid in 64-bit mode.

### Operation

```

IF 64-Bit Mode
  THEN
    #UD;
ELSE
  IF OperandSize = 32 (* Instruction = POPAD *)
    THEN
      EDI ← Pop();
      ESI ← Pop();
      EBP ← Pop();
      Increment ESP by 4; (* Skip next 4 bytes of stack *)
      EBX ← Pop();
      EDX ← Pop();
      ECX ← Pop();
      EAX ← Pop();
  ELSE (* OperandSize = 16, instruction = POPA *)
    DI ← Pop();
    SI ← Pop();
  
```

## INSTRUCTION SET REFERENCE, N-Z

```
    BP ← Pop();
    Increment ESP by 2; (* Skip next 2 bytes of stack *)
    BX ← Pop();
    DX ← Pop();
    CX ← Pop();
    AX ← Pop();
FI;
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

- #SS(0) If the starting or ending stack address is not within the stack segment.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

### Real-Address Mode Exceptions

- #SS If the starting or ending stack address is not within the stack segment.

### Virtual-8086 Mode Exceptions

- #SS(0) If the starting or ending stack address is not within the stack segment.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If an unaligned memory reference is made while alignment checking is enabled.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

- #UD If in 64-bit mode.

## POPF/POPFD/POPFDQ—Pop Stack into EFLAGS Register

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9D	POPF	Valid	Valid	Pop top of stack into lower 16 bits of EFLAGS.
9D	POPFD	N.E.	Valid	Pop top of stack into EFLAGS.
REX.W + 9D	POPFDQ	Valid	N.E.	Pop top of stack and zero-extend into RFLAGS.

### Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register, or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). These instructions reverse the operation of the PUSHF/PUSHFD instructions.

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16; the POPFD instruction is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size to 16 for POPF and to 32 for POPFD. Others may treat the mnemonics as synonyms (POPF/POPFD) and use the setting of the operand-size attribute to determine the size of values to pop from the stack.

The effect of POPF/POPFD on the EFLAGS register changes, depending on the mode of operation. When the processor is operating in protected mode at privilege level 0 (or in real-address mode, the equivalent to privilege level 0), all non-reserved flags in the EFLAGS register except VIP, VIF, and VM may be modified. VIP, VIF and VM remain unaffected.

When operating in protected mode with a privilege level greater than 0, but less than or equal to IOPL, all flags can be modified except the IOPL field and VIP, VIF, and VM. Here, the IOPL flags are unaffected, the VIP and VIF flags are cleared, and the VM flag is unaffected. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur but privileged bits do not change.

When operating in virtual-8086 mode, the IOPL must be equal to 3 to use POPF/POPFD instructions; VM, RF, IOPL, VIP, and VIF are unaffected. If the IOPL is less than 3, POPF/POPFD causes a general-protection exception (#GP).

In 64-bit mode, use REX.W to pop the top of stack to RFLAGS. The mnemonic assigned is POPFDQ (note that the 32-bit operand is not encodable). POPFDQ pops 64 bits from the stack, loads the lower 32 bits into RFLAGS, and zero extends the upper bits of RFLAGS.

See Chapter 3 of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for more information about the EFLAGS registers.

## INSTRUCTION SET REFERENCE, N-Z

### Operation

```
IF VM = 0 (* Not in Virtual-8086 Mode *)
  THEN IF CPL = 0
    THEN
      IF OperandSize = 32;
        THEN
          EFLAGS ← Pop(); (* 32-bit pop *)
          (* All non-reserved flags except VIP, VIF, and VM can be modified;
             VIP and VIF are cleared; VM and all reserved bits are unaffected *)
        ELSE IF (OperandSize = 64)
          RFLAGS = Pop(); (* 64-bit pop *)
          (* All non-reserved flags except VIP, VIF, and VM can be modified; VIP
             and VIF are cleared; VM and all reserved bits are unaffected *)
        ELSE (* OperandSize = 16 *)
          EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
          (* All non-reserved flags can be modified *)
      FI;
    ELSE (* CPL > 0 *)
      IF OperandSize = 32
        THEN
          IF CPL > IOPL
            THEN
              EFLAGS ← Pop(); (* 32-bit pop *)
              (* All non-reserved bits except IF, IOPL, VIP, and VIF can be
                 modified; IF, IOPL, and VM, and all reserved bits are unaffected;
                 VIP and VIF are cleared *)
            ELSE
              EFLAGS ← Pop(); (* 32-bit pop *)
              (* All non-reserved bits except IOPL, VIP, and VIF can be
                 modified; IOPL, VM, and all reserved bits are unaffected;
                 VIP and VIF are cleared *)
          FI;
        ELSE IF (OperandSize = 64)
          IF CPL > IOPL
            THEN
              RFLAGS ← Pop(); (* 64-bit pop *)
              (* All non-reserved bits except IF, IOPL, VIP, and VIF can
                 be modified; IF, IOPL, VM, and all reserved bits are unaffected;
                 VIP and VIF are cleared *)
            ELSE
              RFLAGS ← Pop(); (* 64-bit pop *)
              (* All non-reserved bits except IOPL, VIP, and VIF can be
                 modified; IOPL, VM, and all reserved bits are unaffected;
                 VIP and VIF are cleared *)
          FI;
        ELSE (* OperandSize = 16 *)
          EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
          (* All non-reserved bits except IOPL can be modified; IOPL and all
             reserved bits are unaffected *)
      FI;
    ELSE (* In Virtual-8086 Mode *)
      IF IOPL = 3
        THEN IF OperandSize = 32
```

```

THEN
    EFLAGS ← Pop();
    (* All non-reserved bits except VM, RF, IOPL, VIP, and VIF can be
    modified; VM, RF, IOPL, VIP, VIF, and all reserved bits are unaffected *)
ELSE
    EFLAGS[15:0] ← Pop(); FI;
    (* All non-reserved bits except IOPL can be modified;
    IOPL and all reserved bits are unaffected *)
ELSE (* IOPL < 3 *)
    #GP(0); (* Trap to virtual-8086 monitor *)
FI;
FI;
FI;

```

### Flags Affected

All flags except the reserved bits and the VM bit.

### Protected Mode Exceptions

#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

### Real-Address Mode Exceptions

#SS	If the top of stack is not within the stack segment.
-----	--

### Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.  If an attempt is made to execute the POPF/POPFQ instruction with an operand-size override prefix.
#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(U)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is

## POR—Bitwise Logical OR

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F EB /r	POR <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Bitwise OR of <i>mm/m64</i> and <i>mm</i> .
66 0F EB /r	POR <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Bitwise OR of <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST ← DEST OR SRC;

### Intel C/C++ Compiler Intrinsic Equivalent

POR            \_\_m64 \_\_mm\_or\_si64(\_\_m64 m1, \_\_m64 m2)

POR            \_\_m128i \_\_mm\_or\_si128(\_\_m128i m1, \_\_m128i m2)

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0)            If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)            If a memory operand effective address is outside the SS segment limit.

## INSTRUCTION SET REFERENCE, N-Z

#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.



**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PREFETCH $h$ —Prefetch Data Into Caches

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 18 /1	PREFETCHT0 $m8$	Valid	Valid	Move data from $m8$ closer to the processor using T0 hint.
0F 18 /2	PREFETCHT1 $m8$	Valid	Valid	Move data from $m8$ closer to the processor using T1 hint.
0F 18 /3	PREFETCHT2 $m8$	Valid	Valid	Move data from $m8$ closer to the processor using T2 hint.
0F 18 /0	PREFETCHNTA $m8$	Valid	Valid	Move data from $m8$ closer to the processor using NTA hint.

### Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
  - Pentium III processor—1st- or 2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T1 (temporal data with respect to first level cache)—prefetch data into level 2 cache and higher.
  - Pentium III processor—2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T2 (temporal data with respect to second level cache)—prefetch data into level 2 cache and higher.
  - Pentium III processor—2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.
  - Pentium III processor—1st-level cache
  - Pentium 4 and Intel Xeon processors—2nd-level cache

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte. Use of any ModR/M value other than the specified ones will lead to unpredictable behavior.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The `PREFETCHh` instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A `PREFETCHh` instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a `PREFETCHh` instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A `PREFETCHh` instruction is also unordered with respect to CLFLUSH instructions, other `PREFETCHh` instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

FETCH (m8);

### Intel C/C++ Compiler Intrinsic Equivalent

```
void_mm_prefetch(char *p, int i)
```

The argument “\*p” gives the address of the byte (and corresponding cache line) to be prefetched. The value “i” gives a constant (`_MM_HINT_T0`, `_MM_HINT_T1`, `_MM_HINT_T2`, or `_MM_HINT_NTA`) that specifies the type of prefetch operation to be performed.

### Numeric Exceptions

None.

### Exceptions (All Operating Modes)

None.

## PSADBW—Compute Sum of Absolute Differences

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F F6 /r	PSADBW <i>mm1</i> , <i>mm2/m64</i>	Valid	Valid	Computes the absolute differences of the packed unsigned byte integers from <i>mm2/m64</i> and <i>mm1</i> ; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r	PSADBW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Computes the absolute differences of the packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> ; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.

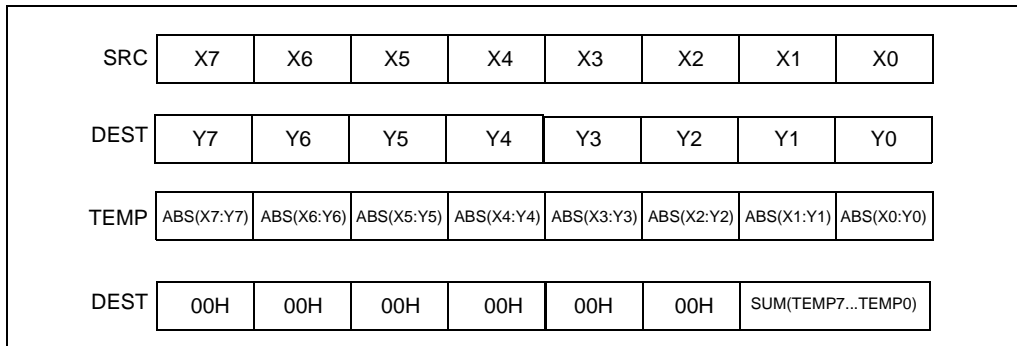
### Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Figure 4-5 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).



**Figure 4-5. PSADBW Instruction Operation Using 64-bit Operands**

## Operation

PSADBW instructions when using 64-bit operands:

```
TEMP0 ← ABS(DEST[7:0] – SRC[7:0]);
(* Repeat operation for bytes 2 through 6 *)
TEMP7 ← ABS(DEST[63:56] – SRC[63:56]);
DEST[15:0] ← SUM(TEMP0:TEMP7);
DEST[63:16] ← 000000000000H;
```

PSADBW instructions when using 128-bit operands:

```
TEMP0 ← ABS(DEST[7:0] – SRC[7:0]);
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(DEST[127:120] – SRC[127:120]);
DEST[15:0] ← SUM(TEMP0:TEMP7);
DEST[63:6] ← 000000000000H;
DEST[79:64] ← SUM(TEMP8:TEMP15);
DEST[127:80] ← 000000000000H;
```

## Intel C/C++ Compiler Intrinsic Equivalent

```
PSADBW    __m64_mm_sad_pu8(__m64 a, __m64 b)
```

```
PSADBW    __m128i_mm_sad_epu8(__m128i a, __m128i b)
```

## Flags Affected

None.

## Numeric Exceptions

None.

## INSTRUCTION SET REFERENCE, N-Z

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PSHUFD—Shuffle Packed Doublewords

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 70 /r ib	PSHUFD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	Valid	Valid	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .

### Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-6 shows the operation of the PSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location in the destination operand. For example, bits 0 and 1 of the order operand select the contents of doubleword 0 of the destination operand. The encoding of bits 0 and 1 of the order operand (see the field encoding in Figure 4-6) determines which doubleword from the source operand will be copied to doubleword 0 of the destination operand.

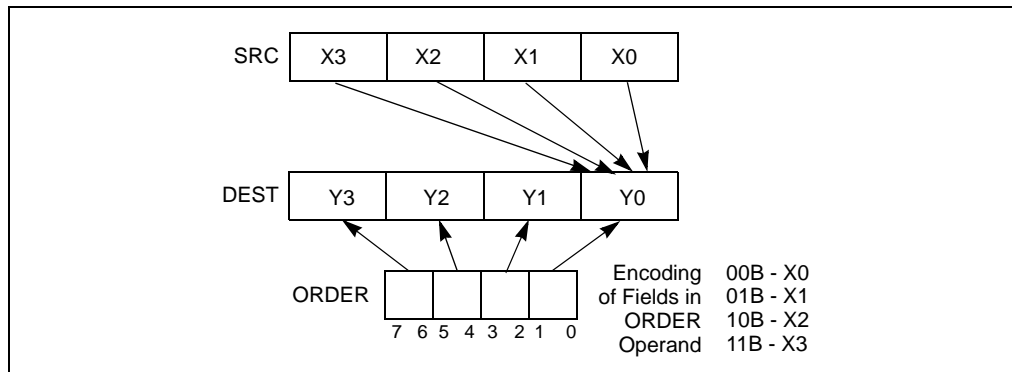


Figure 4-6. PSHUFD Instruction Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).



**Operation**

```

DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

PSHUFD      __m128i _mm_shuffle_epi32(__m128i a, int n)

```

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.

**Real-Address Mode Exceptions**

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.

## INSTRUCTION SET REFERENCE, N-Z

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.

## PSHUFHW—Shuffle Packed High Words

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 70 /r ib	PSHUFHW <i>xmm1</i> , <i>xmm2</i> / <i>m128</i> , <i>imm8</i>	Valid	Valid	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .

### Description

Copies words from the high quadword of the source operand (second operand) and inserts them in the high quadword of the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-6. For the PSHUFHW instruction, each 2-bit field in the order operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the order operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand.

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[63:0] ← SRC[63:0];
DEST[79:64] ← (SRC >> (ORDER[1:0] * 16))[79:64];
DEST[95:80] ← (SRC >> (ORDER[3:2] * 16))[79:64];
DEST[111:96] ← (SRC >> (ORDER[5:4] * 16))[79:64];
DEST[127:112] ← (SRC >> (ORDER[7:6] * 16))[79:64];
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
PSHUFHW    __m128i _mm_shufflehi_epi16(__m128i a, int n)
```

### Flags Affected

None.

### Numeric Exceptions

None.

## INSTRUCTION SET REFERENCE, N-Z

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.

### Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.

## PSHUFLW—Shuffle Packed Low Words

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F 70 /r ib	PSHUFLW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	Valid	Valid	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .

### Description

Copies words from the low quadword of the source operand (second operand) and inserts them in the low quadword of the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-6. For the PSHUFLW instruction, each 2-bit field in the order operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the order operand fields select words (0, 1, 2, or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand.

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[15:0] ← (SRC >> (ORDER[1:0] * 16))[15:0];
DEST[31:16] ← (SRC >> (ORDER[3:2] * 16))[15:0];
DEST[47:32] ← (SRC >> (ORDER[5:4] * 16))[15:0];
DEST[63:48] ← (SRC >> (ORDER[7:6] * 16))[15:0];
DEST[127:64] ← SRC[127:64];
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
PSHUFLW    __m128i _mm_shufflelo_epi16(__m128i a, int n)
```

### Flags Affected

None.

### Numeric Exceptions

None.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.

**Real-Address Mode Exceptions**

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.



## PSHUFW—Shuffle Packed Words

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 70 /r ib	PSHUFW <i>mm1</i> , <i>mm2/m64</i> , <i>imm8</i>	Valid	Valid	Shuffle the words in <i>mm2/m64</i> based on the encoding in <i>imm8</i> and store the result in <i>mm1</i> .

### Description

Copies words from the source operand (second operand) and inserts them in the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-6. For the PSHUFW instruction, each 2-bit field in the order operand selects the contents of one word location in the destination operand. The encodings of the order operand fields select words from the source operand to be copied to the destination operand.

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register. The order operand is an 8-bit immediate. Note that this instruction permits a word in the source operand to be copied to more than one word location in the destination operand.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[15:0] ← (SRC >> (ORDER[1:0] * 16))[15:0];
DEST[31:16] ← (SRC >> (ORDER[3:2] * 16))[15:0];
DEST[47:32] ← (SRC >> (ORDER[5:4] * 16))[15:0];
DEST[63:48] ← (SRC >> (ORDER[7:6] * 16))[15:0];
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
PSHUFW      __m64 _mm_shuffle_pi16(__m64 a, int n)
```

### Flags Affected

None.

### Numeric Exceptions

None.

## INSTRUCTION SET REFERENCE, N-Z

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.
#NM	If CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.
#NM	If CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#UD	If CR0.EM[bit 2] = 1.
#NM	If CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**PSLLDQ—Shift Double Quadword Left Logical**

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 73 /7 ib	PSLLDQ <i>xmm1</i> , <i>imm8</i>	Valid	Valid	Shift <i>xmm1</i> left by <i>imm8</i> bytes while shifting in 0s.

**Description**

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The destination operand is an XMM register. The count operand is an 8-bit immediate.

**Operation**

```
TEMP ← COUNT;
IF (TEMP > 15) THEN TEMP ← 16; FI;
DEST ← DEST << (TEMP * 8);
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
PSLLDQ    __m128i _mm_slli_si128 ( __m128i a, int imm)
```

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Protected Mode Exceptions**

```
#UD          If CR0.EM[bit 2] = 1.
              If CR4.OSFXSR[bit 9] = 0.
              If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM          If CR0.TS[bit 3] = 1.
```

**Real-Address Mode Exceptions**

Same exceptions as in Protected Mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Protected Mode.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

Same exceptions as in Protected Mode.

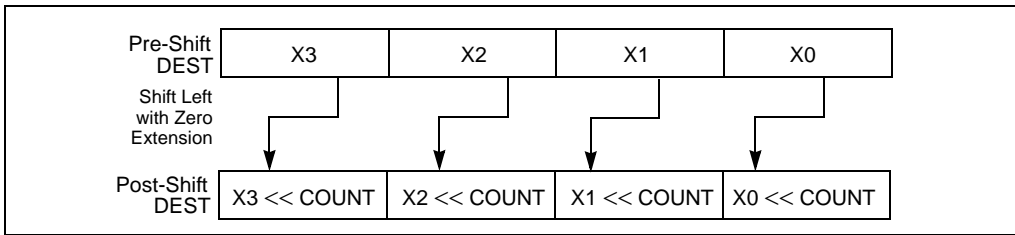
## PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F F1 /r	PSLLW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Shift words in <i>mm</i> left <i>mm/m64</i> while shifting in 0s.
66 0F F1 /r	PSLLW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Shift words in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 71 /6 ib	PSLLW <i>xmm1</i> , <i>imm8</i>	Valid	Valid	Shift words in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 71 /6 ib	PSLLW <i>xmm1</i> , <i>imm8</i>	Valid	Valid	Shift words in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
0F F2 /r	PSLLD <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Shift doublewords in <i>mm</i> left by <i>mm/m64</i> while shifting in 0s.
66 0F F2 /r	PSLLD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Shift doublewords in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 72 /6 ib	PSLLD <i>mm</i> , <i>imm8</i>	Valid	Valid	Shift doublewords in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 72 /6 ib	PSLLD <i>xmm1</i> , <i>imm8</i>	Valid	Valid	Shift doublewords in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
0F F3 /r	PSLLQ <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Shift quadword in <i>mm</i> left by <i>mm/m64</i> while shifting in 0s.
66 0F F3 /r	PSLLQ <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Shift quadwords in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
0F 73 /6 ib	PSLLQ <i>mm</i> , <i>imm8</i>	Valid	Valid	Shift quadword in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 0F 73 /6 ib	PSLLQ <i>xmm1</i> , <i>imm8</i>	Valid	Valid	Shift quadwords in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.

**Description**

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-7 gives an example of shifting words in a 64-bit operand.

The destination operand may be an MMX technology register or an XMM register; the count operand can be either an MMX technology register or an 64-bit memory location, an XMM register or a 128-bit memory location, or an 8-bit immediate. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.



**Figure 4-7. PSSLW, PSLLD, and PSSLQ Instruction Operation Using 64-bit Operand**

The PSSLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the PSLLD instruction shifts each of the doublewords in the destination operand; and the PSSLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PSSLW instruction with 64-bit operand:

```
IF (COUNT > 15)
  THEN
    DEST[64:0] ← 0000000000000000H;
  ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] << COUNT);
  FI;
```

PSLLD instruction with 64-bit operand:

```
IF (COUNT > 31)
  THEN
    DEST[64:0] ← 0000000000000000H;
  ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
  FI;
```

PSSLQ instruction with 64-bit operand:

```
IF (COUNT > 63)
  THEN
    DEST[64:0] ← 0000000000000000H;
  ELSE
    DEST ← ZeroExtend(DEST << COUNT);
  FI;
```

## INSTRUCTION SET REFERENCE, N-Z

PSLLW instruction with 128-bit operand:

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
THEN
    DEST[128:0] ← 00000000000000000000000000000000H;
ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(DEST[127:112] << COUNT);
FI;
```

PSLLD instruction with 128-bit operand:

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 31)
THEN
    DEST[128:0] ← 00000000000000000000000000000000H;
ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
    (* Repeat shift operation for 2nd and 3rd doublewords *)
    DEST[127:96] ← ZeroExtend(DEST[127:96] << COUNT);
FI;
```

PSLLQ instruction with 128-bit operand:

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 63)
THEN
    DEST[128:0] ← 00000000000000000000000000000000H;
ELSE
    DEST[63:0] ← ZeroExtend(DEST[63:0] << COUNT);
    DEST[127:64] ← ZeroExtend(DEST[127:64] << COUNT);
FI;
```

### Intel C/C++ Compiler Intrinsic Equivalents

PSLLW	__m64 _mm_slli_pi16 (__m64 m, int count)
PSLLW	__m64 _mm_sll_pi16(__m64 m, __m64 count)
PSLLW	__m128i _mm_slli_pi16(__m64 m, int count)
PSLLW	__m128i _mm_slli_pi16(__m128i m, __m128i count)
PSLLD	__m64 _mm_slli_pi32(__m64 m, int count)
PSLLD	__m64 _mm_sll_pi32(__m64 m, __m64 count)
PSLLD	__m128i _mm_slli_epi32(__m128i m, int count)
PSLLD	__m128i _mm_sll_epi32(__m128i m, __m128i count)
PSLLQ	__m64 _mm_slli_si64(__m64 m, int count)
PSLLQ	__m64 _mm_sll_si64(__m64 m, __m64 count)
PSLLQ	__m128i _mm_slli_si64(__m128i m, int count)



PSLLQ            \_\_m128i \_\_mm\_sll\_si64(\_\_m128i m, \_\_m128i count)

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

## INSTRUCTION SET REFERENCE, N-Z

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#UD If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PSRAW/PSRAD—Shift Packed Data Right Arithmetic

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F E1 /r	PSRAW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Shift words in <i>mm</i> right by <i>mm/m64</i> while shifting in sign bits.
66 0F E1 /r	PSRAW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Shift words in <i>xmm1</i> right by <i>xmm2/m128</i> while shifting in sign bits.
0F 71 /4 ib	PSRAW <i>mm</i> , <i>imm8</i>	Valid	Valid	Shift words in <i>mm</i> right by <i>imm8</i> while shifting in sign bits.
66 0F 71 /4 ib	PSRAW <i>xmm1</i> , <i>imm8</i>	Valid	Valid	Shift words in <i>xmm1</i> right by <i>imm8</i> while shifting in sign bits.
0F E2 /r	PSRAD <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Shift doublewords in <i>mm</i> right by <i>mm/m64</i> while shifting in sign bits.
66 0F E2 /r	PSRAD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Shift doubleword in <i>xmm1</i> right by <i>xmm2/m128</i> while shifting in sign bits.
0F 72 /4 ib	PSRAD <i>mm</i> , <i>imm8</i>	Valid	Valid	Shift doublewords in <i>mm</i> right by <i>imm8</i> while shifting in sign bits.
66 0F 72 /4 ib	PSRAD <i>xmm1</i> , <i>imm8</i>	Valid	Valid	Shift doublewords in <i>xmm1</i> right by <i>imm8</i> while shifting in sign bits.

### Description

Shifts the bits in the individual data elements (words or doublewords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words) or 31 (for doublewords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-8 gives an example of shifting words in a 64-bit operand.)

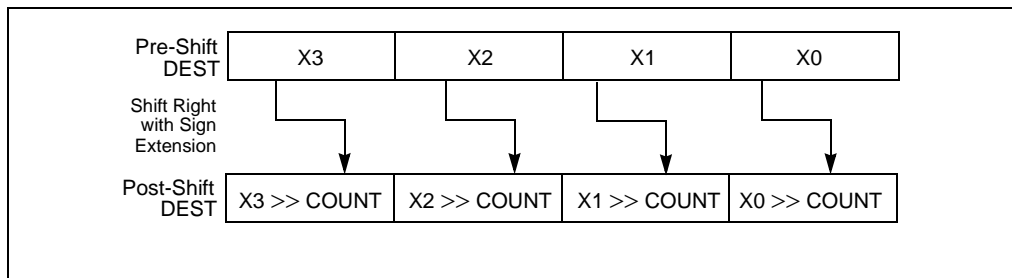


Figure 4-8. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

## INSTRUCTION SET REFERENCE, N-Z

The destination operand may be an MMX technology register or an XMM register; the count operand can be either an MMX technology register or a 64-bit memory location, an XMM register or a 128-bit memory location, or an 8-bit immediate. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

The PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PSRAW instruction with 64-bit operand:

```
IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd words *)
DEST[63:48] ← SignExtend(DEST[63:48] >> COUNT);
```

PSRAD instruction with 64-bit operand:

```
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(DEST[31:0] >> COUNT);
DEST[63:32] ← SignExtend(DEST[63:32] >> COUNT);
```

PSRAW instruction with 128-bit operand:

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] ← SignExtend(DEST[127:112] >> COUNT);
```

PSRAD instruction with 128-bit operand:

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(DEST[31:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd doublewords *)
DEST[127:96] ← SignExtend(DEST[127:96] >> COUNT);
```

**Intel C/C++ Compiler Intrinsic Equivalents**

PSRAW	<code>__m64 __mm_srai_pi16 (__m64 m, int count)</code>
PSRAW	<code>__m64 __mm_sraw_pi16 (__m64 m, __m64 count)</code>
PSRAD	<code>__m64 __mm_srai_pi32 (__m64 m, int count)</code>
PSRAD	<code>__m64 __mm_sra_pi32 (__m64 m, __m64 count)</code>
PSRAW	<code>__m128i __mm_srai_epi16(__m128i m, int count)</code>
PSRAW	<code>__m128i __mm_sra_epi16(__m128i m, __m128i count)</code>
PSRAD	<code>__m128i __mm_srai_epi32 (__m128i m, int count)</code>
PSRAD	<code>__m128i __mm_sra_epi32 (__m128i m, __m128i count)</code>

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
--------	---

## INSTRUCTION SET REFERENCE, N-Z

#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PSRLDQ—Shift Double Quadword Right Logical

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 73 /3 ib	PSRLDQ <i>xmm1, imm8</i>	Valid	Valid	Shift <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.

### Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The destination operand is an XMM register. The count operand is an 8-bit immediate.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
TEMP ← COUNT;
IF (TEMP > 15) THEN TEMP ← 16; FI;
DEST ← DEST >> (temp * 8);
```

### Intel C/C++ Compiler Intrinsic Equivalents

```
PSRLDQ    __m128i _mm_srli_si128 ( __m128i a, int imm)
```

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

```
#UD          If CR0.EM[bit 2] = 1.
              If CR4.OSFXSR[bit 9] = 0.
              If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM          If CR0.TS[bit 3] = 1.
```

## **INSTRUCTION SET REFERENCE, N-Z**

### **Real-Address Mode Exceptions**

Same exceptions as in Protected Mode.

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Protected Mode.

### **Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

### **64-Bit Mode Exceptions**

Same exceptions as in Protected Mode.

### **Numeric Exceptions**

None.



## PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

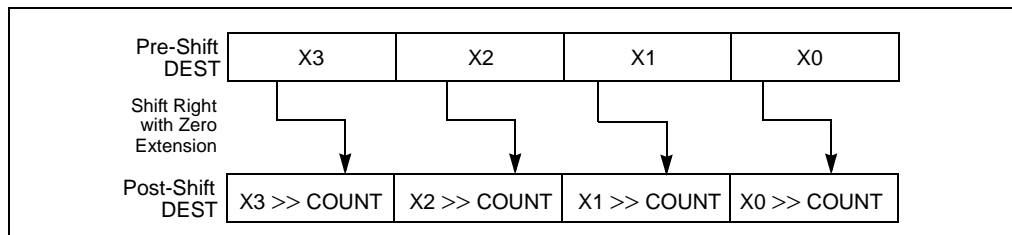
Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F D1 /r	PSRLW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Shift words in <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 0F D1 /r	PSRLW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Shift words in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
0F 71 /2 ib	PSRLW <i>mm</i> , <i>imm8</i>	Valid	Valid	Shift words in <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 0F 71 /2 ib	PSRLW <i>xmm1</i> , <i>imm8</i>	Valid	Valid	Shift words in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
0F D2 /r	PSRLD <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Shift doublewords in <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 0F D2 /r	PSRLD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Shift doublewords in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
0F 72 /2 ib	PSRLD <i>mm</i> , <i>imm8</i>	Valid	Valid	Shift doublewords in <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 0F 72 /2 ib	PSRLD <i>xmm1</i> , <i>imm8</i>	Valid	Valid	Shift doublewords in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
0F D3 /r	PSRLQ <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Shift <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 0F D3 /r	PSRLQ <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Shift quadwords in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
0F 73 /2 ib	PSRLQ <i>mm</i> , <i>imm8</i>	Valid	Valid	Shift <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 0F 73 /2 ib	PSRLQ <i>xmm1</i> , <i>imm8</i>	Valid	Valid	Shift quadwords in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.

### Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-9 gives an example of shifting words in a 64-bit operand.

The destination operand may be an MMX technology register or an XMM register; the count operand can be either an MMX technology register or a 64-bit memory location, an XMM register or a 128-bit memory location, or an 8-bit immediate. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

## INSTRUCTION SET REFERENCE, N-Z



**Figure 4-9. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand**

The PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the PSRLD instruction shifts each of the double-words in the destination operand; and the PSRLQ instruction shifts the quadword (or quad-words) in the destination operand.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PSRLW instruction with 64-bit operand:

```

IF (COUNT > 15)
  THEN
    DEST[64:0] ← 0000000000000000H
  ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] >> COUNT);
  FI;

```

PSRLD instruction with 64-bit operand:

```

IF (COUNT > 31)
  THEN
    DEST[64:0] ← 0000000000000000H
  ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] >> COUNT);
  FI;

```

PSRLQ instruction with 64-bit operand:

```

IF (COUNT > 63)
  THEN
    DEST[64:0] ← 0000000000000000H
  ELSE
    DEST ← ZeroExtend(DEST >> COUNT);
  FI;

```

PSRLW instruction with 128-bit operand:

```

COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
  THEN
    DEST[128:0] ← 00000000000000000000000000000000H
  ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(DEST[127:112] >> COUNT);
  FI;

```

PSRLD instruction with 128-bit operand:

```

COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 31)
  THEN
    DEST[128:0] ← 00000000000000000000000000000000H
  ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd doublewords *)
    DEST[127:96] ← ZeroExtend(DEST[127:96] >> COUNT);
  FI;

```

PSRLQ instruction with 128-bit operand:

```

COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
  THEN
    DEST[128:0] ← 00000000000000000000000000000000H
  ELSE
    DEST[63:0] ← ZeroExtend(DEST[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(DEST[127:64] >> COUNT);
  FI;

```

### Intel C/C++ Compiler Intrinsic Equivalents

PSRLW	__m64 __mm_srli_pi16(__m64 m, int count)
PSRLW	__m64 __mm_srl_pi16 (__m64 m, __m64 count)
PSRLW	__m128i __mm_srli_epi16 (__m128i m, int count)
PSRLW	__m128i __mm_srl_epi16 (__m128i m, __m128i count)
PSRLD	__m64 __mm_srli_pi32 (__m64 m, int count)
PSRLD	__m64 __mm_srl_pi32 (__m64 m, __m64 count)
PSRLD	__m128i __mm_srli_epi32 (__m128i m, int count)
PSRLD	__m128i __mm_srl_epi32 (__m128i m, __m128i count)
PSRLQ	__m64 __mm_srli_si64 (__m64 m, int count)
PSRLQ	__m64 __mm_srl_si64 (__m64 m, __m64 count)
PSRLQ	__m128i __mm_srli_epi64 (__m128i m, int count)

## INSTRUCTION SET REFERENCE, N-Z

PSRLQ            \_\_m128i \_\_mm\_srl\_epi64 (\_\_m128i m, \_\_m128i count)

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

**64-Bit Mode Exceptions**

#GP(0) If the memory address is in a non-canonical form.  
(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#UD If CR0.EM[bit 2] = 1.  
(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  
(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**PSUBB/PSUBW/PSUBD—Subtract Packed Integers**

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F F8 /r	PSUBB <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Subtract packed byte integers in <i>mm/m64</i> from packed byte integers in <i>mm</i> .
66 0F F8 /r	PSUBB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Subtract packed byte integers in <i>xmm2/m128</i> from packed byte integers in <i>xmm1</i> .
0F F9 /r	PSUBW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Subtract packed word integers in <i>mm/m64</i> from packed word integers in <i>mm</i> .
66 0F F9 /r	PSUBW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Subtract packed word integers in <i>xmm2/m128</i> from packed word integers in <i>xmm1</i> .
0F FA /r	PSUBD <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Subtract packed doubleword integers in <i>mm/m64</i> from packed doubleword integers in <i>mm</i> .
66 0F FA /r	PSUBD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Subtract packed doubleword integers in <i>xmm2/mem128</i> from packed doubleword integers in <i>xmm1</i> .

**Description**

Performs an SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of an SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the PSUBB, PSUBW, and PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

PSUBB instruction with 64-bit operands:

$DEST[7:0] \leftarrow DEST[7:0] - SRC[7:0];$

(\* Repeat subtract operation for 2nd through 7th byte \*)

$DEST[63:56] \leftarrow DEST[63:56] - SRC[63:56];$

PSUBB instruction with 128-bit operands:

$DEST[7:0] \leftarrow DEST[7:0] - SRC[7:0];$

(\* Repeat subtract operation for 2nd through 14th byte \*)

$DEST[127:120] \leftarrow DEST[111:120] - SRC[127:120];$

PSUBW instruction with 64-bit operands:

$DEST[15:0] \leftarrow DEST[15:0] - SRC[15:0];$

(\* Repeat subtract operation for 2nd and 3rd word \*)

$DEST[63:48] \leftarrow DEST[63:48] - SRC[63:48];$

PSUBW instruction with 128-bit operands:

$DEST[15:0] \leftarrow DEST[15:0] - SRC[15:0];$

(\* Repeat subtract operation for 2nd through 7th word \*)

$DEST[127:112] \leftarrow DEST[127:112] - SRC[127:112];$

PSUBD instruction with 64-bit operands:

$DEST[31:0] \leftarrow DEST[31:0] - SRC[31:0];$

$DEST[63:32] \leftarrow DEST[63:32] - SRC[63:32];$

PSUBD instruction with 128-bit operands:

$DEST[31:0] \leftarrow DEST[31:0] - SRC[31:0];$

(\* Repeat subtract operation for 2nd and 3rd doubleword \*)

$DEST[127:96] \leftarrow DEST[127:96] - SRC[127:96];$

## Intel C/C++ Compiler Intrinsic Equivalents

PSUBB `__m64 _mm_sub_pi8(__m64 m1, __m64 m2)`

PSUBW `__m64 _mm_sub_pi16(__m64 m1, __m64 m2)`

PSUBD `__m64 _mm_sub_pi32(__m64 m1, __m64 m2)`

PSUBB `__m128i _mm_sub_epi8 (__m128i a, __m128i b)`

PSUBW `__m128i _mm_sub_epi16 (__m128i a, __m128i b)`

PSUBD `__m128i _mm_sub_epi32 (__m128i a, __m128i b)`

## INSTRUCTION SET REFERENCE, N-Z

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.



**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#UD If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PSUBQ—Subtract Packed Quadword Integers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F FB /r	PSUBQ <i>mm1, mm2/m64</i>	Valid	Valid	Subtract quadword integer in <i>mm1</i> from <i>mm2 /m64</i> .
66 0F FB /r	PSUBQ <i>xmm1, xmm2/m128</i>	Valid	Valid	Subtract packed quadword integers in <i>xmm1</i> from <i>xmm2 /m128</i> .

### Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location, or it can be two packed quadword integers stored in an XMM register or an 128-bit memory location. The destination operand can be a quadword integer stored in an MMX technology register or two packed quadword integers stored in an XMM register. When packed quadword operands are used, an SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the PSUBQ instruction can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PSUBQ instruction with 64-Bit operands:

$$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] - \text{SRC}[63:0];$$

PSUBQ instruction with 128-Bit operands:

$$\begin{aligned} \text{DEST}[63:0] &\leftarrow \text{DEST}[63:0] - \text{SRC}[63:0]; \\ \text{DEST}[127:64] &\leftarrow \text{DEST}[127:64] - \text{SRC}[127:64]; \end{aligned}$$

### Intel C/C++ Compiler Intrinsic Equivalents

PSUBQ            `__m64 _mm_sub_si64(__m64 m1, __m64 m2)`

PSUBQ            `__m128i _mm_sub_epi64(__m128i m1, __m128i m2)`

### Flags Affected

None.

**Numeric Exceptions**

None.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

## INSTRUCTION SET REFERENCE, N-Z

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.  (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F E8 /r	PSUBSB <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Subtract signed packed bytes in <i>mm/m64</i> from signed packed bytes in <i>mm</i> and saturate results.
66 0F E8 /r	PSUBSB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Subtract packed signed byte integers in <i>xmm2/m128</i> from packed signed byte integers in <i>xmm1</i> and saturate results.
0F E9 /r	PSUBSW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Subtract signed packed words in <i>mm/m64</i> from signed packed words in <i>mm</i> and saturate results.
66 0F E9 /r	PSUBSW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Subtract packed signed word integers in <i>xmm2/m128</i> from packed signed word integers in <i>xmm1</i> and saturate results.

### Description

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of an SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## INSTRUCTION SET REFERENCE, N-Z

### Operation

PSUBSB instruction with 64-bit operands:

DEST[7:0] ← SaturateToSignedByte (DEST[7:0] – SRC (7:0));  
(\* Repeat subtract operation for 2nd through 7th bytes \*)  
DEST[63:56] ← SaturateToSignedByte (DEST[63:56] – SRC[63:56] );

PSUBSB instruction with 128-bit operands:

DEST[7:0] ← SaturateToSignedByte (DEST[7:0] – SRC[7:0]);  
(\* Repeat subtract operation for 2nd through 14th bytes \*)  
DEST[127:120] ← SaturateToSignedByte (DEST[111:120] – SRC[127:120]);

PSUBSW instruction with 64-bit operands

DEST[15:0] ← SaturateToSignedWord (DEST[15:0] – SRC[15:0] );  
(\* Repeat subtract operation for 2nd and 7th words \*)  
DEST[63:48] ← SaturateToSignedWord (DEST[63:48] – SRC[63:48] );

PSUBSW instruction with 128-bit operands

DEST[15:0] ← SaturateToSignedWord (DEST[15:0] – SRC[15:0]);  
(\* Repeat subtract operation for 2nd through 7th words \*)  
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] – SRC[127:112]);

### Intel C/C++ Compiler Intrinsic Equivalents

PSUBSB        \_\_m64 \_mm\_subs\_pi8(\_\_m64 m1, \_\_m64 m2)  
PSUBSB        \_\_m128i \_mm\_subs\_epi8(\_\_m128i m1, \_\_m128i m2)  
PSUBSW        \_\_m64 \_mm\_subs\_pi16(\_\_m64 m1, \_\_m64 m2)  
PSUBSW        \_\_m128i \_mm\_subs\_epi16(\_\_m128i m1, \_\_m128i m2)

### Flags Affected

None.

### Numeric Exceptions

None.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  (128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form. (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F D8 /r	PSUBUSB <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Subtract unsigned packed bytes in <i>mm/m64</i> from unsigned packed bytes in <i>mm</i> and saturate result.
66 0F D8 /r	PSUBUSB <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Subtract packed unsigned byte integers in <i>xmm2/m128</i> from packed unsigned byte integers in <i>xmm1</i> and saturate result.
0F D9 /r	PSUBUSW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Subtract unsigned packed words in <i>mm/m64</i> from unsigned packed words in <i>mm</i> and saturate result.
66 0F D9 /r	PSUBUSW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Subtract packed unsigned word integers in <i>xmm2/m128</i> from packed unsigned word integers in <i>xmm1</i> and saturate result.

### Description

Performs an SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of an SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## INSTRUCTION SET REFERENCE, N-Z

### Operation

PSUBUSB instruction with 64-bit operands:

DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC (7:0) );  
(\* Repeat add operation for 2nd through 7th bytes \*)  
DEST[63:56] ← SaturateToUnsignedByte (DEST[63:56] – SRC[63:56]);

PSUBUSB instruction with 128-bit operands:

DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC[7:0]);  
(\* Repeat add operation for 2nd through 14th bytes \*)  
DEST[127:120] ← SaturateToUnsignedByte (DEST[127:120] – SRC[127:120]);

PSUBUSW instruction with 64-bit operands:

DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0] );  
(\* Repeat add operation for 2nd and 3rd words \*)  
DEST[63:48] ← SaturateToUnsignedWord (DEST[63:48] – SRC[63:48] );

PSUBUSW instruction with 128-bit operands:

DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0]);  
(\* Repeat add operation for 2nd through 7th words \*)  
DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] – SRC[127:112]);

### Intel C/C++ Compiler Intrinsic Equivalents

PSUBUSB     \_\_m64 \_mm\_subs\_pu8(\_\_m64 m1, \_\_m64 m2)  
PSUBUSB     \_\_m128i \_mm\_subs\_epu8(\_\_m128i m1, \_\_m128i m2)  
PSUBUSW     \_\_m64 \_mm\_subs\_pu16(\_\_m64 m1, \_\_m64 m2)  
PSUBUSW     \_\_m128i \_mm\_subs\_epu16(\_\_m128i m1, \_\_m128i m2)

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0)       If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
  
(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  
  
#SS(0)       If a memory operand effective address is outside the SS segment limit.  
  
#UD           If CR0.EM[bit 2] = 1.  
  
(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  
  
(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.  (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.

## INSTRUCTION SET REFERENCE, N-Z

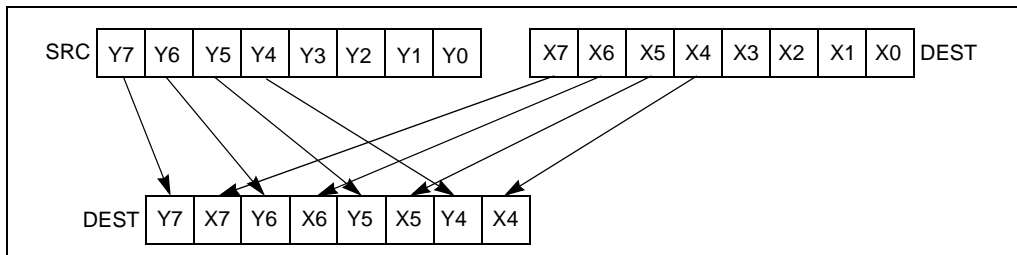
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 68 /r	PUNPCKHBW <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Unpack and interleave high-order bytes from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 68 /r	PUNPCKHBW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Unpack and interleave high-order bytes from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 69 /r	PUNPCKHWD <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Unpack and interleave high-order words from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 69 /r	PUNPCKHWD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Unpack and interleave high-order words from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 6A /r	PUNPCKHDQ <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Unpack and interleave high-order doublewords from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 0F 6A /r	PUNPCKHDQ <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Unpack and interleave high-order doublewords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
66 0F 6D /r	PUNPCKHQDQ <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Unpack and interleave high-order quadwords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .

### Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-10 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.



**Figure 4-10. PUNPCKHBW Instruction Operation Using 64-bit Operands**

## INSTRUCTION SET REFERENCE, N-Z

The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

PUNPCKHBW instruction with 64-bit operands:

```
DEST[7:0] ← DEST[39:32];
DEST[15:8] ← SRC[39:32];
DEST[23:16] ← DEST[47:40];
DEST[31:24] ← SRC[47:40];
DEST[39:32] ← DEST[55:48];
DEST[47:40] ← SRC[55:48];
DEST[55:48] ← DEST[63:56];
DEST[63:56] ← SRC[63:56];
```

PUNPCKHW instruction with 64-bit operands:

```
DEST[15:0] ← DEST[47:32];
DEST[31:16] ← SRC[47:32];
DEST[47:32] ← DEST[63:48];
DEST[63:48] ← SRC[63:48];
```

PUNPCKHDQ instruction with 64-bit operands:

```
DEST[31:0] ← DEST[63:32];
DEST[63:32] ← SRC[63:32];
```

PUNPCKHBW instruction with 128-bit operands:

$DEST[7:0] \leftarrow DEST[71:64];$   
 $DEST[15:8] \leftarrow SRC[71:64];$   
 $DEST[23:16] \leftarrow DEST[79:72];$   
 $DEST[31:24] \leftarrow SRC[79:72];$   
 $DEST[39:32] \leftarrow DEST[87:80];$   
 $DEST[47:40] \leftarrow SRC[87:80];$   
 $DEST[55:48] \leftarrow DEST[95:88];$   
 $DEST[63:56] \leftarrow SRC[95:88];$   
 $DEST[71:64] \leftarrow DEST[103:96];$   
 $DEST[79:72] \leftarrow SRC[103:96];$   
 $DEST[87:80] \leftarrow DEST[111:104];$   
 $DEST[95:88] \leftarrow SRC[111:104];$   
 $DEST[103:96] \leftarrow DEST[119:112];$   
 $DEST[111:104] \leftarrow SRC[119:112];$   
 $DEST[119:112] \leftarrow DEST[127:120];$   
 $DEST[127:120] \leftarrow SRC[127:120];$

PUNPCKHWD instruction with 128-bit operands:

$DEST[15:0] \leftarrow DEST[79:64];$   
 $DEST[31:16] \leftarrow SRC[79:64];$   
 $DEST[47:32] \leftarrow DEST[95:80];$   
 $DEST[63:48] \leftarrow SRC[95:80];$   
 $DEST[79:64] \leftarrow DEST[111:96];$   
 $DEST[95:80] \leftarrow SRC[111:96];$   
 $DEST[111:96] \leftarrow DEST[127:112];$   
 $DEST[127:112] \leftarrow SRC[127:112];$

PUNPCKHDQ instruction with 128-bit operands:

$DEST[31:0] \leftarrow DEST[95:64];$   
 $DEST[63:32] \leftarrow SRC[95:64];$   
 $DEST[95:64] \leftarrow DEST[127:96];$   
 $DEST[127:96] \leftarrow SRC[127:96];$

PUNPCKHQDQ instruction:

$DEST[63:0] \leftarrow DEST[127:64];$   
 $DEST[127:64] \leftarrow SRC[127:64];$

## INSTRUCTION SET REFERENCE, N-Z

### Intel C/C++ Compiler Intrinsic Equivalents

PUNPCKHBW `__m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)`  
PUNPCKHBW `__m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)`  
PUNPCKHWD `__m64 _mm_unpackhi_pi16(__m64 m1, __m64 m2)`  
PUNPCKHWD `__m128i _mm_unpackhi_epi16(__m128i m1, __m128i m2)`  
PUNPCKHDQ `__m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)`  
PUNPCKHDQ `__m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)`  
PUNPCKHQDQ `__m128i _mm_unpackhi_epi64 (__m128i a, __m128i b)`

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If CR0.EM[bit 2] = 1.  
(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  
If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to FFFFH.  
(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.



#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. (128-bit version only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ— Unpack Low Data

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 60 /r	PUNPCKLBW <i>mm</i> , <i>mm/m32</i>	Valid	Valid	Interleave low-order bytes from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 0F 60 /r	PUNPCKLBW <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Interleave low-order bytes from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 61 /r	PUNPCKLWD <i>mm</i> , <i>mm/m32</i>	Valid	Valid	Interleave low-order words from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 0F 61 /r	PUNPCKLWD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Interleave low-order words from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
0F 62 /r	PUNPCKLDQ <i>mm</i> , <i>mm/m32</i>	Valid	Valid	Interleave low-order doublewords from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 0F 62 /r	PUNPCKLDQ <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Interleave low-order doublewords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
66 0F 6C /r	PUNPCKLQDQ <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Interleave low-order quadword from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> register.

### Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-11 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

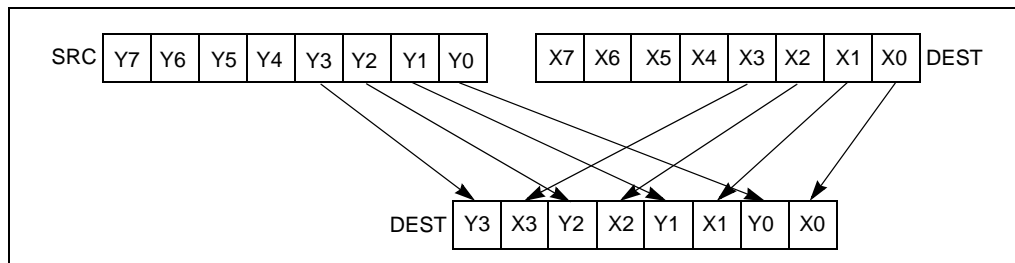


Figure 4-11. PUNPCKLBW Instruction Operation Using 64-bit Operands

The source operand can be an MMX technology register or a 32-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

PUNPCKLBW instruction with 64-bit operands:

```
DEST[63:56] ← SRC[31:24];
DEST[55:48] ← DEST[31:24];
DEST[47:40] ← SRC[23:16];
DEST[39:32] ← DEST[23:16];
DEST[31:24] ← SRC[15:8];
DEST[23:16] ← DEST[15:8];
DEST[15:8] ← SRC[7:0];
DEST[7:0] ← DEST[7:0];
```

PUNPCKLWD instruction with 64-bit operands:

```
DEST[63:48] ← SRC[31:16];
DEST[47:32] ← DEST[31:16];
DEST[31:16] ← SRC[15:0];
DEST[15:0] ← DEST[15:0];
```

PUNPCKLDQ instruction with 64-bit operands:

```
DEST[63:32] ← SRC[31:0];
DEST[31:0] ← DEST[31:0];
```

PUNPCKLBW instruction with 128-bit operands:

```
DEST[7:0] ← DEST[7:0];
DEST[15:8] ← SRC[7:0];
DEST[23:16] ← DEST[15:8];
DEST[31:24] ← SRC[15:8];
DEST[39:32] ← DEST[23:16];
```

## INSTRUCTION SET REFERENCE, N-Z

DEST[47:40] ← SRC[23:16];  
DEST[55:48] ← DEST[31:24];  
DEST[63:56] ← SRC[31:24];  
DEST[71:64] ← DEST[39:32];  
DEST[79:72] ← SRC[39:32];  
DEST[87:80] ← DEST[47:40];  
DEST[95:88] ← SRC[47:40];  
DEST[103:96] ← DEST[55:48];  
DEST[111:104] ← SRC[55:48];  
DEST[119:112] ← DEST[63:56];  
DEST[127:120] ← SRC[63:56];

PUNPCKLWD instruction with 128-bit operands:

DEST[15:0] ← DEST[15:0];  
DEST[31:16] ← SRC[15:0];  
DEST[47:32] ← DEST[31:16];  
DEST[63:48] ← SRC[31:16];  
DEST[79:64] ← DEST[47:32];  
DEST[95:80] ← SRC[47:32];  
DEST[111:96] ← DEST[63:48];  
DEST[127:112] ← SRC[63:48];

PUNPCKLDQ instruction with 128-bit operands:

DEST[31:0] ← DEST[31:0];  
DEST[63:32] ← SRC[31:0];  
DEST[95:64] ← DEST[63:32];  
DEST[127:96] ← SRC[63:32];

PUNPCKLQDQ

DEST[63:0] ← DEST[63:0];  
DEST[127:64] ← SRC[63:0];

### Intel C/C++ Compiler Intrinsic Equivalents

PUNPCKLBW \_\_m64 \_mm\_unpacklo\_pi8 (\_\_m64 m1, \_\_m64 m2)  
PUNPCKLBW \_\_m128i \_mm\_unpacklo\_epi8 (\_\_m128i m1, \_\_m128i m2)  
PUNPCKLWD \_\_m64 \_mm\_unpacklo\_pi16 (\_\_m64 m1, \_\_m64 m2)  
PUNPCKLWD \_\_m128i \_mm\_unpacklo\_epi16 (\_\_m128i m1, \_\_m128i m2)  
PUNPCKLDQ \_\_m64 \_mm\_unpacklo\_pi32 (\_\_m64 m1, \_\_m64 m2)  
PUNPCKLDQ \_\_m128i \_mm\_unpacklo\_epi32 (\_\_m128i m1, \_\_m128i m2)  
PUNPCKLQDQ \_\_m128i \_mm\_unpacklo\_epi64 (\_\_m128i m1, \_\_m128i m2)

### Flags Affected

None.

**Numeric Exceptions**

None.

**Protected Mode Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.

(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#UD If CR0.EM[bit 2] = 1.

(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.

If CPUID.01H:EDX.SSE2[bit 26] = 0.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

## INSTRUCTION SET REFERENCE, N-Z

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

(128-bit version only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#UD If CR0.EM[bit 2] = 1.

#NM If CR0.TS[bit 3] = 1.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH <i>r/m16</i>	Valid	Valid	Push <i>r/m16</i> .
FF /6	PUSH <i>r/m32</i>	N.E.	Valid	Push <i>r/m32</i> .
FF /6	PUSH <i>r/m64</i>	Valid	N.E.	Push <i>r/m64</i> . Default operand size 64-bits.
50+ <i>rw</i>	PUSH <i>r16</i>	Valid	Valid	Push <i>r16</i> .
50+ <i>rd</i>	PUSH <i>r32</i>	N.E.	Valid	Push <i>r32</i> .
50+ <i>rd</i>	PUSH <i>r64</i>	Valid	N.E.	Push <i>r64</i> . Default operand size 64-bits.
6A	PUSH <i>imm8</i>	Valid	Valid	Push sign-extended <i>imm8</i> . <i>Stack pointer is incremented by the size of stack pointer.</i>
68	PUSH <i>imm16</i>	Valid	Valid	Push sign-extended <i>imm16</i> . <i>Stack pointer is incremented by the size of stack pointer.</i>
68	PUSH <i>imm32</i>	Valid	Valid	Push sign-extended <i>imm32</i> . <i>Stack pointer is incremented by the size of stack pointer.</i>
0E	PUSH CS	Invalid	Valid	Push CS.
16	PUSH SS	Invalid	Valid	Push SS.
1E	PUSH DS	Invalid	Valid	Push DS.
06	PUSH ES	Invalid	Valid	Push ES.
0F A0	PUSH FS	Valid	Valid	Push FS and decrement stack pointer by 16 bits.
0F A0	PUSH FS	N.E.	Valid	Push FS and decrement stack pointer by 32 bits.
0F A0	PUSH FS	Valid	N.E.	Push FS. Default operand size 64-bits. (66H override causes 16-bit operation).
0F A8	PUSH GS	Valid	Valid	Push GS and decrement stack pointer by 16 bits.
0F A8	PUSH GS	N.E.	Valid	Push GS and decrement stack pointer by 32 bits.
0F A8	PUSH GS	Valid	N.E.	Push GS, default operand size 64-bits. (66H override causes 16-bit operation).

**NOTES:**

\* See IA-32 Architecture Compatibility section below.

### Description

Decrements the stack pointer and then stores the source operand on the top of the stack. The address-size attribute of the stack segment determines the stack pointer size (16, 32 or 64 bits). The operand-size attribute of the current code segment determines the amount the stack pointer is decremented (2, 4 or 8 bytes).

In non-64-bit modes: if the address-size and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is decremented by 4. If both attributes are 16, the 16-bit SP register (stack pointer) is decremented by 2.

If the source operand is an immediate and its size is less than the address size of the stack, a sign-extended value is pushed on the stack. If the source operand is the FS or GS and its size is less than the address size of the stack, the zero-extended value is pushed on the stack.

The B flag in the stack segment's segment descriptor determines the stack's address-size attribute. The D flag in the current code segment's segment descriptor (with prefixes), determines the operand-size attribute and the address-size attribute of the source operand. Pushing a 16-bit operand when the stack address-size attribute is 32 can result in a misaligned stack pointer (a stack pointer that is not be aligned on a doubleword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. Thus if a PUSH instruction uses a memory operand in which the ESP register is used for computing the operand address, the address of the operand is computed before the ESP register is decremented.

In the real-address mode, if the ESP or SP register is 1 when the PUSH instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

In 64-bit mode, the instruction's default operation size is 64 bits. In a push, the 64-bit RSP register (stack pointer) is decremented by 8. A 66H override causes 16-bit operation. Note that pushing a 16-bit operand can result in the stack pointer misaligned to 8-byte boundary.

### IA-32 Architecture Compatibility

For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true in the real-address and virtual-8086 modes.) For the Intel 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).



**Operation**

```

IF StackAddrSize = 64
  THEN
    IF OperandSize = 64
      THEN
        RSP ← (RSP – 8);
        IF (SRC is FS or GS)
          THEN
            TEMP = ZeroExtend64(SRC);
          ELSE IF (SRC is IMMEDIATE)
            TEMP = SignExtend64(SRC); FI;
          ELSE
            TEMP = SRC;
        FI
        RSP ← TEMP; (* Push quadword *)
      ELSE (* OperandSize = 16; 66H used *)
        RSP ← (RSP – 2);
        RSP ← SRC; (* Push word *)
      FI;
    ELSE IF StackAddrSize = 32
      THEN
        IF OperandSize = 32
          THEN
            ESP ← (ESP – 4);
            IF (SRC is FS or GS)
              THEN
                TEMP = ZeroExtend32(SRC);
              ELSE IF (SRC is IMMEDIATE)
                TEMP = SignExtend32(SRC); FI;
              ELSE
                TEMP = SRC;
            FI;
            SS:ESP ← TEMP; (* Push doubleword *)
          ELSE (* OperandSize = 16*)
            ESP ← (ESP – 2);
            SS:ESP ← SRC; (* Push word *)
          FI;
        ELSE StackAddrSize = 16
          IF OperandSize = 16
            THEN
              SP ← (SP – 2);
              SS:SP ← SRC; (* Push word *)
            ELSE (* OperandSize = 32 *)
              SP ← (SP – 4);
              SS:SP ← SRC; (* Push doubleword *)
            FI;
          FI;
        FI;
    FI;
  FI;

```

## INSTRUCTION SET REFERENCE, N-Z

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.  If the new value of the SP or ESP register is outside the stack segment limit.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(U)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## PUSHA/PUSHAD—Push All General-Purpose Registers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
60	PUSHA	Invalid	Valid	Push AX, CX, DX, BX, original SP, BP, SI, and DI.
60	PUSHAD	Invalid	Valid	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.

### Description

Pushes the contents of the general-purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, EBP, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). These instructions perform the reverse operation of the POPA/POPAD instructions. The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the “Operation” section below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used. Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when the PUSHA/PUSHAD instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation

IF 64-bit Mode

THEN #UD

FI;

IF OperandSize = 32 (\* PUSHAD instruction \*)

THEN

Temp ← (ESP);

Push(EAX);

Push(ECX);

Push(EDX);

Push(EBX);

Push(Temp);

## INSTRUCTION SET REFERENCE, N-Z

```
    Push(EBP);
    Push(ESI);
    Push(EDI);
ELSE (* OperandSize = 16, PUSHA instruction *)
    Temp ← (SP);
    Push(AX);
    Push(CX);
    Push(DX);
    Push(BX);
    Push(Temp);
    Push(BP);
    Push(SI);
    Push(DI);
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

- #SS(0) If the starting or ending stack address is outside the stack segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

### Real-Address Mode Exceptions

- #GP If the ESP or SP register contains 7, 9, 11, 13, or 15.

### Virtual-8086 Mode Exceptions

- #GP(0) If the ESP or SP register contains 7, 9, 11, 13, or 15.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If an unaligned memory reference is made while alignment checking is enabled.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

- #UD If in 64-bit mode.

## PUSHF/PUSHFD—Push EFLAGS Register onto the Stack

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9C	PUSHF	Valid	Valid	Push lower 16 bits of EFLAGS.
9C	PUSHFD	N.E.	Valid	Push EFLAGS.
9C	PUSHFQ	Valid	N.E.	Push RFLAGS.

### Description

Decrements the stack pointer by 4 (if the current operand-size attribute is 32) and pushes the entire contents of the EFLAGS register onto the stack, or decrements the stack pointer by 2 (if the operand-size attribute is 16) and pushes the lower 16 bits of the EFLAGS register (that is, the FLAGS register) onto the stack. These instructions reverse the operation of the POPF/POPFQ instructions.

When copying the entire EFLAGS register to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, the values for these flags are cleared in the EFLAGS image stored on the stack. See Chapter 3 of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for more information about the EFLAGS register.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In 64-bit mode, the instruction's default operation is to decrement the stack pointer (RSP) by 8 and push RFLAGS on the stack. 16-bit operation is supported using the operand size override prefix 66H. 32-bit operand size cannot be encoded in this mode. When copying RFLAGS to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, values for these flags are cleared in the RFLAGS image stored on the stack.

When in virtual-8086 mode and the I/O privilege level (IOPL) is less than 3, the PUSHF/PUSHFD instruction causes a general protection exception (#GP).

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when the PUSHA/PUSHAD instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

## INSTRUCTION SET REFERENCE, N-Z

### Operation

```
IF (PE = 0) or (PE = 1 and ((VM = 0) or (VM = 1 and IOPL = 3)))
(* Real-Address Mode, Protected mode, or Virtual-8086 mode with IOPL equal to 3 *)
THEN
    IF OperandSize = 32
    THEN
        push (EFLAGS AND 00FCFFFFH);
        (* VM and RF EFLAG bits are cleared in image stored on the stack *)
    ELSE
        push (EFLAGS); (* Lower 16 bits only *)
    FI;
ELSE IF 64-bit MODE (* In 64-bit Mode *)
    IF OperandSize = 64
    THEN
        push (RFLAGS AND 00000000_00FCFFFFH);
        (* VM and RF RFLAG bits are cleared in image stored on the stack; *)
    ELSE
        push (EFLAGS); (* Lower 16 bits only *)
    FI;
ELSE (* In Virtual-8086 Mode with IOPL less than 3 *)
    #GP(0); (* Trap to virtual-8086 monitor *)
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

#SS(0)	If the new value of the ESP register is outside the stack segment boundary.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(U)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

## PXOR—Logical Exclusive OR

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F EF /r	PXOR <i>mm</i> , <i>mm/m64</i>	Valid	Valid	Bitwise XOR of <i>mm/m64</i> and <i>mm</i> .
66 0F EF /r	PXOR <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Bitwise XOR of <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST ← DEST XOR SRC;

### Intel C/C++ Compiler Intrinsic Equivalent

PXOR            \_\_m64 \_\_mm\_xor\_si64 (\_\_m64 m1, \_\_m64 m2)

PXOR            \_\_m128i \_\_mm\_xor\_si128 (\_\_m128i a, \_\_m128i b)

### Flags Affected

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0)            If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
  
(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)            If a memory operand effective address is outside the SS segment limit.



#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1. (128-bit operations only) If CR4.OSFXSR[bit 9] = 0. (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.  (128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
UD	If CR0.EM[bit 2] = 1.  (128-bit operations only) If CR4.OSFXSR[bit 9] = 0.  (128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## RCL/RCR/ROL/ROR—Rotate

Opcode**	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D0 /2	RCL <i>r/m8</i> , 1	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) left once.
REX + D0 /2	RCL <i>r/m8*</i> , 1	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) left once.
D2 /2	RCL <i>r/m8</i> , CL	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) left CL times.
REX + D2 /2	RCL <i>r/m8*</i> , CL	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) left CL times.
C0 /2 <i>ib</i>	RCL <i>r/m8</i> , <i>imm8</i>	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) left <i>imm8</i> times.
REX + C0 /2 <i>ib</i>	RCL <i>r/m8*</i> , <i>imm8</i>	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) left <i>imm8</i> times.
D1 /2	RCL <i>r/m16</i> , 1	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) left once.
D3 /2	RCL <i>r/m16</i> , CL	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) left CL times.
C1 /2 <i>ib</i>	RCL <i>r/m16</i> , <i>imm8</i>	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) left <i>imm8</i> times.
D1 /2	RCL <i>r/m32</i> , 1	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) left once.
REX.W + D1 /2	RCL <i>r/m64</i> , 1	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) left once. Uses a 6 bit count.
D3 /2	RCL <i>r/m32</i> , CL	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) left CL times.
REX.W + D3 /2	RCL <i>r/m64</i> , CL	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) left CL times. Uses a 6 bit count.
C1 /2 <i>ib</i>	RCL <i>r/m32</i> , <i>imm8</i>	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) left <i>imm8</i> times.
REX.W + C1 /2 <i>ib</i>	RCL <i>r/m64</i> , <i>imm8</i>	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) left <i>imm8</i> times. Uses a 6 bit count.
D0 /3	RCR <i>r/m8</i> , 1	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) right once.
REX + D0 /3	RCR <i>r/m8*</i> , 1	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) right once.
D2 /3	RCR <i>r/m8</i> , CL	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) right CL times.
REX + D2 /3	RCR <i>r/m8*</i> , CL	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) right CL times.
C0 /3 <i>ib</i>	RCR <i>r/m8</i> , <i>imm8</i>	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) right <i>imm8</i> times.
REX + C0 /3 <i>ib</i>	RCR <i>r/m8*</i> , <i>imm8</i>	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) right <i>imm8</i> times.
D1 /3	RCR <i>r/m16</i> , 1	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) right once.
D3 /3	RCR <i>r/m16</i> , CL	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) right CL times.
C1 /3 <i>ib</i>	RCR <i>r/m16</i> , <i>imm8</i>	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) right <i>imm8</i> times.
D1 /3	RCR <i>r/m32</i> , 1	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) right once. Uses a 6 bit count.

## INSTRUCTION SET REFERENCE, N-Z

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
REX.W + D1 /3	RCR <i>r/m64</i> , 1	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) right once. Uses a 6 bit count.
D3 /3	RCR <i>r/m32</i> , CL	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) right CL times.
REX.W + D3 /3	RCR <i>r/m64</i> , CL	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) right CL times. Uses a 6 bit count.
C1 /3 <i>ib</i>	RCR <i>r/m32</i> , <i>imm8</i>	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) right <i>imm8</i> times.
REX.W + C1 /3 <i>ib</i>	RCR <i>r/m64</i> , <i>imm8</i>	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) right <i>imm8</i> times. Uses a 6 bit count.
D0 /0	ROL <i>r/m8</i> , 1	Valid	Valid	Rotate 8 bits <i>r/m8</i> left once.
REX + D0 /0	ROL <i>r/m8*</i> , 1	Valid	N.E.	Rotate 8 bits <i>r/m8</i> left once
D2 /0	ROL <i>r/m8</i> , CL	Valid	Valid	Rotate 8 bits <i>r/m8</i> left CL times.
REX + D2 /0	ROL <i>r/m8*</i> , CL	Valid	N.E.	Rotate 8 bits <i>r/m8</i> left CL times.
C0 /0 <i>ib</i>	ROL <i>r/m8</i> , <i>imm8</i>	Valid	Valid	Rotate 8 bits <i>r/m8</i> left <i>imm8</i> times.
REX + C0 /0 <i>ib</i>	ROL <i>r/m8*</i> , <i>imm8</i>	Valid	N.E.	Rotate 8 bits <i>r/m8</i> left <i>imm8</i> times.
D1 /0	ROL <i>r/m16</i> , 1	Valid	Valid	Rotate 16 bits <i>r/m16</i> left once.
D3 /0	ROL <i>r/m16</i> , CL	Valid	Valid	Rotate 16 bits <i>r/m16</i> left CL times.
C1 /0 <i>ib</i>	ROL <i>r/m16</i> , <i>imm8</i>	Valid	Valid	Rotate 16 bits <i>r/m16</i> left <i>imm8</i> times.
D1 /0	ROL <i>r/m32</i> , 1	Valid	Valid	Rotate 32 bits <i>r/m32</i> left once.
REX.W + D1 /0	ROL <i>r/m64</i> , 1	Valid	N.E.	Rotate 64 bits <i>r/m64</i> left once. Uses a 6 bit count.
D3 /0	ROL <i>r/m32</i> , CL	Valid	Valid	Rotate 32 bits <i>r/m32</i> left CL times.
REX.W + D3 /0	ROL <i>r/m64</i> , CL	Valid	N.E.	Rotate 64 bits <i>r/m64</i> left CL times. Uses a 6 bit count.
C1 /0 <i>ib</i>	ROL <i>r/m32</i> , <i>imm8</i>	Valid	Valid	Rotate 32 bits <i>r/m32</i> left <i>imm8</i> times.
C1 /0 <i>ib</i>	ROL <i>r/m64</i> , <i>imm8</i>	Valid	N.E.	Rotate 64 bits <i>r/m64</i> left <i>imm8</i> times. Uses a 6 bit count.
D0 /1	ROR <i>r/m8</i> , 1	Valid	Valid	Rotate 8 bits <i>r/m8</i> right once.
REX + D0 /1	ROR <i>r/m8*</i> , 1	Valid	N.E.	Rotate 8 bits <i>r/m8</i> right once.
D2 /1	ROR <i>r/m8</i> , CL	Valid	Valid	Rotate 8 bits <i>r/m8</i> right CL times.
REX + D2 /1	ROR <i>r/m8*</i> , CL	Valid	N.E.	Rotate 8 bits <i>r/m8</i> right CL times.
C0 /1 <i>ib</i>	ROR <i>r/m8</i> , <i>imm8</i>	Valid	Valid	Rotate 8 bits <i>r/m16</i> right <i>imm8</i> times.
REX + C0 /1 <i>ib</i>	ROR <i>r/m8*</i> , <i>imm8</i>	Valid	N.E.	Rotate 8 bits <i>r/m16</i> right <i>imm8</i> times.
D1 /1	ROR <i>r/m16</i> , 1	Valid	Valid	Rotate 16 bits <i>r/m16</i> right once.
D3 /1	ROR <i>r/m16</i> , CL	Valid	Valid	Rotate 16 bits <i>r/m16</i> right CL times.
C1 /1 <i>ib</i>	ROR <i>r/m16</i> , <i>imm8</i>	Valid	Valid	Rotate 16 bits <i>r/m16</i> right <i>imm8</i> times.
D1 /1	ROR <i>r/m32</i> , 1	Valid	Valid	Rotate 32 bits <i>r/m32</i> right once.

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
REX.W + D1 /1	ROR <i>r/m64</i> , 1	Valid	N.E.	Rotate 64 bits <i>r/m64</i> right once. Uses a 6 bit count.
D3 /1	ROR <i>r/m32</i> , CL	Valid	Valid	Rotate 32 bits <i>r/m32</i> right CL times.
REX.W + D3 /1	ROR <i>r/m64</i> , CL	Valid	N.E.	Rotate 64 bits <i>r/m64</i> right CL times. Uses a 6 bit count.
C1 /1 <i>ib</i>	ROR <i>r/m32</i> , <i>imm8</i>	Valid	Valid	Rotate 32 bits <i>r/m32</i> right <i>imm8</i> times.
REX.W + C1 /1 <i>ib</i>	ROR <i>r/m64</i> , <i>imm8</i>	Valid	N.E.	Rotate 64 bits <i>r/m64</i> right <i>imm8</i> times. Uses a 6 bit count.

**NOTES:**

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.

\*\* See IA-32 Architecture Compatibility section below.

**Description**

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. In legacy and compatibility mode, the processor restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except that a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

In 64-bit mode, using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Use of REX.W promotes the first operand to 64 bits and causes the count operand to become a 6-bit counter.

## INSTRUCTION SET REFERENCE, N-Z

### IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

### Operation

(\* RCL and RCR instructions \*)

SIZE  $\leftarrow$  OperandSize;

CASE (determine count) OF

    SIZE  $\leftarrow$  8: tempCOUNT  $\leftarrow$  (COUNT AND 1FH) MOD 9;

    SIZE  $\leftarrow$  16: tempCOUNT  $\leftarrow$  (COUNT AND 1FH) MOD 17;

    SIZE  $\leftarrow$  32: tempCOUNT  $\leftarrow$  COUNT AND 1FH;

    SIZE  $\leftarrow$  64: tempCOUNT  $\leftarrow$  COUNT AND 3FH;

ESAC;

(\* RCL instruction operation \*)

WHILE (tempCOUNT  $\neq$  0)

    DO

        tempCF  $\leftarrow$  MSB(DEST);

        DEST  $\leftarrow$  (DEST \* 2) + CF;

        CF  $\leftarrow$  tempCF;

        tempCOUNT  $\leftarrow$  tempCOUNT - 1;

    OD;

ELIHW;

IF COUNT = 1

    THEN OF  $\leftarrow$  MSB(DEST) XOR CF;

    ELSE OF is undefined;

FI;

(\* RCR instruction operation \*)

IF COUNT = 1

    THEN OF  $\leftarrow$  MSB(DEST) XOR CF;

    ELSE OF is undefined;

FI;

WHILE (tempCOUNT  $\neq$  0)

    DO

        tempCF  $\leftarrow$  LSB(SRC);

        DEST  $\leftarrow$  (DEST / 2) + (CF \* 2<sup>SIZE</sup>);

        CF  $\leftarrow$  tempCF;

        tempCOUNT  $\leftarrow$  tempCOUNT - 1;

    OD;

(\* ROL and ROR instructions \*)

SIZE  $\leftarrow$  OperandSize;

CASE (determine count) OF

```

SIZE ← 8:  tempCOUNT ← (COUNT AND 1FH) MOD 8; (* Mask count before MOD *)
SIZE ← 16: tempCOUNT ← (COUNT AND 1FH) MOD 16;
SIZE ← 32: tempCOUNT ← (COUNT AND 1FH) MOD 32;
SIZE ← 64: tempCOUNT ← (COUNT AND 1FH) MOD 64;
ESAC;

```

```

(* ROL instruction operation *)
IF (tempCOUNT > 0) (* Prevents updates to CF *)
  WHILE (tempCOUNT ≠ 0)
    DO
      tempCF ← MSB(DEST);
      DEST ← (DEST * 2) + tempCF;
      tempCOUNT ← tempCOUNT - 1;
    OD;
  ELIHW;
  CF ← LSB(DEST);
  IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
  FI;
FI;

```

```

(* ROR instruction operation *)
IF tempCOUNT > 0) (* Prevent updates to CF *)
  WHILE (tempCOUNT ≠ 0)
    DO
      tempCF ← LSB(SRC);
      DEST ← (DEST / 2) + (tempCF * 2SIZE);
      tempCOUNT ← tempCOUNT - 1;
    OD;
  ELIHW;
  CF ← MSB(DEST);
  IF COUNT = 1
    THEN OF ← MSB(DEST) XOR MSB - 1(DEST);
    ELSE OF is undefined;
  FI;
FI;

```

### Flags Affected

The CF flag contains the value of the bit shifted into it. The OF flag is affected only for single-bit rotates (see “Description” above); it is undefined for multi-bit rotates. The SF, ZF, AF, and PF flags are not affected.

## INSTRUCTION SET REFERENCE, N-Z

### Protected Mode Exceptions

#GP(0)	If the source operand is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the source operand is located in a nonwritable segment. If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 53 /r	RCPPS <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Computes the approximate reciprocals of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .

### Description

Performs an SIMD computation of the approximate reciprocals of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of an SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to  $|1.1111111110100000000000B * 2^{125}|$  are guaranteed to not produce tiny results; input values less than or equal to  $|1.00000000000110000000001B * 2^{126}|$  are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[31:0] ← APPROXIMATE(1.0/(SRC[31:0]));
DEST[63:32] ← APPROXIMATE(1.0/(SRC[63:32]));
DEST[95:64] ← APPROXIMATE(1.0/(SRC[95:64]));
DEST[127:96] ← APPROXIMATE(1.0/(SRC[127:96]));
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
RCCPS    __m128 _mm_rcp_ps(__m128 a)
```

## INSTRUCTION SET REFERENCE, N-Z

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

### Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

## RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 53 /r	RCPSS <i>xmm1</i> , <i>xmm2/m32</i>	Valid	Valid	Computes the approximate reciprocal of the scalar single-precision floating-point value in <i>xmm2/m32</i> and stores the result in <i>xmm1</i> .

### Description

Computes of an approximate reciprocal of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to  $|1.1111111110100000000000B * 2^{125}|$  are guaranteed to not produce tiny results; input values less than or equal to  $|1.00000000000110000000001B * 2^{126}|$  are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[31:0] ← APPROX (1.0/(SRC[31:0]));  
(\* DEST[127:32] unchanged \*)

### Intel C/C++ Compiler Intrinsic Equivalent

RCPSS            `__m128 _mm_rcp_ss(__m128 a)`

### SIMD Floating-Point Exceptions

None.

**Protected Mode Exceptions**

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	For unaligned memory reference.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## RDMSR—Read from Model Specific Register

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 32	RDMSR	Valid	Valid	Load MSR specified by ECX into EDX:EAX.
REX.W + 0F 32	RDMSR	Valid	N.E.	Load MSR specified by RCX into RDX:RAX.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Description

Loads the contents of a 64-bit model specific register (MSR) specified in an index register into registers EDX:EAX. The input value loaded into the index register is the address of the MSR to be read. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. If fewer than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined. In non-64-bit mode, the index register is specified in ECX. In 64-bit mode, the index register is specified in RCX and the higher 32-bits of RDX and RAX are cleared.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring, and machine check errors. Appendix B, “Model-Specific Registers (MSRs),” in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B*, lists all the MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The CPUID instruction should be used to determine whether MSRs are supported (EDX[5]=1) before using this instruction.

### IA-32 Architecture Compatibility

The MSRs and the ability to read them with the RDMSR instruction were introduced into the IA-32 Architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 21 of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

## INSTRUCTION SET REFERENCE, N-Z

### Operation

```
IF 64-Bit Mode and REX.W used
  THEN
    RAX[31:0] ← MSR(RCX)[31:0];
    RAX[63:32] ← 0;
    RDX[31:0] ← MSR(RCX)[63:32];
    RDX[63:32] ← 0;
  ELSE
    (* Non-64-bit modes, 64-bit mode default *)
    EDX-EAX ← MSR[ECX];
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.  
If the value in ECX specifies a reserved or unimplemented MSR address.

### Real-Address Mode Exceptions

#GP If the value in ECX specifies a reserved or unimplemented MSR address.

### Virtual-8086 Mode Exceptions

#GP(0) The RDMSR instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0.  
If the value in ECX or RCX specifies a reserved or unimplemented MSR address.



## RDPMC—Read Performance-Monitoring Counters

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 33	RDPMC	Valid	Valid	Read performance-monitoring counter specified by ECX into EDX:EAX.

### Description

Loads the 40-bit performance-monitoring counter specified in the ECX register into registers EDX:EAX. The EDX register is loaded with the high-order 8 bits of the counter and the EAX register is loaded with the low-order 32 bits. The counter to be read is specified with an unsigned integer placed in the ECX register.

The indices used to specify performance counters are model-specific and may vary by processor implementations. See Table 4-2 for valid indices for each processor family.

**Table 4-2. Valid Performance Counter Index Range for RDPMC**

Processor Family	CPUID Family/Model/ Other Signatures	Valid PMC Index Range	40-bit Counters
P6	Family 06H	0, 1	0, 1
Pentium 4, Intel Xeon processors	Family 0FH; Model 00H, 01H, 02H	$\geq 0$ and $\leq 17$	$\geq 0$ and $\leq 17$
Pentium 4, Intel Xeon processors	(Family 0FH; Model 03H, 04H, 06H) and (L3 is absent)	$\geq 0$ and $\leq 17$	$\geq 0$ and $\leq 17$
Pentium M processors	Family 06H, Model 09H, 0DH	0, 1	0, 1
64-bit Intel Xeon processors with L3 (see Chapter 18 of <i>IA-32 Intel® Architecture Software Developer's Manual, Volume 3B</i> )	(Family 0FH; Model 03H, 04H) and (L3 is present)	$\geq 0$ and $\leq 25$	$\geq 0$ and $\leq 17$
Intel® Core™ Solo and Intel® Core™ Duo processors	Family 06H, Model 0EH	0, 1	0, 1

The Pentium 4 and Intel Xeon processors also support “fast” (32-bit) and “slow” (40-bit) reads on the first 18 performance counters. Selected this option using ECX[bit 31]. If bit 31 is set, RDPMC reads only the low 32 bits of the selected performance counter. If bit 31 is clear, all 40 bits are read. A 32-bit result is returned in EAX and EDX is set to 0. A 32-bit read executes faster on Pentium 4 processors and Intel Xeon processors than a full 40-bit read.

On 64-bit Intel Xeon processors with L3, performance counters with indices 18-25 are 32-bit counters. EDX is cleared after executing RDPMC for these counters.

## INSTRUCTION SET REFERENCE, N-Z

When in protected or virtual 8086 mode, the performance-monitoring counters enabled (PCE) flag in register CR4 restricts the use of the RDPMC instruction as follows. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.)

The performance-monitoring counters can also be read with the RDMSR instruction, when executing at privilege level 0.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Appendix A, “Performance Monitoring Events,” in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B*, lists the events that can be counted for the Pentium 4, Intel Xeon, and earlier IA-32 processors.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPCM instruction.

In the Pentium 4 and Intel Xeon processors, performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

In 64-bit mode, RDPMC behavior is unchanged from 32-bit mode. The upper 32 bits of RAX and RDX are cleared.

### Operation

(\* P6 family processors and Pentium processor with MMX technology \*)

IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))

THEN

EAX ← PMC(ECX)[31:0];

EDX ← PMC(ECX)[39:32];

ELSE (\* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 \*)

#GP(0);

FI;

(\* Processors with CPUID family 15 \*)

IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))

THEN IF (ECX[30:0] = 0:17)

```

THEN IF ECX[31] = 0
    THEN IF 64-Bit Mode
        THEN
            RAX[31:0] ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
            RAX[63:32] ← 0;
            RDX[31:0] ← PMC(ECX[30:0])[39:32];
            RDX[63:32] ← 0;
        ELSE
            EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
            EDX ← PMC(ECX[30:0])[39:32];
        FI;
    ELSE IF ECX[31] = 1
        THEN IF 64-Bit Mode
            THEN
                RAX[31:0] ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
                RAX[63:32] ← 0;
                RDX ← 0;
            ELSE
                EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
                EDX ← 0;
            FI;
        FI;
    ELSE IF (*64-bit Intel Xeon processor with L3 *)
        THEN IF (ECX[30:0] = 18:25
            EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
            EDX ← 0;
        FI;
    ELSE (* Invalid PMC index in ECX[30:0], see Table 4-4. *)
        GP(0);
    FI;
ELSE (* CR4.PCE = 0 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;

```

## INSTRUCTION SET REFERENCE, N-Z

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.  
If an invalid performance counter index is specified (see Table 4-2).  
(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.

### Real-Address Mode Exceptions

#GP If an invalid performance counter index is specified (see Table 4-2).  
(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.

### Virtual-8086 Mode Exceptions

#GP(0) If the PCE flag in the CR4 register is clear.  
If an invalid performance counter index is specified (see Table 4-2).  
(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.  
If an invalid performance counter index is specified in ECX[30:0] (see Table 4-2).

## RDTSC—Read Time-Stamp Counter

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 31	RDTSC	Valid	Valid	Read time-stamp counter into EDX:EAX.

### Description

In legacy, compatibility and default 64-bit mode; loads the current value of the processor's time-stamp counter into the EDX:EAX registers. The time-stamp counter is contained in a 64-bit MSR. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register.

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See “Time Stamp Counter” in Chapter 18 of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

When in protected or virtual 8086 mode, the time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the TSD flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDTSC instruction is always enabled.)

The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed.

This instruction was introduced by the Pentium processor.

In 64-bit mode, RDTSC behavior is unchanged from 32-bit mode. The upper 32 bits of RAX and RDX are cleared.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 21 of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

## INSTRUCTION SET REFERENCE, N-Z

### Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
  THEN
    IF 64-Bit Mode
      THEN
        RAX[31:0] ← TimeStampCounter[31:0];
        RAX[63:32] ← 0;
        RDX[31:0] ← TimeStampCounter[63:32];
        RDX[63:32] ← 0;
      ELSE
        EDX:EAX ← TimeStampCounter;
    FI;
  ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)                    If the TSD flag in register CR4 is set and the CPL is greater than 0.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

#GP(0)                    If the TSD flag in register CR4 is set.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

Same exceptions as in Protected Mode.

## REP/REPE/REPZ/REPNE/REP NZ—Repeat String Operation Prefix

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 6C	REP INS <i>m8</i> , DX	Valid	Valid	Input (E)CX bytes from port DX into ES:[(E)DI].
F3 6C	REP INS <i>m8</i> , DX	Valid	N.E.	Input RCX bytes from port DX into [RDI].
F3 6D	REP INS <i>m16</i> , DX	Valid	Valid	Input (E)CX words from port DX into ES:[(E)DI].
F3 6D	REP INS <i>m32</i> , DX	Valid	Valid	Input (E)CX doublewords from port DX into ES:[(E)DI].
F3 6D	REP INS <i>r/m32</i> , DX	Valid	N.E.	Input RCX default size from port DX into [RDI].
F3 A4	REP MOVS <i>m8</i> , <i>m8</i>	Valid	Valid	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI].
REX.W + F3 A4	REP MOVS <i>m8</i> , <i>m8</i>	Valid	N.E.	Move RCX bytes from [RSI] to [RDI].
F3 A5	REP MOVS <i>m16</i> , <i>m16</i>	Valid	Valid	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI].
F3 A5	REP MOVS <i>m32</i> , <i>m32</i>	Valid	Valid	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].
REX.W + F3 A5	REP MOVS <i>m64</i> , <i>m64</i>	Valid	N.E.	Move RCX quadwords from [RSI] to [RDI].
F3 6E	REP OUTS DX, <i>r/m8</i>	Valid	Valid	Output (E)CX bytes from DS:[(E)SI] to port DX.
REX.W + F3 6E	REP OUTS DX, <i>r/m8</i> *	Valid	N.E.	Output RCX bytes from [RSI] to port DX.
F3 6F	REP OUTS DX, <i>r/m16</i>	Valid	Valid	Output (E)CX words from DS:[(E)SI] to port DX.
F3 6F	REP OUTS DX, <i>r/m32</i>	Valid	Valid	Output (E)CX doublewords from DS:[(E)SI] to port DX.
REX.W + F3 6F	REP OUTS DX, <i>r/m32</i>	Valid	N.E.	Output RCX default size from [RSI] to port DX.
F3 AC	REP LODS AL	Valid	Valid	Load (E)CX bytes from DS:[(E)SI] to AL.
REX.W + F3 AC	REP LODS AL	Valid	N.E.	Load RCX bytes from [RSI] to AL.
F3 AD	REP LODS AX	Valid	Valid	Load (E)CX words from DS:[(E)SI] to AX.
F3 AD	REP LODS EAX	Valid	Valid	Load (E)CX doublewords from DS:[(E)SI] to EAX.
REX.W + F3 AD	REP LODS RAX	Valid	N.E.	Load RCX quadwords from [RSI] to RAX.

## INSTRUCTION SET REFERENCE, N-Z

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 AA	REP STOS <i>m8</i>	Valid	Valid	Fill (E)CX bytes at ES:[(E)DI] with AL.
REX.W + F3 AA	REP STOS <i>m8</i>	Valid	N.E.	Fill RCX bytes at [RDI] with AL.
F3 AB	REP STOS <i>m16</i>	Valid	Valid	Fill (E)CX words at ES:[(E)DI] with AX.
F3 AB	REP STOS <i>m32</i>	Valid	Valid	Fill (E)CX doublewords at ES:[(E)DI] with EAX.
REX.W + F3 AB	REP STOS <i>m64</i>	Valid	N.E.	Fill RCX quadwords at [RDI] with RAX.
F3 A6	REPE CMPS <i>m8, m8</i>	Valid	Valid	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].
REX.W + F3 A6	REPE CMPS <i>m8, m8</i>	Valid	N.E.	Find non-matching bytes in [RDI] and [RSI].
F3 A7	REPE CMPS <i>m16, m16</i>	Valid	Valid	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI].
F3 A7	REPE CMPS <i>m32, m32</i>	Valid	Valid	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI].
REX.W + F3 A7	REPE CMPS <i>m64, m64</i>	Valid	N.E.	Find non-matching quadwords in [RDI] and [RSI].
F3 AE	REPE SCAS <i>m8</i>	Valid	Valid	Find non-AL byte starting at ES:[(E)DI].
REX.W + F3 AE	REPE SCAS <i>m8</i>	Valid	N.E.	Find non-AL byte starting at [RDI].
F3 AF	REPE SCAS <i>m16</i>	Valid	Valid	Find non-AX word starting at ES:[(E)DI].
F3 AF	REPE SCAS <i>m32</i>	Valid	Valid	Find non-EAX doubleword starting at ES:[(E)DI].
REX.W + F3 AF	REPE SCAS <i>m64</i>	Valid	N.E.	Find non-RAX quadword starting at [RDI].
F2 A6	REPNE CMPS <i>m8, m8</i>	Valid	Valid	Find matching bytes in ES:[(E)DI] and DS:[(E)SI].
REX.W + F2 A6	REPNE CMPS <i>m8, m8</i>	Valid	N.E.	Find matching bytes in [RDI] and [RSI].
F2 A7	REPNE CMPS <i>m16, m16</i>	Valid	Valid	Find matching words in ES:[(E)DI] and DS:[(E)SI].
F2 A7	REPNE CMPS <i>m32, m32</i>	Valid	Valid	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI].
REX.W + F2 A7	REPNE CMPS <i>m64, m64</i>	Valid	N.E.	Find matching doublewords in [RDI] and [RSI].



Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F2 AE	REPNE SCAS <i>m8</i>	Valid	Valid	Find AL, starting at ES:[(E)DI].
REX.W + F2 AE	REPNE SCAS <i>m8</i>	Valid	N.E.	Find AL, starting at [RDI].
F2 AF	REPNE SCAS <i>m16</i>	Valid	Valid	Find AX, starting at ES:[(E)DI].
F2 AF	REPNE SCAS <i>m32</i>	Valid	Valid	Find EAX, starting at ES:[(E)DI].
REX.W + F2 AF	REPNE SCAS <i>m64</i>	Valid	N.E.	Find RAX, starting at [RDI].

**NOTES:**

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.

**Description**

Repeats a string instruction the number of times specified in the count register or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP prefix is undefined when used with non-string instructions.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. All of these repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0. See Table 4-3.

**Table 4-3. Repeat Prefixes**

Repeat Prefix	Termination Condition 1*	Termination Condition 2
REP	RCX or (E)CX = 0	None
REPE/REPZ	RCX or (E)CX = 0	ZF = 0
REPNE/REPNZ	RCX or (E)CX = 0	ZF = 1

**NOTES:**

\* Count register is CX, ECX or RCX by default, depending on attributes of the operating modes. In 64-bit mode, if default operation size is 32 bits, the count register becomes RCX when a REX.W prefix is used.

The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

## INSTRUCTION SET REFERENCE, N-Z

When the REPE/REPZ and REPNE/REPZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute. Note that a REP STOS instruction is the fastest way to initialize a large block of memory.

In 64-bit mode, default operation size is 32 bits. The default count register is RCX for REP INS and REP OUTS; it is ECX for other instructions. REX.W does not promote operation to 64-bit for REP INS and REP OUTS. However, using an REX prefix in the form of REX.W does promote operation to 64-bit operands for other REP/REPNE/REPZ/REPZ instructions. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```
IF AddressSize = 16
  THEN
    Use CX for CountReg;
  ELSE IF AddressSize = 64 and REX.W used
    THEN Use RCX for CountReg; FI;
  ELSE
    Use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
  DO
    Service pending interrupts (if any);
    Execute associated string instruction;
    CountReg ← (CountReg – 1);
    IF CountReg = 0
      THEN exit WHILE loop; FI;
    IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
      or (Repeat prefix is REPZ or REPNE) and (ZF = 1)
      THEN exit WHILE loop; FI;
  OD;
```

### Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

### Exceptions (All Operating Modes)

None; however, exceptions can be generated by the instruction a repeat prefix is associated with.

### 64-Bit Mode Exceptions

#GP(0)                    If the memory address is in a non-canonical form.

## RET—Return from Procedure

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
C3	RET	Valid	Valid	Near return to calling procedure.
CB	RET	Valid	Valid	Far return to calling procedure.
C2 <i>iw</i>	RET <i>imm16</i>	Valid	Valid	Near return to calling procedure and pop <i>imm16</i> bytes from stack.
CA <i>iw</i>	RET <i>imm16</i>	Valid	Valid	Far return to calling procedure and pop <i>imm16</i> bytes from stack.

### Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- Near return—A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- Far return—A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- Inter-privilege-level far return—A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled “Calling Procedures Using Call and RET” in Chapter 6 of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure's stack and the calling procedure's stack (that is, the stack being returned to).

In 64-bit mode, the default operation size of this instruction is the stack size, i.e. 64 bits.

## Operation

(\* Near return \*)

IF instruction = Near return

THEN;

IF OperandSize = 32

THEN

IF top 4 bytes of stack not within stack limits

THEN #SS(0); FI;

EIP ← Pop();

ELSE

IF OperandSize = 64

THEN

IF top 8 bytes of stack not within stack limits

THEN #SS(0); FI;

RIP ← Pop();

ELSE (\* OperandSize = 16 \*)

IF top 2 bytes of stack not within stack limits

THEN #SS(0); FI;

tempEIP ← Pop();

tempEIP ← tempEIP AND 0000FFFFH;

IF tempEIP not within code segment limits

THEN #GP(0); FI;

EIP ← tempEIP;

FI;

FI;

IF instruction has immediate operand

THEN IF StackAddressSize = 32

THEN

ESP ← ESP + SRC; (\* Release parameters from stack \*)

ELSE

IF StackAddressSize = 64

THEN

## INSTRUCTION SET REFERENCE, N-Z

```

        RSP ← RSP + SRC; (* Release parameters from stack *)
    ELSE (* StackAddressSize = 16 *)
        SP ← SP + SRC; (* Release parameters from stack *)
    FI;
FI;
FI;
FI;
FI;
(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) or (PE = 1 AND VM = 1)) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF top 12 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            ELSE (* OperandSize = 16 *)
                IF top 6 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                tempEIP ← Pop();
                tempEIP ← tempEIP AND 0000FFFFH;
                IF tempEIP not within code segment limits
                    THEN #GP(0); FI;
                EIP ← tempEIP;
                CS ← Pop(); (* 16-bit pop *)
            FI;
        IF instruction has immediate operand
            THEN
                SP ← SP + (SRC AND FFFFH); (* Release parameters from stack *)
        FI;
FI;
(* Protected mode, not virtual-8086 mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 0) and instruction = far RET
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
                ELSE (* OperandSize = 16 *)
                    IF second word on stack is not within stack limits
                        THEN #SS(0); FI;
            FI;
        FI;
    IF return code segment selector is NULL
        THEN #GP(0); FI;
    IF return code segment selector addresses descriptor beyond descriptor table limit
```

```

    THEN #GP(selector); FI;
    Obtain descriptor to which return code segment selector points from descriptor table;
    IF return code segment descriptor is not a code segment
        THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is non-conforming and return code
    segment DPL ≠ return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is not present
        THEN #NP(selector); FI;
    IF return code segment selector RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
    FI;
FI;

```

## RETURN-SAME-PRIVILEGE-LEVEL:

```

    IF the return instruction pointer is not within ther return code segment limit
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            ESP ← ESP + SRC; (* Release parameters from stack *)
        ELSE (* OperandSize = 16 *)
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop *)
            ESP ← ESP + SRC; (* Release parameters from stack *)
    FI;

```

## RETURN-OUTER-PRIVILEGE-LEVEL:

```

    IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
    or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
        THEN #SS(0); FI;
    Read return segment selector;
    IF stack segment selector is NULL
        THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
        THEN #GP(selector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL ≠ RPL of the return code segment selector
    or stack segment is not a writable data segment
    or stack segment descriptor DPL ≠ RPL of the return code segment selector

```

## INSTRUCTION SET REFERENCE, N-Z

```
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor
            information also loaded *)
        CS(RPL) ← CPL;
        ESP ← ESP + SRC; (* Release parameters from called procedure's stack *)
        tempESP ← Pop();
        tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; segment
            descriptor information also loaded *)
        ESP ← tempESP;
        SS ← tempSS;
    ELSE (* OperandSize = 16 *)
        EIP ← Pop();
        EIP ← EIP AND 0000FFFFH;
        CS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
        CS(RPL) ← CPL;
        ESP ← ESP + SRC; (* Release parameters from called procedure's stack *)
        tempESP ← Pop();
        tempSS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
        ESP ← tempESP;
        SS ← tempSS;
FI;

FOR each of segment register (ES, FS, GS, and DS)
    DO
        IF segment register points to data or non-conforming code segment
            and CPL > segment descriptor DPL (* DPL in hidden part of segment register *)
            THEN SegmentSelector ← 0; (* Segment selector invalid *)
        FI;
    OD;

For each of ES, FS, GS, and DS
    DO
        IF segment selector index is not within descriptor table limits
            or segment descriptor indicates the segment is not a data or
            readable code segment
            or if the segment is a data or non-conforming code segment
            and the segment descriptor's DPL < CPL or RPL of code segment's
            segment selector
            THEN SegmentSelector ← 0; (* Segment selector invalid *)
    OD;
```



ESP ← ESP + SRC; (\* Release parameters from calling procedure's stack \*)

(\* IA-32e Mode \*)

```

IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far RET
  THEN
    IF OperandSize = 32
      THEN
        IF second doubleword on stack is not within stack limits
          THEN #SS(0); FI;
        IF first or second doubleword on stack is not in canonical space
          THEN #SS(0); FI;
      ELSE
        IF OperandSize = 16
          THEN
            IF second word on stack is not within stack limits
              THEN #SS(0); FI;
            IF first or second word on stack is not in canonical space
              THEN #SS(0); FI;
          ELSE (* OperandSize = 64 *)
            IF first or second quadword on stack is not in canonical space
              THEN #SS(0); FI;
        FI
      FI;
    IF return code segment selector is NULL
      THEN GP(0); FI;
    IF return code segment selector addresses descriptor beyond descriptor table limit
      THEN GP(selector); FI;
    IF return code segment selector addresses descriptor in non-canonical space
      THEN GP(selector); FI;
    Obtain descriptor to which return code segment selector points from descriptor table;
    IF return code segment descriptor is not a code segment
      THEN #GP(selector); FI;
    IF return code segment descriptor has L-bit = 1 and D-bit = 1
      THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
      THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
      THEN #GP(selector); FI;
    IF return code segment descriptor is non-conforming
    and return code segment DPL ≠ return code segment selector RPL
      THEN #GP(selector); FI;
    IF return code segment descriptor is not present
      THEN #NP(selector); FI;
    IF return code segment selector RPL > CPL
      THEN GOTO IA-32E-MODE-RETURN-OUTER-PRIVILEGE-LEVEL;
      ELSE GOTO IA-32E-MODE-RETURN-SAME-PRIVILEGE-LEVEL;
  
```

## INSTRUCTION SET REFERENCE, N-Z

FI;  
FI;

### IA-32E-MODE-RETURN-SAME-PRIVILEGE-LEVEL:

IF the return instruction pointer is not within the return code segment limit

THEN #GP(0); FI;

IF the return instruction pointer is not within canonical address space

THEN #GP(0); FI;

IF OperandSize = 32

THEN

EIP ← Pop();

CS ← Pop(); (\* 32-bit pop, high-order 16 bits discarded \*)

ESP ← ESP + SRC; (\* Release parameters from stack \*)

ELSE

IF OperandSize = 16

THEN

EIP ← Pop();

EIP ← EIP AND 0000FFFFH;

CS ← Pop(); (\* 16-bit pop \*)

ESP ← ESP + SRC; (\* Release parameters from stack \*)

ELSE (\* OperandSize = 64 \*)

RIP ← Pop();

CS ← Pop(); (\* 64-bit pop, high-order 48 bits discarded \*)

ESP ← ESP + SRC; (\* Release parameters from stack \*)

FI;

FI;

### IA-32E-MODE-RETURN-OUTER-PRIVILEGE-LEVEL:

IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)

or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)

THEN #SS(0); FI;

IF top (16 + SRC) bytes of stack are not in canonical address space (OperandSize = 32)

or top (8 + SRC) bytes of stack are not in canonical address space (OperandSize = 16)

or top (32 + SRC) bytes of stack are not in canonical address space (OperandSize = 64)

THEN #SS(0); FI;

Read return stack segment selector;

IF stack segment selector is NULL

THEN

IF new CS descriptor L-bit = 0

THEN #GP(selector);

IF stack segment selector RPL = 3

THEN #GP(selector);

FI;

IF return stack segment descriptor is not within descriptor table limits

THEN #GP(selector); FI;

IF return stack segment descriptor is in non-canonical address space

THEN #GP(selector); FI;

```

Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor
        information also loaded *)
        CS(RPL) ← CPL;
        ESP ← ESP + SRC; (* Release parameters from called procedure's stack *)
        tempESP ← Pop();
        tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor
        information also loaded *)
        ESP ← tempESP;
        SS ← tempSS;
    ELSE
        IF OperandSize = 16
            THEN
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                ESP ← ESP + SRC; (* release parameters from called
                procedure's stack *)
                tempESP ← Pop();
                tempSS ← Pop(); (* 16-bit pop; segment descriptor information loaded *)
                ESP ← tempESP;
                SS ← tempSS;
            ELSE (* OperandSize = 64 *)
                RIP ← Pop();
                CS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; segment
                descriptor information loaded *)
                CS(RPL) ← CPL;
                ESP ← ESP + SRC; (* Release parameters from called procedure's
                stack *)
                tempESP ← Pop();
                tempSS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; segment
                descriptor information also loaded *)
                ESP ← tempESP;

```

## INSTRUCTION SET REFERENCE, N-Z

```
                SS ← tempSS;
FI;
FI;

FOR each of segment register (ES, FS, GS, and DS)
DO
    IF segment register points to data or non-conforming code segment
    and CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
    THEN SegmentSelector ← 0; (* SegmentSelector invalid *)
    FI;
OD;

For each of ES, FS, GS, and DS
DO
    IF segment selector index is not within descriptor table limits
    or segment descriptor indicates the segment is not a data or readable code segment
    or if the segment is a data or non-conforming code segment
    and the segment descriptor's DPL < CPL or RPL of code segment's segment selector
    THEN SegmentSelector ← 0; (* SegmentSelector invalid *)
OD;

ESP ESP + SRC; (* Release parameters from calling procedure's stack *)
```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector NULL. If the return instruction pointer is not within the return code segment limit
#GP(selector)	If the RPL of the return code segment selector is less than the CPL. If the return code or stack segment selector index is not within its descriptor table limits. If the return code segment descriptor does not indicate a code segment. If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector If the stack segment is not a writable data segment. If the stack segment selector RPL is not equal to the RPL of the return code segment selector.

	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
#SS(0)	If the top bytes of stack are not within stack limits. If the return stack segment is not present.
#NP(selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

### Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit
#SS	If the top bytes of stack are not within stack limits.

### Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit
#SS(0)	If the top bytes of stack are not within stack limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when alignment checking is enabled.

### Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

### 64-Bit Mode Exceptions

#GP(0)	If the return instruction pointer is non-canonical. If the return instruction pointer is not within the return code segment limit. If the stack segment selector is NULL going back to compatibility mode. If the stack segment selector is NULL going back to CPL3 64-bit mode. If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode. If the return code segment selector is NULL.
#GP(selector)	If the proposed segment descriptor for a code segment does not indicate it is a code segment. If the proposed new code segment descriptor has both the D-bit and L-bit set.

## INSTRUCTION SET REFERENCE, N-Z

If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.

If CPL is greater than the RPL of the code segment selector.

If the DPL of a conforming-code segment is greater than the return code segment selector RPL.

If a segment selector index is outside its descriptor table limits.

If a segment descriptor memory address is non-canonical.

If the stack segment is not a writable data segment.

If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.

If the stack segment selector RPL is not equal to the RPL of the return code segment selector.

#SS(0) If an attempt to pop a value off the stack violates the SS limit.

If an attempt to pop a value off the stack causes a non-canonical address to be referenced.

#NP(selector) If the return code or stack segment is not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## RSM—Resume from System Management Mode

Opcode	Instruction	Non-SMM Mode	SMM Mode	Description
0F AA	RSM	Invalid	Valid	Resume operation of interrupted program.

### Description

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SMM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium and Intel486 processors only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

The SMM state map used by RSM supports resuming processor context for non-64-bit modes and 64-bit mode.

See Chapter 24, “System Management,” in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3B*, for more information about SMM and the behavior of the RSM instruction.

### Operation

```
ReturnFromSMM;
IF (IA-32e mode supported)
  THEN
    ProcessorState ← Restore(SMMDump(IA-32e SMM STATE MAP));
  Else
    ProcessorState ← Restore(SMMDump(Non-32-Bit-Mode SMM STATE MAP));
FI
```

### Flags Affected

All.

### Protected Mode Exceptions

#UD If an attempt is made to execute this instruction when the processor is not in SMM.

## **INSTRUCTION SET REFERENCE, N-Z**

### **Real-Address Mode Exceptions**

Same exceptions as in Protected Mode.

### **Virtual-8086 Mode Exceptions**

Same exceptions as in Protected Mode.

### **Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

### **64-Bit Mode Exceptions**

Same exceptions as in Protected Mode.



## RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 52 /r	RSQRTPS <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Computes the approximate reciprocals of the square roots of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .

### Description

Performs an SIMD computation of the approximate reciprocals of the square roots of the four packed single-precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of an SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than -0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[31:0] ← APPROXIMATE(1.0/SQRT(SRC[31:0]));
DEST[63:32] ← APPROXIMATE(1.0/SQRT(SRC[63:32]));
DEST[95:64] ← APPROXIMATE(1.0/SQRT(SRC[95:64]));
DEST[127:96] ← APPROXIMATE(1.0/SQRT(SRC[127:96]));
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
RSQRTPS    __m128 _mm_rsqrtps(__m128 a)
```

### SIMD Floating-Point Exceptions

None.

## INSTRUCTION SET REFERENCE, N-Z

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE[bit 25] = 0.

### Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE[bit 25] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

## RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 52 /r	RSQRTSS <i>xmm1</i> , <i>xmm2/m32</i>	Valid	Valid	Computes the approximate reciprocal of the square root of the low single-precision floating-point value in <i>xmm2/m32</i> and stores the results in <i>xmm1</i> .

### Description

Computes an approximate reciprocal of the square root of the low single-precision floating-point value in the source operand (second operand) stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order double-words of the destination operand remain unchanged. See Figure 10-6 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than  $-0.0$ ), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[31:0] ← APPROXIMATE(1.0/SQRT(SRC[31:0]));  
(\* DEST[127:32] unchanged \*)

### Intel C/C++ Compiler Intrinsic Equivalent

RSQRTSS     \_\_m128 \_mm\_rsqr\_ss(\_\_m128 a)

### SIMD Floating-Point Exceptions

None.

**Protected Mode Exceptions**

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SAHF—Store AH into Flags

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9E	SAHF	Invalid*	Valid	Loads SF, ZF, AF, PF, and CF from AH into EFLAGS register.

\* Valid in specific steppings. See Description section.

### Description

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). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS register remain as shown in the “Operation” section below.

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

### Operation

```
IF IA-64 Mode
  THEN
    IF CPUID.80000001.ECX[0] = 1;
      THEN
        RFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
      ELSE
        #UD;
    FI
  ELSE
    EFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
FI;
```

### Flags Affected

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are unaffected, with the values remaining 1, 0, and 0, respectively.

### Protected Mode Exceptions

None.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

None.

## INSTRUCTION SET REFERENCE, N-Z

### Compatibility Mode Exceptions

None.

### 64-Bit Mode Exceptions

#UD                    If CPUID.80000001.ECX[0] = 0.



## SAL/SAR/SHL/SHR—Shift

Opcode***	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D0 /4	SAL <i>r/m8</i> , 1	Valid	Valid	Multiply <i>r/m8</i> by 2, once.
REX + D0 /4	SAL <i>r/m8**</i> , 1	Valid	N.E.	Multiply <i>r/m8</i> by 2, once.
D2 /4	SAL <i>r/m8</i> , CL	Valid	Valid	Multiply <i>r/m8</i> by 2, CL times.
REX + D2 /4	SAL <i>r/m8**</i> , CL	Valid	N.E.	Multiply <i>r/m8</i> by 2, CL times.
C0 /4 <i>ib</i>	SAL <i>r/m8</i> , <i>imm8</i>	Valid	Valid	Multiply <i>r/m8</i> by 2, <i>imm8</i> times.
REX + C0 /4 <i>ib</i>	SAL <i>r/m8**</i> , <i>imm8</i>	Valid	N.E.	Multiply <i>r/m8</i> by 2, <i>imm8</i> times.
D1 /4	SAL <i>r/m16</i> , 1	Valid	Valid	Multiply <i>r/m16</i> by 2, once.
D3 /4	SAL <i>r/m16</i> , CL	Valid	Valid	Multiply <i>r/m16</i> by 2, CL times.
C1 /4 <i>ib</i>	SAL <i>r/m16</i> , <i>imm8</i>	Valid	Valid	Multiply <i>r/m16</i> by 2, <i>imm8</i> times.
D1 /4	SAL <i>r/m32</i> , 1	Valid	Valid	Multiply <i>r/m32</i> by 2, once.
REX.W + D1 /4	SAL <i>r/m64</i> , 1	Valid	N.E.	Multiply <i>r/m64</i> by 2, once.
D3 /4	SAL <i>r/m32</i> , CL	Valid	Valid	Multiply <i>r/m32</i> by 2, CL times.
REX.W + D3 /4	SAL <i>r/m64</i> , CL	Valid	N.E.	Multiply <i>r/m64</i> by 2, CL times.
C1 /4 <i>ib</i>	SAL <i>r/m32</i> , <i>imm8</i>	Valid	Valid	Multiply <i>r/m32</i> by 2, <i>imm8</i> times.
REX.W + C1 /4 <i>ib</i>	SAL <i>r/m64</i> , <i>imm8</i>	Valid	N.E.	Multiply <i>r/m64</i> by 2, <i>imm8</i> times.
D0 /7	SAR <i>r/m8</i> , 1	Valid	Valid	Signed divide* <i>r/m8</i> by 2, once.
REX + D0 /7	SAR <i>r/m8**</i> , 1	Valid	N.E.	Signed divide* <i>r/m8</i> by 2, once.
D2 /7	SAR <i>r/m8</i> , CL	Valid	Valid	Signed divide* <i>r/m8</i> by 2, CL times.
REX + D2 /7	SAR <i>r/m8**</i> , CL	Valid	N.E.	Signed divide* <i>r/m8</i> by 2, CL times.
C0 /7 <i>ib</i>	SAR <i>r/m8</i> , <i>imm8</i>	Valid	Valid	Signed divide* <i>r/m8</i> by 2, <i>imm8</i> time.
REX + C0 /7 <i>ib</i>	SAR <i>r/m8**</i> , <i>imm8</i>	Valid	N.E.	Signed divide* <i>r/m8</i> by 2, <i>imm8</i> times.
D1 /7	SAR <i>r/m16</i> , 1	Valid	Valid	Signed divide* <i>r/m16</i> by 2, once.
D3 /7	SAR <i>r/m16</i> , CL	Valid	Valid	Signed divide* <i>r/m16</i> by 2, CL times.
C1 /7 <i>ib</i>	SAR <i>r/m16</i> , <i>imm8</i>	Valid	Valid	Signed divide* <i>r/m16</i> by 2, <i>imm8</i> times.
D1 /7	SAR <i>r/m32</i> , 1	Valid	Valid	Signed divide* <i>r/m32</i> by 2, once.
REX.W + D1 /7	SAR <i>r/m64</i> , 1	Valid	N.E.	Signed divide* <i>r/m64</i> by 2, once.

## INSTRUCTION SET REFERENCE, N-Z

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D3 /7	SAR <i>r/m32</i> , CL	Valid	Valid	Signed divide* <i>r/m32</i> by 2, CL times.
REX.W + D3 /7	SAR <i>r/m64</i> , CL	Valid	N.E.	Signed divide* <i>r/m64</i> by 2, CL times.
C1 /7 <i>ib</i>	SAR <i>r/m32</i> , <i>imm8</i>	Valid	Valid	Signed divide* <i>r/m32</i> by 2, <i>imm8</i> times.
REX.W + C1 /7 <i>ib</i>	SAR <i>r/m64</i> , <i>imm8</i>	Valid	N.E.	Signed divide* <i>r/m64</i> by 2, <i>imm8</i> times
D0 /4	SHL <i>r/m8</i> , 1	Valid	Valid	Multiply <i>r/m8</i> by 2, once.
REX + D0 /4	SHL <i>r/m8**</i> , 1	Valid	N.E.	Multiply <i>r/m8</i> by 2, once.
D2 /4	SHL <i>r/m8</i> , CL	Valid	Valid	Multiply <i>r/m8</i> by 2, CL times.
REX + D2 /4	SHL <i>r/m8**</i> , CL	Valid	N.E.	Multiply <i>r/m8</i> by 2, CL times.
C0 /4 <i>ib</i>	SHL <i>r/m8</i> , <i>imm8</i>	Valid	Valid	Multiply <i>r/m8</i> by 2, <i>imm8</i> times.
REX + C0 /4 <i>ib</i>	SHL <i>r/m8**</i> , <i>imm8</i>	Valid	N.E.	Multiply <i>r/m8</i> by 2, <i>imm8</i> times.
D1 /4	SHL <i>r/m16</i> ,1	Valid	Valid	Multiply <i>r/m16</i> by 2, once.
D3 /4	SHL <i>r/m16</i> , CL	Valid	Valid	Multiply <i>r/m16</i> by 2, CL times.
C1 /4 <i>ib</i>	SHL <i>r/m16</i> , <i>imm8</i>	Valid	Valid	Multiply <i>r/m16</i> by 2, <i>imm8</i> times.
D1 /4	SHL <i>r/m32</i> ,1	Valid	Valid	Multiply <i>r/m32</i> by 2, once.
REX.W + D1 /4	SHL <i>r/m64</i> ,1	Valid	N.E.	Multiply <i>r/m64</i> by 2, once.
D3 /4	SHL <i>r/m32</i> , CL	Valid	Valid	Multiply <i>r/m32</i> by 2, CL times.
REX.W + D3 /4	SHL <i>r/m64</i> , CL	Valid	N.E.	Multiply <i>r/m64</i> by 2, CL times.
C1 /4 <i>ib</i>	SHL <i>r/m32</i> , <i>imm8</i>	Valid	Valid	Multiply <i>r/m32</i> by 2, <i>imm8</i> times.
REX.W + C1 /4 <i>ib</i>	SHL <i>r/m64</i> , <i>imm8</i>	Valid	N.E.	Multiply <i>r/m64</i> by 2, <i>imm8</i> times.
D0 /5	SHR <i>r/m8</i> ,1	Valid	Valid	Unsigned divide <i>r/m8</i> by 2, once.
REX + D0 /5	SHR <i>r/m8**</i> , 1	Valid	N.E.	Unsigned divide <i>r/m8</i> by 2, once.
D2 /5	SHR <i>r/m8</i> , CL	Valid	Valid	Unsigned divide <i>r/m8</i> by 2, CL times.
REX + D2 /5	SHR <i>r/m8**</i> , CL	Valid	N.E.	Unsigned divide <i>r/m8</i> by 2, CL times.
C0 /5 <i>ib</i>	SHR <i>r/m8</i> , <i>imm8</i>	Valid	Valid	Unsigned divide <i>r/m8</i> by 2, <i>imm8</i> times.
REX + C0 /5 <i>ib</i>	SHR <i>r/m8**</i> , <i>imm8</i>	Valid	N.E.	Unsigned divide <i>r/m8</i> by 2, <i>imm8</i> times.
D1 /5	SHR <i>r/m16</i> , 1	Valid	Valid	Unsigned divide <i>r/m16</i> by 2, once.
D3 /5	SHR <i>r/m16</i> , CL	Valid	Valid	Unsigned divide <i>r/m16</i> by 2, CL times
C1 /5 <i>ib</i>	SHR <i>r/m16</i> , <i>imm8</i>	Valid	Valid	Unsigned divide <i>r/m16</i> by 2, <i>imm8</i> times.

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D1 /5	SHR <i>r/m32</i> , 1	Valid	Valid	Unsigned divide <i>r/m32</i> by 2, once.
REX.W + D1 /5	SHR <i>r/m64</i> , 1	Valid	N.E.	Unsigned divide <i>r/m64</i> by 2, once.
D3 /5	SHR <i>r/m32</i> , CL	Valid	Valid	Unsigned divide <i>r/m32</i> by 2, CL times.
REX.W + D3 /5	SHR <i>r/m64</i> , CL	Valid	N.E.	Unsigned divide <i>r/m64</i> by 2, CL times.
C1 /5 <i>ib</i>	SHR <i>r/m32</i> , <i>imm8</i>	Valid	Valid	Unsigned divide <i>r/m32</i> by 2, <i>imm8</i> times.
REX.W + C1 /5 <i>ib</i>	SHR <i>r/m64</i> , <i>imm8</i>	Valid	N.E.	Unsigned divide <i>r/m64</i> by 2, <i>imm8</i> times.

**NOTES:**

- \* Not the same form of division as IDIV; rounding is toward negative infinity.
- \*\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.
- \*\*\* See IA-32 Architecture Compatibility section below.

**Description**

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W is used). The count range is limited to 0 to 31 (or 63 if 64-bit mode and REX.W is used). A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value (see Figure 7-9 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*).

## INSTRUCTION SET REFERENCE, N-Z

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the “quotient” of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the “remainder” is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

In 64-bit mode, the instruction’s default operation size is 32 bits and the mask width for CL is 5 bits. Using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using an REX prefix in the form of REX.W promotes operation to 64-bits and sets the mask width for CL to 6 bits. See the summary chart at the beginning of this section for encoding data and limits.

### IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

### Operation

IF 64-Bit Mode and using REX.W

THEN

countMASK ← 3FH;

ELSE

countMASK ← 1FH;

FI

tempCOUNT ← (COUNT AND countMASK);

tempDEST ← DEST;

WHILE (tempCOUNT ≠ 0)

DO

IF instruction is SAL or SHL

THEN

CF ← MSB(DEST);

ELSE (\* Instruction is SAR or SHR \*)

CF ← LSB(DEST);

FI;

IF instruction is SAL or SHL

```

THEN
    DEST ← DEST * 2;
ELSE
    IF instruction is SAR
        THEN
            DEST ← DEST / 2; (* Signed divide, rounding toward negative infinity *)
        ELSE (* Instruction is SHR *)
            DEST ← DEST / 2; (* Unsigned divide *)
    FI;
FI;
tempCOUNT ← tempCOUNT - 1;
OD;

(* Determine overflow for the various instructions *)
IF (COUNT and countMASK) = 1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF ← MSB(DEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF ← 0;
                    ELSE (* Instruction is SHR *)
                        OF ← MSB(tempDEST);
                FI;
        FI;
    ELSE IF (COUNT AND countMASK) = 0
        THEN
            All flags unchanged;
        ELSE (* COUNT not 1 or 0 *)
            OF ← undefined;
    FI;
FI;

```

### Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see “Description” above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

## INSTRUCTION SET REFERENCE, N-Z

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SBB—Integer Subtraction with Borrow

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
1C <i>ib</i>	SBB AL, <i>imm8</i>	Valid	Valid	Subtract with borrow <i>imm8</i> from AL.
1D <i>iw</i>	SBB AX, <i>imm16</i>	Valid	Valid	Subtract with borrow <i>imm16</i> from AX.
1D <i>id</i>	SBB EAX, <i>imm32</i>	Valid	Valid	Subtract with borrow <i>imm32</i> from EAX.
REX.W + 1D <i>id</i>	SBB RAX, <i>imm32</i>	Valid	N.E.	Subtract with borrow sign-extended <i>imm32</i> to 64-bits from RAX.
80 /3 <i>ib</i>	SBB <i>r/m8</i> , <i>imm8</i>	Valid	Valid	Subtract with borrow <i>imm8</i> from <i>r/m8</i> .
REX + 80 /3 <i>ib</i>	SBB <i>r/m8*</i> , <i>imm8</i>	Valid	N.E.	Subtract with borrow <i>imm8</i> from <i>r/m8</i> .
81 /3 <i>iw</i>	SBB <i>r/m16</i> , <i>imm16</i>	Valid	Valid	Subtract with borrow <i>imm16</i> from <i>r/m16</i> .
81 /3 <i>id</i>	SBB <i>r/m32</i> , <i>imm32</i>	Valid	Valid	Subtract with borrow <i>imm32</i> from <i>r/m32</i> .
REX.W + 81 /3 <i>id</i>	SBB <i>r/m64</i> , <i>imm32</i>	Valid	N.E.	Subtract with borrow sign-extended <i>imm32</i> to 64-bits from <i>r/m64</i> .
83 /3 <i>ib</i>	SBB <i>r/m16</i> , <i>imm8</i>	Valid	Valid	Subtract with borrow sign-extended <i>imm8</i> from <i>r/m16</i> .
83 /3 <i>ib</i>	SBB <i>r/m32</i> , <i>imm8</i>	Valid	Valid	Subtract with borrow sign-extended <i>imm8</i> from <i>r/m32</i> .
REX.W + 83 /3 <i>ib</i>	SBB <i>r/m64</i> , <i>imm8</i>	Valid	N.E.	Subtract with borrow sign-extended <i>imm8</i> from <i>r/m64</i> .
18 /r	SBB <i>r/m8</i> , <i>r8</i>	Valid	Valid	Subtract with borrow <i>r8</i> from <i>r/m8</i> .
REX + 18 /r	SBB <i>r/m8*</i> , <i>r8</i>	Valid	N.E.	Subtract with borrow <i>r8</i> from <i>r/m8</i> .
19 /r	SBB <i>r/m16</i> , <i>r16</i>	Valid	Valid	Subtract with borrow <i>r16</i> from <i>r/m16</i> .
19 /r	SBB <i>r/m32</i> , <i>r32</i>	Valid	Valid	Subtract with borrow <i>r32</i> from <i>r/m32</i> .
REX.W + 19 /r	SBB <i>r/m64</i> , <i>r64</i>	Valid	N.E.	Subtract with borrow <i>r64</i> from <i>r/m64</i> .
1A /r	SBB <i>r8</i> , <i>r/m8</i>	Valid	Valid	Subtract with borrow <i>r/m8</i> from <i>r8</i> .
REX + 1A /r	SBB <i>r8*</i> , <i>r/m8*</i>	Valid	N.E.	Subtract with borrow <i>r/m8</i> from <i>r8</i> .
1B /r	SBB <i>r16</i> , <i>r/m16</i>	Valid	Valid	Subtract with borrow <i>r/m16</i> from <i>r16</i> .

## INSTRUCTION SET REFERENCE, N-Z

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
1B /r	SBB <i>r32, r/m32</i>	Valid	Valid	Subtract with borrow <i>r/m32</i> from <i>r32</i> .
REX.W + 1B /r	SBB <i>r64, r/m64</i>	Valid	N.E.	Subtract with borrow <i>r/m64</i> from <i>r64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.

## Description

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SBB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using an REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

$DEST \leftarrow (DEST - (SRC + CF));$

## Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.



**Protected Mode Exceptions**

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.+

## SCAS/SCASB/SCASW/SCASD—Scan String

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
AE	SCAS <i>m8</i>	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI, then set status flags <sup>a</sup> .
AF	SCAS <i>m16</i>	Valid	Valid	Compare AX with word at ES:(E)DI or RDI, then set status flags. <sup>a</sup>
AF	SCAS <i>m32</i>	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags. <sup>a</sup>
REX.W + AF	SCAS <i>m64</i>	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.
AE	SCASB	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI then set status flags. <sup>a</sup>
AF	SCASW	Valid	Valid	Compare AX with word at ES:(E)DI or RDI then set status flags. <sup>a</sup>
AF	SCASD	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags. <sup>a</sup>
REX.W + AF	SCASQ	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.

### NOTES:

- a. In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

### Description

In non-64-bit modes and in default 64-bit mode: this instruction compares a byte, word, doubleword or quadword specified using a memory operand with the value in AL, AX, or EAX. It then sets status flags in EFLAGS recording the results. The memory operand address is read from ES:(E)DI register (depending on the address-size attribute of the instruction and the current operational mode). Note that ES cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed. The explicit-operand form and the no-operands form. The explicit-operand form (specified using the SCAS mnemonic) allows a memory operand to be specified explicitly. The memory operand must be a symbol that indicates the size and location of the operand value. The register operand is then automatically selected to match the size of the memory operand (AL register for byte comparisons, AX for word comparisons, EAX for doubleword comparisons). The explicit-operand form is provided to allow documentation. Note that the documentation provided by this form can be misleading. That is, the memory operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword) but it does not have to specify the correct location. The location is always specified by ES:(E)DI.

The no-operands form of the instruction uses a short form of SCAS. Again, ES:(E)DI is assumed to be the memory operand and AL, AX, or EAX is assumed to be the register operand. The size of operands is selected by the mnemonic: SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison).

After the comparison, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented. The register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

SCAS, SCASB, SCASW, SCASD, and SCASQ can be preceded by the REP prefix for block comparisons of ECX bytes, words, doublewords, or quadwords. Often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of status flags. See “REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

In 64-bit mode, the instruction’s default address size is 64-bits, 32-bit address size is supported using the prefix 67H. Using an REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The 64-bit no-operand mnemonic is SCASQ. Address of the memory operand is specified in either RDI or EDI, and AL/AX/EAX/RAX may be used as the register operand. After a comparison, the destination register is incremented or decremented by the current operand size (depending on the value of the DF flag). See the summary chart at the beginning of this section for encoding data and limits.

## Operation

Non-64-bit Mode:

IF (Byte comparison)

THEN

temp ← AL – SRC;

SetStatusFlags(temp);

THEN IF DF = 0

THEN (E)DI ← (E)DI + 1;

ELSE (E)DI ← (E)DI – 1; FI;

ELSE IF (Word comparison)

THEN

temp ← AX – SRC;

SetStatusFlags(temp);

IF DF = 0

THEN (E)DI ← (E)DI + 2;

ELSE (E)DI ← (E)DI – 2; FI;

FI;

ELSE IF (Doubleword comparison)

THEN

temp ← EAX – SRC;

SetStatusFlags(temp);

IF DF = 0

THEN (E)DI ← (E)DI + 4;

ELSE (E)DI ← (E)DI – 4; FI;

FI;

FI;

## INSTRUCTION SET REFERENCE, N-Z

64-bit Mode:

```
IF (Byte comparison)
  THEN
    temp ← AL – SRC;
    SetStatusFlags(temp);
    THEN IF DF = 0
      THEN (R|E)DI ← (R|E)DI + 1;
      ELSE (R|E)DI ← (R|E)DI – 1; FI;
  ELSE IF (Word comparison)
    THEN
      temp ← AX – SRC;
      SetStatusFlags(temp);
      IF DF = 0
        THEN (R|E)DI ← (R|E)DI + 2;
        ELSE (R|E)DI ← (R|E)DI – 2; FI;
    FI;
  ELSE IF (Doubleword comparison)
    THEN
      temp ← EAX – SRC;
      SetStatusFlags(temp);
      IF DF = 0
        THEN (R|E)DI ← (R|E)DI + 4;
        ELSE (R|E)DI ← (R|E)DI – 4; FI;
    FI;
  ELSE IF (Quadword comparison using REX.W )
    THEN
      temp ← RAX – SRC;
      SetStatusFlags(temp);
      IF DF = 0
        THEN (R|E)DI ← (R|E)DI + 8;
        ELSE (R|E)DI ← (R|E)DI – 8;
      FI;
    FI;
  FI;
F
```

### Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the temporary result of the comparison.

### Protected Mode Exceptions

- |                 |   |
|-----------------|---|
| #GP(0)          | If a memory operand effective address is outside the limit of the ES segment. |
|                 | If the ES register contains a NULL segment selector.                          |
|                 | If an illegal memory operand effective address in the ES segment is given.    |
| #PF(fault-code) | If a page fault occurs.   |

#AC(0)            If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### **Real-Address Mode Exceptions**

#GP                If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS                If a memory operand effective address is outside the SS segment limit.

## INSTRUCTION SET REFERENCE, N-Z

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SETcc—Set Byte on Condition

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 97	SETA <i>r/m8</i>	Valid	Valid	Set byte if above (CF=0 and ZF=0).
REX + 0F 97	SETA <i>r/m8</i> *	Valid	N.E.	Set byte if above (CF=0 and ZF=0).
0F 93	SETAE <i>r/m8</i>	Valid	Valid	Set byte if above or equal (CF=0).
REX + 0F 93	SETAE <i>r/m8</i> *	Valid	N.E.	Set byte if above or equal (CF=0).
0F 92	SETB <i>r/m8</i>	Valid	Valid	Set byte if below (CF=1).
REX + 0F 92	SETB <i>r/m8</i> *	Valid	N.E.	Set byte if below (CF=1).
0F 96	SETBE <i>r/m8</i>	Valid	Valid	Set byte if below or equal (CF=1 or ZF=1).
REX + 0F 96	SETBE <i>r/m8</i> *	Valid	N.E.	Set byte if below or equal (CF=1 or ZF=1).
0F 92	SETC <i>r/m8</i>	Valid	Valid	Set byte if carry (CF=1).
REX + 0F 92	SETC <i>r/m8</i> *	Valid	N.E.	Set byte if carry (CF=1).
0F 94	SETE <i>r/m8</i>	Valid	Valid	Set byte if equal (ZF=1).
REX + 0F 94	SETE <i>r/m8</i> *	Valid	N.E.	Set byte if equal (ZF=1).
0F 9F	SETG <i>r/m8</i>	Valid	Valid	Set byte if greater (ZF=0 and SF=OF).
REX + 0F 9F	SETG <i>r/m8</i> *	Valid	N.E.	Set byte if greater (ZF=0 and SF=OF).
0F 9D	SETGE <i>r/m8</i>	Valid	Valid	Set byte if greater or equal (SF=OF).
REX + 0F 9D	SETGE <i>r/m8</i> *	Valid	N.E.	Set byte if greater or equal (SF=OF).
0F 9C	SETL <i>r/m8</i>	Valid	Valid	Set byte if less (SF≠ OF).
REX + 0F 9C	SETL <i>r/m8</i> *	Valid	N.E.	Set byte if less (SF≠ OF).
0F 9E	SETLE <i>r/m8</i>	Valid	Valid	Set byte if less or equal (ZF=1 or SF≠ OF).
REX + 0F 9E	SETLE <i>r/m8</i> *	Valid	N.E.	Set byte if less or equal (ZF=1 or SF≠ OF).
0F 96	SETNA <i>r/m8</i>	Valid	Valid	Set byte if not above (CF=1 or ZF=1).
REX + 0F 96	SETNA <i>r/m8</i> *	Valid	N.E.	Set byte if not above (CF=1 or ZF=1).
0F 92	SETNAE <i>r/m8</i>	Valid	Valid	Set byte if not above or equal (CF=1).
REX + 0F 92	SETNAE <i>r/m8</i> *	Valid	N.E.	Set byte if not above or equal (CF=1).
0F 93	SETNB <i>r/m8</i>	Valid	Valid	Set byte if not below (CF=0).
REX + 0F 93	SETNB <i>r/m8</i> *	Valid	N.E.	Set byte if not below (CF=0).
0F 97	SETNBE <i>r/m8</i>	Valid	Valid	Set byte if not below or equal (CF=0 and ZF=0).

## INSTRUCTION SET REFERENCE, N-Z

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
REX + 0F 97	SETNBE <i>r/m8</i> *	Valid	N.E.	Set byte if not below or equal (CF=0 and ZF=0).
0F 93	SETNC <i>r/m8</i>	Valid	Valid	Set byte if not carry (CF=0).
REX + 0F 93	SETNC <i>r/m8</i> *	Valid	N.E.	Set byte if not carry (CF=0).
0F 95	SETNE <i>r/m8</i>	Valid	Valid	Set byte if not equal (ZF=0).
REX + 0F 95	SETNE <i>r/m8</i> *	Valid	N.E.	Set byte if not equal (ZF=0).
0F 9E	SETNG <i>r/m8</i>	Valid	Valid	Set byte if not greater (ZF=1 or SF≠ OF)
REX + 0F 9E	SETNG <i>r/m8</i> *	Valid	N.E.	Set byte if not greater (ZF=1 or SF≠ OF).
0F 9C	SETNGE <i>r/m8</i>	Valid	Valid	Set byte if not greater or equal (SF≠ OF).
REX + 0F 9C	SETNGE <i>r/m8</i> *	Valid	N.E.	Set byte if not greater or equal (SF≠ OF).
0F 9D	SETNL <i>r/m8</i>	Valid	Valid	Set byte if not less (SF=OF).
REX + 0F 9D	SETNL <i>r/m8</i> *	Valid	N.E.	Set byte if not less (SF=OF).
0F 9F	SETNLE <i>r/m8</i>	Valid	Valid	Set byte if not less or equal (ZF=0 and SF=OF).
REX + 0F 9F	SETNLE <i>r/m8</i> *	Valid	N.E.	Set byte if not less or equal (ZF=0 and SF=OF).
0F 91	SETNO <i>r/m8</i>	Valid	Valid	Set byte if not overflow (OF=0).
REX + 0F 91	SETNO <i>r/m8</i> *	Valid	N.E.	Set byte if not overflow (OF=0).
0F 9B	SETNP <i>r/m8</i>	Valid	Valid	Set byte if not parity (PF=0).
REX + 0F 9B	SETNP <i>r/m8</i> *	Valid	N.E.	Set byte if not parity (PF=0).
0F 99	SETNS <i>r/m8</i>	Valid	Valid	Set byte if not sign (SF=0).
REX + 0F 99	SETNS <i>r/m8</i> *	Valid	N.E.	Set byte if not sign (SF=0).
0F 95	SETNZ <i>r/m8</i>	Valid	Valid	Set byte if not zero (ZF=0).
REX + 0F 95	SETNZ <i>r/m8</i> *	Valid	N.E.	Set byte if not zero (ZF=0).
0F 90	SETO <i>r/m8</i>	Valid	Valid	Set byte if overflow (OF=1)
REX + 0F 90	SETO <i>r/m8</i> *	Valid	N.E.	Set byte if overflow (OF=1).
0F 9A	SETP <i>r/m8</i>	Valid	Valid	Set byte if parity (PF=1).
REX + 0F 9A	SETP <i>r/m8</i> *	Valid	N.E.	Set byte if parity (PF=1).
0F 9A	SETPE <i>r/m8</i>	Valid	Valid	Set byte if parity even (PF=1).
REX + 0F 9A	SETPE <i>r/m8</i> *	Valid	N.E.	Set byte if parity even (PF=1).
0F 9B	SETPO <i>r/m8</i>	Valid	Valid	Set byte if parity odd (PF=0).
REX + 0F 9B	SETPO <i>r/m8</i> *	Valid	N.E.	Set byte if parity odd (PF=0).
0F 98	SETS <i>r/m8</i>	Valid	Valid	Set byte if sign (SF=1).
REX + 0F 98	SETS <i>r/m8</i> *	Valid	N.E.	Set byte if sign (SF=1).
0F 94	SETZ <i>r/m8</i>	Valid	Valid	Set byte if zero (ZF=1).
REX + 0F 94	SETZ <i>r/m8</i> *	Valid	N.E.	Set byte if zero (ZF=1).

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.



## Description

Sets the destination operand to 0 or 1 depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (*cc*) indicates the condition being tested for.

The terms “above” and “below” are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms “greater” and “less” are associated with the SF and OF flags and refer to the relationship between two signed integer values.

Many of the SET<sub>cc</sub> instruction opcodes have alternate mnemonics. For example, SETG (set byte if greater) and SETNLE (set if not less or equal) have the same opcode and test for the same condition: ZF equals 0 and SF equals 0F. These alternate mnemonics are provided to make code more intelligible. Appendix B, “EFLAGS Condition Codes,” in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 1*, shows the alternate mnemonics for various test conditions.

Some languages represent a logical one as an integer with all bits set. This representation can be obtained by choosing the logically opposite condition for the SET<sub>cc</sub> instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

In IA-64 mode, the operand size is fixed at 8 bits. Use of REX prefix enable uniform addressing to additional byte registers. Otherwise, this instruction’s operation is the same as in legacy mode and compatibility mode.

## Operation

```
IF condition
  THEN DEST ← 1;
  ELSE DEST ← 0;
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

## INSTRUCTION SET REFERENCE, N-Z

### Real-Address Mode Exceptions

- #GP                    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS                    If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

- #GP(0)                If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0)                If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code)      If a page fault occurs.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

- #SS(0)                If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0)                If the memory address is in a non-canonical form.
- #PF(fault-code)      If a page fault occurs.

## SFENCE—Store Fence

Opcode	Instruction	64-Bit Mode	Compat /Leg Mode	Description
0F AE /7	SFENCE	Valid	Valid	Serializes store operations.

### Description

Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction. This serializing operation guarantees that every store instruction that precedes in program order the SFENCE instruction is globally visible before any store instruction that follows the SFENCE instruction is globally visible. The SFENCE instruction is ordered with respect store instructions, other SFENCE instructions, any MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to load instructions or the LFENCE instruction.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of insuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

Wait\_On\_Following\_Stores\_Until(preceding\_stores\_globally\_visible);

### Intel C/C++ Compiler Intrinsic Equivalent

void\_mm\_sfence(void)

### Exceptions (All Operating Modes)

None.

## SGDT—Store Global Descriptor Table Register

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /0	SGDT <i>m</i>	Valid	Valid	Store GDTR to <i>m</i> .

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Description

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a memory location:

In legacy or compatibility mode, the destination operand is a 6-byte memory location. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in bytes 3-5, and byte 6 is zero-filled. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In IA-32e mode, the operand size is fixed at 8+2 bytes. The instruction stores an 8-byte base and a 2-byte limit.

SGDT is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated. See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in Chapter 3, *IA-32 Intel® Architecture Software Developer’s Manual, Volume 2A*, for information on loading the GDTR and IDTR.

### IA-32 Architecture Compatibility

The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium 4, Intel Xeon, P6 processor family, Pentium, Intel486, and Intel386™ processors fill these bits with 0s.

### Operation

IF instruction is SGDT

    IF OperandSize = 16

        THEN

            DEST[0:15] ← GDTR(Limit);

            DEST[16:39] ← GDTR(Base); (\* 24 bits of base address stored \*)

            DEST[40:47] ← 0;

        ELSE IF (32-bit Operand Size)

            DEST[0:15] ← GDTR(Limit);

            DEST[16:47] ← GDTR(Base); (\* Full 32-bit base address stored \*)

        FI;

        ELSE (\* 64-bit Operand Size \*)

            DEST[0:15] ← GDTR(Limit);

DEST[16:79] ← GDTR(Base); (\* Full 64-bit base address stored \*)

FI;

FI;

### Flags Affected

None.

### Protected Mode Exceptions

#UD	If the destination operand is a register.
#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#UD	If the destination operand is a register.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

#UD	If the destination operand is a register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the destination operand is a register.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SHLD—Double Precision Shift Left

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F A4	SHLD <i>r/m16, r16, imm8</i>	Valid	Valid	Shift <i>r/m16</i> to left <i>imm8</i> places while shifting bits from <i>r16</i> in from the right.
0F A5	SHLD <i>r/m16, r16, CL</i>	Valid	Valid	Shift <i>r/m16</i> to left CL places while shifting bits from <i>r16</i> in from the right.
0F A4	SHLD <i>r/m32, r32, imm8</i>	Valid	Valid	Shift <i>r/m32</i> to left <i>imm8</i> places while shifting bits from <i>r32</i> in from the right.
REX.W + 0F A4	SHLD <i>r/m64, r64, imm8</i>	Valid	N.E.	Shift <i>r/m64</i> to left <i>imm8</i> places while shifting bits from <i>r64</i> in from the right.
0F A5	SHLD <i>r/m32, r32, CL</i>	Valid	Valid	Shift <i>r/m32</i> to left CL places while shifting bits from <i>r32</i> in from the right.
REX.W + 0F A5	SHLD <i>r/m64, r64, CL</i>	Valid	N.E.	Shift <i>r/m64</i> to left CL places while shifting bits from <i>r64</i> in from the right.

### Description

The SHLD instruction is used for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or in the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode; only bits 0 through 4 of the count are used. This masks the count to a value between 0 and 31. If a count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction's default operation size is 32 bits. Using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using an REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

## INSTRUCTION SET REFERENCE, N-Z

### Operation

```
IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT ← COUNT MOD 64;
    ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE
        IF COUNT > SIZE
            THEN (* Bad parameters *)
                DEST is undefined;
                CF, OF, SF, ZF, AF, PF are undefined;
            ELSE (* Perform the shift *)
                CF ← BIT[DEST, SIZE – COUNT];
                (* Last bit shifted out on exit *)
                FOR i ← SIZE – 1 DOWN TO COUNT
                    DO
                        Bit(DEST, i) ← Bit(DEST, i – COUNT);
                    OD;
                FOR i ← COUNT – 1 DOWN TO 0
                    DO
                        BIT[DEST, i] ← BIT[Src, i – COUNT + SIZE];
                    OD;
            FI;
FI;
```

### Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

### Protected Mode Exceptions

- |                 |   |
|-----------------|---|
| #GP(0)          | If the destination is located in a non-writable segment.                                  |
|                 | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
|                 | If the DS, ES, FS, or GS register contains a NULL segment selector.                       |
| #SS(0)          | If a memory operand effective address is outside the SS segment limit.                    |
| #PF(fault-code) | If a page fault occurs.   |



#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SHRD—Double Precision Shift Right

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F AC	SHRD <i>r/m16</i> , <i>r16</i> , <i>imm8</i>	Valid	Valid	Shift <i>r/m16</i> to right <i>imm8</i> places while shifting bits from <i>r16</i> in from the left.
0F AD	SHRD <i>r/m16</i> , <i>r16</i> , CL	Valid	Valid	Shift <i>r/m16</i> to right CL places while shifting bits from <i>r16</i> in from the left.
0F AC	SHRD <i>r/m32</i> , <i>r32</i> , <i>mm8</i>	Valid	Valid	Shift <i>r/m32</i> to right <i>imm8</i> places while shifting bits from <i>r32</i> in from the left.
REX.W + 0F AC	SHRD <i>r/m64</i> , <i>r64</i> , <i>imm8</i>	Valid	N.E.	Shift <i>r/m64</i> to right <i>imm8</i> places while shifting bits from <i>r64</i> in from the left.
0F AD	SHRD <i>r/m32</i> , <i>r32</i> , CL	Valid	Valid	Shift <i>r/m32</i> to right CL places while shifting bits from <i>r32</i> in from the left.
REX.W + 0F AD	SHRD <i>r/m64</i> , <i>r64</i> , CL	Valid	N.E.	Shift <i>r/m64</i> to right CL places while shifting bits from <i>r64</i> in from the left.

## Description

The SHRD instruction is useful for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode, the width of the count mask is 5 bits. Only bits 0 through 4 of the count register are used (masking the count to a value between 0 and 31). If the count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction's default operation size is 32 bits. Using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using an REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

```

IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT ← COUNT MOD 64;
    ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE
        IF COUNT > SIZE
            THEN (* Bad parameters *)
                DEST is undefined;
                CF, OF, SF, ZF, AF, PF are undefined;
            ELSE (* Perform the shift *)
                CF ← BIT[DEST, COUNT – 1]; (* Last bit shifted out on exit *)
                FOR i ← 0 TO SIZE – 1 – COUNT
                    DO
                        BIT[DEST, i] ← BIT[DEST, i + COUNT];
                    OD;
                FOR i ← SIZE – COUNT TO SIZE – 1
                    DO
                        BIT[DEST,i] ← BIT[SRC, i + COUNT – SIZE];
                    OD;
            FI;
FI;

```

**Flags Affected**

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

**Protected Mode Exceptions**

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## INSTRUCTION SET REFERENCE, N-Z

### Real-Address Mode Exceptions

- #GP                    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS                    If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

- #GP(0)                If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0)                If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code)      If a page fault occurs.
- #AC(0)                If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

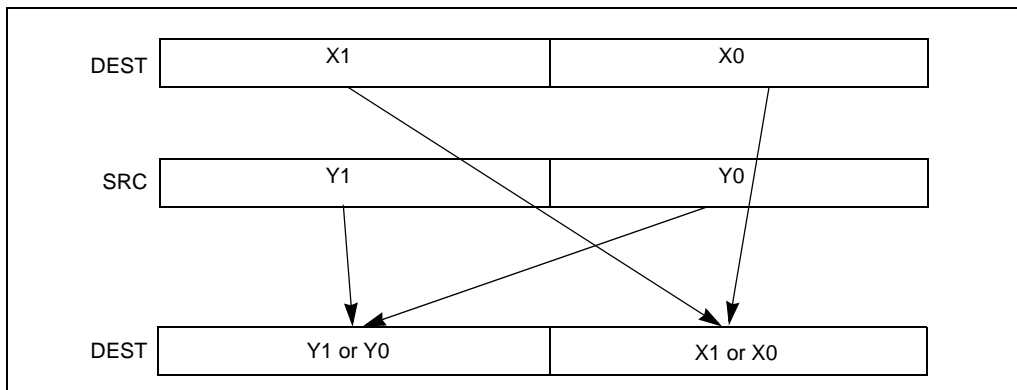
- #SS(0)                If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0)                If the memory address is in a non-canonical form.
- #PF(fault-code)      If a page fault occurs.
- #AC(0)                If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SHUFPD—Shuffle Packed Double-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
66 0F C6 /r ib	SHUFPD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	Valid	Valid	Shuffle packed double-precision floating-point values selected by <i>imm8</i> from <i>xmm1</i> and <i>xmm2/m128</i> to <i>xmm1</i> .

### Description

Moves either of the two packed double-precision floating-point values from destination operand (first operand) into the low quadword of the destination operand; moves either of the two packed double-precision floating-point values from the source operand into to the high quadword of the destination operand (see Figure 4-12). The select operand (third operand) determines which values are moved to the destination operand.



**Figure 4-12. SHUFPD Shuffle Operation**

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The select operand is an 8-bit immediate: bit 0 selects which value is moved from the destination operand to the result (where 0 selects the low quadword and 1 selects the high quadword) and bit 1 selects which value is moved from the source operand to the result. Bits 2 through 7 of the select operand are reserved and must be set to 0.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## INSTRUCTION SET REFERENCE, N-Z

### Operation

```
IF SELECT[0] = 0
    THEN DEST[63:0] ← DEST[63:0];
    ELSE DEST[63:0] ← DEST[127:64]; FI;
IF SELECT[1] = 0
    THEN DEST[127:64] ← SRC[63:0];
    ELSE DEST[127:64] ← SRC[127:64]; FI;
```

### Intel C/C++ Compiler Intrinsic Equivalent

SHUFPD     \_\_m128d \_mm\_shuffle\_pd(\_\_m128d a, \_\_m128d b, unsigned int imm8)

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#GP(0)           For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  
If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0)           For an illegal address in the SS segment.

#PF(fault-code)   For a page fault.

#NM              If CR0.TS[bit 3] = 1.

#UD              If CR0.EM[bit 2] = 1.  
If CR4.OSFXSR[bit 9] = 0.  
If CPUID.01H:EDX.SSE2[bit 26] = 0.

### Real-Address Mode Exceptions

#GP(0)           If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  
If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM              If CR0.TS[bit 3] = 1.

#UD              If CR0.EM[bit 2] = 1.  
If CR4.OSFXSR[bit 9] = 0.  
If CPUID.01H:EDX.SSE2[bit 26] = 0.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.  If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.

## SHUFPS—Shuffle Packed Single-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F C6 /r ib	SHUFPS <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	Valid	Valid	Shuffle packed single-precision floating-point values selected by <i>imm8</i> from <i>xmm1</i> and <i>xmm1/m128</i> to <i>xmm1</i> .

### Description

Moves two of the four packed single-precision floating-point values from the destination operand (first operand) into the low quadword of the destination operand; moves two of the four packed single-precision floating-point values from the source operand (second operand) into to the high quadword of the destination operand (see Figure 4-13). The select operand (third operand) determines which values are moved to the destination operand.

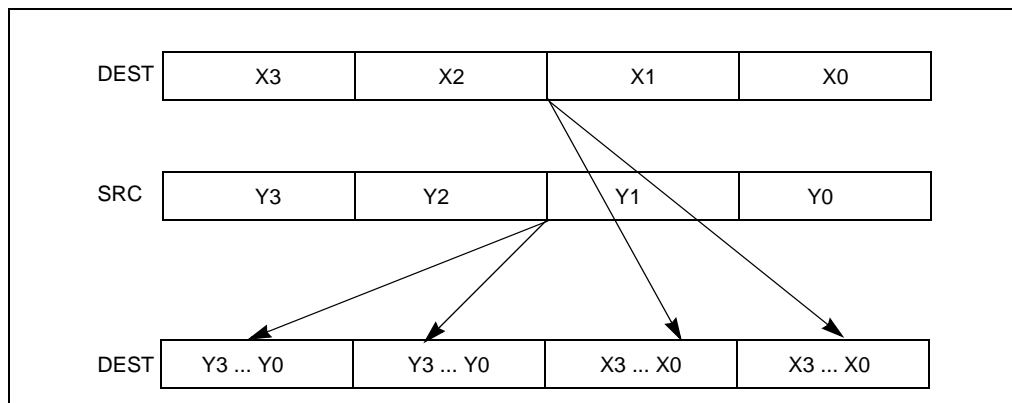


Figure 4-13. SHUFPS Shuffle Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The select operand is an 8-bit immediate: bits 0 and 1 select the value to be moved from the destination operand to the low doubleword of the result, bits 2 and 3 select the value to be moved from the destination operand to the second doubleword of the result, bits 4 and 5 select the value to be moved from the source operand to the third doubleword of the result, and bits 6 and 7 select the value to be moved from the source operand to the high doubleword of the result.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).



**Operation**

```

CASE (SELECT[1:0]) OF
  0: DEST[31:0] ← DEST[31:0];
  1: DEST[31:0] ← DEST[63:32];
  2: DEST[31:0] ← DEST[95:64];
  3: DEST[31:0] ← DEST[127:96];
ESAC;

CASE (SELECT[3:2]) OF
  0: DEST[63:32] ← DEST[31:0];
  1: DEST[63:32] ← DEST[63:32];
  2: DEST[63:32] ← DEST[95:64];
  3: DEST[63:32] ← DEST[127:96];
ESAC;

CASE (SELECT[5:4]) OF
  0: DEST[95:64] ← SRC[31:0];
  1: DEST[95:64] ← SRC[63:32];
  2: DEST[95:64] ← SRC[95:64];
  3: DEST[95:64] ← SRC[127:96];
ESAC;

CASE (SELECT[7:6]) OF
  0: DEST[127:96] ← SRC[31:0];
  1: DEST[127:96] ← SRC[63:32];
  2: DEST[127:96] ← SRC[95:64];
  3: DEST[127:96] ← SRC[127:96];
ESAC;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

SHUFPS      `__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)`

**SIMD Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.

## INSTRUCTION SET REFERENCE, N-Z

#UD                    If CR0.EM[bit 2] = 1.  
                          If CR4.OSFXSR[bit 9] = 0.  
                          If CPUID.01H:EDX.SSE[bit 25] = 0.

### Real-Address Mode Exceptions

#GP(0)                If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  
                          If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM                    If CR0.TS[bit 3] = 1.

#UD                    If CR0.EM[bit 2] = 1.  
                          If CR4.OSFXSR[bit 9] = 0.  
                          If CPUID.01H:EDX.SSE[bit 25] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)      For a page fault.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)                If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)                If memory operand is not aligned on a 16-byte boundary, regardless of segment.  
                          If the memory address is in a non-canonical form.

#PF(fault-code)      For a page fault.

#NM                    If CR0.TS[bit 3] = 1.

#UD                    If CR0.EM[bit 2] = 1.  
                          If CR4.OSFXSR[bit 9] = 0.  
                          If CPUID.01H:EDX.SSE[bit 25] = 0.

## SIDT—Store Interrupt Descriptor Table Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /1	SIDT <i>m</i>	Valid	Valid	Store IDTR to <i>m</i> .

### Description

Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location.

In non-64-bit modes, if the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte filled with 0s.

In 64-bit mode, the operand size fixed at 8+2 bytes. The instruction stores 8-byte base and 2-byte limit values.

SIDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated. See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in Chapter 3, *IA-32 Intel® Architecture Software Developer’s Manual, Volume 2A*, for information on loading the GDTR and IDTR.

### IA-32 Architecture Compatibility

The 16-bit form of SIDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium 4, Intel Xeon, P6 processor family, Pentium, Intel486, and Intel386 processors fill these bits with 0s.

### Operation

```

IF instruction is SIDT
  THEN
    IF OperandSize = 16
      THEN
        DEST[0:15] ← IDTR(Limit);
        DEST[16:39] ← IDTR(Base); (* 24 bits of base address stored; *)
        DEST[40:47] ← 0;
      ELSE IF (32-bit Operand Size)
        DEST[0:15] ← IDTR(Limit);
        DEST[16:47] ← IDTR(Base); FI; (* Full 32-bit base address stored *)
      ELSE (* 64-bit Operand Size *)
        DEST[0:15] ← IDTR(Limit);
        DEST[16:79] ← IDTR(Base); (* Full 64-bit base address stored *)
    FI;
  FI;

```

## INSTRUCTION SET REFERENCE, N-Z

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If the destination is located in a non-writable segment.  
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the destination operand is a register.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SLDT—Store Local Descriptor Table Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /0	SLDT <i>r/m16</i>	Valid	Valid	Stores segment selector from LDTR in <i>r/m16</i> .
REX.W + 0F 00 /0	SLDT <i>r64/m16</i>	Valid	Valid	Stores segment selector from LDTR in <i>r64/m16</i> .

### Description

Stores the segment selector from the local descriptor table register (LDTR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the segment descriptor (located in the GDT) for the current LDT. This instruction can only be executed in protected mode.

Outside IA-32e mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared for the Pentium 4, Intel Xeon, and P6 family processors. They are undefined for Pentium, Intel486, and Intel386 processors. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In compatibility mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). The behavior of SLDT with a 64-bit register is to zero-extend the 16-bit selector and store it in the register. If the destination is memory and operand size is 64, SLDT will write the 16-bit selector to memory as a 16-bit quantity, regardless of the operand size.

### Operation

DEST ← LDTR(SegmentSelector);

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If the destination is located in a non-writable segment.
- If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#UD	The SLDT instruction is not recognized in real-address mode.
-----	--

### Virtual-8086 Mode Exceptions

#UD	The SLDT instruction is not recognized in virtual-8086 mode.
-----	--

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SMSW—Store Machine Status Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /4	SMSW <i>r/m16</i>	Valid	Valid	Store machine status word to <i>r/m16</i> .
0F 01 /4	SMSW <i>r32/m16</i>	Valid	Valid	Store machine status word in low-order 16 bits of <i>r32/m16</i> ; high-order 16 bits of <i>r32</i> are undefined.
REX.W + 0F 01 /4	SMSW <i>r64/m16</i>	Valid	Valid	Store machine status word in low-order 16 bits of <i>r64/m16</i> ; high-order 16 bits of <i>r32</i> are undefined.

### Description

Stores the machine status word (bits 0 through 15 of control register CR0) into the destination operand. The destination operand can be a general-purpose register or a memory location.

In non-64-bit modes, when the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the high-order 16 bits are undefined. When the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, the behavior of the SMSW instruction is defined by the following examples:

- SMSW *r16* operand size 16, store CR0[15:0] in *r16*
- SMSW *r32* operand size 32, zero-extend CR0[31:0], and store in *r32*
- SMSW *r64* operand size 64, zero-extend CR0[63:0], and store in *r64*
- SMSW *m16* operand size 16, store CR0[15:0] in *m16*
- SMSW *m16* operand size 32, store CR0[15:0] in *m16* (not *m32*)
- SMSW *m16* operands size 64, store CR0[15:0] in *m16* (not *m64*)

SMSW is only useful in operating-system software. However, it is not a privileged instruction and can be used in application programs. The is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on the Pentium 4, Intel Xeon, P6 family, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the machine status word.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 21 of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

DEST ← CR0[15:0];  
(\* Machine status word \*)



**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 51 /r	SQRTPD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Computes square roots of the packed double-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .

### Description

Performs an SIMD computation of the square roots of the two packed double-precision floating-point values in the source operand (second operand) stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of an SIMD double-precision floating-point operation.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[63:0] ← SQRT(SRC[63:0]);
DEST[127:64] ← SQRT(SRC[127:64]);
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
SQRTPD      __m128d _mm_sqrt_pd (m128d a)
```

### SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.

#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.CR4.OSXMMEXCPT(bit 10) is 1. If CPUID.01H:EDX.SSE2[bit 26] = 0.

### Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.

## INSTRUCTION SET REFERENCE, N-Z

#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.

## SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 51 /r	SQRTPS <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Computes square roots of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .

### Description

Performs an SIMD computation of the square roots of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of an SIMD single-precision floating-point operation.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[31:0] ← SQRT(SRC[31:0]);
DEST[63:32] ← SQRT(SRC[63:32]);
DEST[95:64] ← SQRT(SRC[95:64]);
DEST[127:96] ← SQRT(SRC[127:96]);
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
SQRTPS      __m128 _mm_sqrt_ps(__m128 a)
```

### SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.

If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0) For an illegal address in the SS segment.

#PF(fault-code) For a page fault.

## INSTRUCTION SET REFERENCE, N-Z

#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

### Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

## SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F 51 /r	SQRTSD <i>xmm1</i> , <i>xmm2/m64</i>	Valid	Valid	Computes square root of the low double-precision floating-point value in <i>xmm2/m64</i> and stores the results in <i>xmm1</i> .

### Description

Computes the square root of the low double-precision floating-point value in the source operand (second operand) and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of a scalar double-precision floating-point operation.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[63:0] ← SQRT(SRC[63:0]);  
(\* DEST[127:64] unchanged \*)

### Intel C/C++ Compiler Intrinsic Equivalent

SQRTSD        \_\_m128d \_mm\_sqrt\_sd (m128d a)

### SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.



#UD	<p>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</p> <p>If CR0.EM[bit 2] = 1.</p> <p>If CR4.OSFXSR[bit 9] = 0.</p> <p>If CPUID.01H:EDX.SSE2[bit 26] = 0.</p>
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	<p>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</p> <p>If CR0.EM[bit 2] = 1.</p> <p>If CR4.OSFXSR[bit 9] = 0.</p> <p>If CPUID.01H:EDX.SSE2[bit 26] = 0.</p>

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 51 /r	SQRTSS <i>xmm1</i> , <i>xmm2/m32</i>	Valid	Valid	Computes square root of the low single-precision floating-point value in <i>xmm2/m32</i> and stores the results in <i>xmm1</i> .

### Description

Computes the square root of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[31:0] ← SQRT (SRC[31:0]);  
(\* DEST[127:64] unchanged \*)

### Intel C/C++ Compiler Intrinsic Equivalent

SQRTSS        \_\_m128 \_mm\_sqrt\_ss(\_\_m128 a)

### SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMM-MEXCPT[bit 10] = 1.

## INSTRUCTION SET REFERENCE, N-Z

#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## STC—Set Carry Flag

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F9	STC	Valid	Valid	Set CF flag.

### Description

Sets the CF flag in the EFLAGS register.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$CF \leftarrow 1$ ;

### Flags Affected

The CF flag is set. The OF, ZF, SF, AF, and PF flags are unaffected.

### Exceptions (All Operating Modes)

None.

## STD—Set Direction Flag

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
FD	STD	Valid	Valid	Set DF flag.

### Description

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

DF ← 1;

### Flags Affected

The DF flag is set. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

### Exceptions (All Operating Modes)

None.

## STI—Set Interrupt Flag

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
FB	STI	Valid	Valid	Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction.

### Description

If protected-mode virtual interrupts are not enabled, STI sets the interrupt flag (IF) in the EFLAGS register. After the IF flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an STI instruction is followed by an RET instruction, the RET instruction is allowed to execute before external interrupts are recognized<sup>2</sup>. If the STI instruction is followed by a CLI instruction (which clears the IF flag), the effect of the STI instruction is negated.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and NMI interrupts. NMI interrupts may be blocked for one macroinstruction following an STI.

When protected-mode virtual interrupts are enabled, CPL is 3, and IOPL is less than 3; STI sets the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 4-4 indicates the action of the STI instruction depending on the processor's mode of operation and the CPL/IOPL settings of the running program or procedure.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

---

2. The STI instruction delays recognition of interrupts only if it is executed with EFLAGS.IF = 0. In a sequence of STI instructions, only the first instruction in the sequence is guaranteed to delay interrupts. In the following instruction sequence, interrupts may be recognized before RET executes:

```
STI
STI
RET
```



Table 4-4. Decision Table for STI Results

PE	VM	IOPL	CPL	PVI	VIP	VME	STI Result
0	X	X	X	X	X	X	IF = 1
1	0	≥ CPL	X	X	X	X	IF = 1
1	0	< CPL	3	1	0	X	VIF = 1
1	0	< CPL	< 3	X	X	X	GP Fault
1	0	< CPL	X	0	X	X	GP Fault
1	0	< CPL	X	X	1	X	GP Fault
1	1	3	X	X	X	X	IF = 1
1	1	< 3	X	X	0	1	VIF = 1
1	1	< 3	X	X	1	X	GP Fault
1	1	< 3	X	X	X	0	GP Fault
X = This setting has no impact.							

## Operation

```

IF PE = 0 (* Executing in real-address mode *)
  THEN
    IE ← 1; (* Set Interrupt Flag *)
  ELSE (* Executing in protected mode or virtual-8086 mode *)
    IF VM = 0 (* Executing in protected mode*)
      THEN
        IF IOPL ≥ CPL
          THEN
            IE ← 1; (* Set Interrupt Flag *)
          ELSE
            IF (IOPL < CPL) and (CPL = 3) and (VIP = 0)
              THEN
                VIE ← 1; (* Set Virtual Interrupt Flag *)
              ELSE
                #GP(0);
            FI;
          FI;
        ELSE (* Executing in Virtual-8086 mode *)
          IF IOPL = 3
            THEN
              IE ← 1; (* Set Interrupt Flag *)
            ELSE

```

## INSTRUCTION SET REFERENCE, N-Z

```
IF ((IOPL < 3) and (VIP = 0) and (VME = 1))
    THEN
        VIF ← 1; (* Set Virtual Interrupt Flag *)
    ELSE
        #GP(0); (* Trap to virtual-8086 monitor *)
    FI;
FI;
```

### Flags Affected

The IF flag is set to 1; or the VIF flag is set to 1.

### Protected Mode Exceptions

#GP(0) If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

#GP(0) If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

Same exceptions as in Protected Mode.

## STMXCSR—Store MXCSR Register State

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F AE /3	STMXCSR <i>m32</i>	Valid	Valid	Store contents of MXCSR register to <i>m32</i> .

### Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$m32 \leftarrow \text{MXCSR}$ ;

### Intel C/C++ Compiler Intrinsic Equivalent

`_mm_getcsr(void)`

### Exceptions

None.

### Numeric Exceptions

None.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CR0.EM[bit 2] = 1.
#NM	If CR0.TS[bit 3] = 1.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true: CR0.AM[bit 18] = 1, EFLAGS.AC[bit 18] = 1, current CPL = 3.
#UD	If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

## INSTRUCTION SET REFERENCE, N-Z

### Real Address Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#UD	If CR0.EM[bit 2] = 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

### Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CR0.EM[bit 2] = 1.
#NM	If CR0.TS[bit 3] = 1.
#AC	For unaligned memory reference. To enable #AC exceptions, three conditions must be true: CR0.AM[bit 18] = 1, EFLAGS.AC[bit 18] = 1, current CPL = 3
#UD	If CR4.OSFXSR[bit 9] = 0.
#UD	If CPUID.01H:EDX.SSE[bit 25] = 0.

## STOS/STOSB/STOSW/STOSD/STOSQ—Store String

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
AA	STOS <i>m8</i>	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOS <i>m16</i>	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOS <i>m32</i>	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOS <i>m64</i>	Valid	N.E.	Store RAX at address RDI or EDI.
AA	STOSB	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOSW	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOSD	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOSQ	Valid	N.E.	Store RAX at address RDI or EDI.

## Description

In non-64-bit and default 64-bit mode; stores a byte, word, or doubleword from the AL, AX, or EAX register (respectively) into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or ES:DI register (depending on the address-size attribute of the instruction and the mode of operation). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of the instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the STOS mnemonic) allows the destination operand to be specified explicitly. Here, the destination operand should be a symbol that indicates the size and location of the destination value. The source operand is then automatically selected to match the size of the destination operand (the AL register for byte operands, AX for word operands, EAX for doubleword operands). The explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI register. These must be loaded correctly before the store string instruction is executed.

The no-operands form provides “short forms” of the byte, word, doubleword, and quadword versions of the STOS instructions. Here also ES:(E)DI is assumed to be the destination operand and AL, AX, or EAX is assumed to be the source operand. The size of the destination and source

## INSTRUCTION SET REFERENCE, N-Z

operands is selected by the mnemonic: STOSB (byte read from register AL), STOSW (word from AX), STOSD (doubleword from EAX).

After the byte, word, or doubleword is transferred from the register to the memory location, the (E)DI register is incremented or decremented according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the register is incremented; if the DF flag is 1, the register is decremented (the register is incremented or decremented by 1 for byte operations, by 2 for word operations, by 4 for doubleword operations).

In 64-bit mode, the default address size is 64 bits, 32-bit address size is supported using the prefix 67H. Using an REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The promoted no-operand mnemonic is STOSQ. STOSQ (and its explicit operands variant) store a quadword from the RAX register into the destination addressed by RDI or EDI. See the summary chart at the beginning of this section for encoding data and limits.

The STOS, STOSB, STOSW, STOSD, STOSQ instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because data needs to be moved into the AL, AX, or EAX register before it can be stored. See “REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

### Operation

Non-64-bit Mode:

```
IF (Byte store)
  THEN
    DEST ← AL;
    THEN IF DF = 0
      THEN (E)DI ← (E)DI + 1;
      ELSE (E)DI ← (E)DI - 1;
    FI;
  ELSE IF (Word store)
    THEN
      DEST ← AX;
      THEN IF DF = 0
        THEN (E)DI ← (E)DI + 2;
        ELSE (E)DI ← (E)DI - 2;
      FI;
    FI;
  ELSE IF (Doubleword store)
    THEN
      DEST ← EAX;
      THEN IF DF = 0
        THEN (E)DI ← (E)DI + 4;
        ELSE (E)DI ← (E)DI - 4;
      FI;
    FI;
  FI;
```

64-bit Mode:

```

IF (Byte store)
  THEN
    DEST ← AL;
    THEN IF DF = 0
      THEN (R|E)DI ← (R|E)DI + 1;
      ELSE (R|E)DI ← (R|E)DI - 1;
    FI;
  ELSE IF (Word store)
    THEN
      DEST ← AX;
      THEN IF DF = 0
        THEN (R|E)DI ← (R|E)DI + 2;
        ELSE (R|E)DI ← (R|E)DI - 2;
      FI;
    FI;
  ELSE IF (Doubleword store)
    THEN
      DEST ← EAX;
      THEN IF DF = 0
        THEN (R|E)DI ← (R|E)DI + 4;
        ELSE (R|E)DI ← (R|E)DI - 4;
      FI;
    FI;
  ELSE IF (Quadword store using REX.W )
    THEN
      DEST ← RAX;
      THEN IF DF = 0
        THEN (R|E)DI ← (R|E)DI + 8;
        ELSE (R|E)DI ← (R|E)DI - 8;
      FI;
    FI;
  FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

- |                 |   |
|-----------------|---|
| #GP(0)          | If the destination is located in a non-writable segment.<br>If a memory operand effective address is outside the limit of the ES segment. |
|                 | If the ES register contains a NULL segment selector.  |
| #PF(fault-code) | If a page fault occurs.   |

## INSTRUCTION SET REFERENCE, N-Z

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the ES segment limit.

### Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the ES segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## STR—Store Task Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /1	STR <i>r/m16</i>	Valid	Valid	Stores segment selector from TR in <i>r/m16</i> .

### Description

Stores the segment selector from the task register (TR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the task state segment (TSS) for the currently running task.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

In 64-bit mode, operation is the same. The size of the memory operand is fixed at 16 bits. In register stores, the 2-byte TR is zero extended if stored to a 64-bit register.

The STR instruction is useful only in operating-system software. It can only be executed in protected mode.

### Operation

DEST ← TR(SegmentSelector);

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If the destination is a memory operand that is located in a non-writable segment or if the effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

- #UD The STR instruction is not recognized in real-address mode.

## INSTRUCTION SET REFERENCE, N-Z

### Virtual-8086 Mode Exceptions

#UD                      The STR instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#GP(0)                  If the memory address is in a non-canonical form.

#SS(U)                  If the stack address is in a non-canonical form.

#PF(fault-code)        If a page fault occurs.

#AC(0)                  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SUB—Subtract

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
2C <i>ib</i>	SUB AL, <i>imm8</i>	Valid	Valid	Subtract <i>imm8</i> from AL.
2D <i>iw</i>	SUB AX, <i>imm16</i>	Valid	Valid	Subtract <i>imm16</i> from AX.
2D <i>id</i>	SUB EAX, <i>imm32</i>	Valid	Valid	Subtract <i>imm32</i> from EAX.
REX.W + 2D <i>id</i>	SUB RAX, <i>imm32</i>	Valid	N.E.	Subtract <i>imm32</i> sign-extended to 64-bits from RAX.
80 /5 <i>ib</i>	SUB <i>r/m8</i> , <i>imm8</i>	Valid	Valid	Subtract <i>imm8</i> from <i>r/m8</i> .
REX + 80 /5 <i>ib</i>	SUB <i>r/m8*</i> , <i>imm8</i>	Valid	N.E.	Subtract <i>imm8</i> from <i>r/m8</i> .
81 /5 <i>iw</i>	SUB <i>r/m16</i> , <i>imm16</i>	Valid	Valid	Subtract <i>imm16</i> from <i>r/m16</i> .
81 /5 <i>id</i>	SUB <i>r/m32</i> , <i>imm32</i>	Valid	Valid	Subtract <i>imm32</i> from <i>r/m32</i> .
REX.W + 81 /5 <i>id</i>	SUB <i>r/m64</i> , <i>imm32</i>	Valid	N.E.	Subtract <i>imm32</i> sign-extended to 64-bits from <i>r/m64</i> .
83 /5 <i>ib</i>	SUB <i>r/m16</i> , <i>imm8</i>	Valid	Valid	Subtract sign-extended <i>imm8</i> from <i>r/m16</i> .
83 /5 <i>ib</i>	SUB <i>r/m32</i> , <i>imm8</i>	Valid	Valid	Subtract sign-extended <i>imm8</i> from <i>r/m32</i> .
REX.W + 83 /5 <i>ib</i>	SUB <i>r/m64</i> , <i>imm8</i>	Valid	N.E.	Subtract sign-extended <i>imm8</i> from <i>r/m64</i> .
28 /r	SUB <i>r/m8</i> , <i>r8</i>	Valid	Valid	Subtract <i>r8</i> from <i>r/m8</i> .
REX + 28 /r	SUB <i>r/m8*</i> , <i>r8*</i>	Valid	N.E.	Subtract <i>r8</i> from <i>r/m8</i> .
29 /r	SUB <i>r/m16</i> , <i>r16</i>	Valid	Valid	Subtract <i>r16</i> from <i>r/m16</i> .
29 /r	SUB <i>r/m32</i> , <i>r32</i>	Valid	Valid	Subtract <i>r32</i> from <i>r/m32</i> .
REX.W + 29 /r	SUB <i>r/m64</i> , <i>r32</i>	Valid	N.E.	Subtract <i>r64</i> from <i>r/m64</i> .
2A /r	SUB <i>r8</i> , <i>r/m8</i>	Valid	Valid	Subtract <i>r/m8</i> from <i>r8</i> .
REX + 2A /r	SUB <i>r8*</i> , <i>r/m8*</i>	Valid	N.E.	Subtract <i>r/m8</i> from <i>r8</i> .
2B /r	SUB <i>r16</i> , <i>r/m16</i>	Valid	Valid	Subtract <i>r/m16</i> from <i>r16</i> .
2B /r	SUB <i>r32</i> , <i>r/m32</i>	Valid	Valid	Subtract <i>r/m32</i> from <i>r32</i> .
REX.W + 2B /r	SUB <i>r64</i> , <i>r/m64</i>	Valid	N.E.	Subtract <i>r/m64</i> from <i>r64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.

## Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

## INSTRUCTION SET REFERENCE, N-Z

The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

In 64-bit mode, the instruction's default operation size is 32 bits. Using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using an REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

### Operation

DEST ← (DEST – SRC);

### Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SUBPD—Subtract Packed Double-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 5C /r	SUBPD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Subtract packed double-precision floating-point values in <i>xmm2/m128</i> from <i>xmm1</i> .

### Description

Performs an SIMD subtract of the two packed double-precision floating-point values in the source operand (second operand) from the two packed double-precision floating-point values in the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of an SIMD double-precision floating-point operation.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[63:0] ← DEST[63:0] – SRC[63:0];  
DEST[127:64] ← DEST[127:64] – SRC[127:64];

### Intel C/C++ Compiler Intrinsic Equivalent

SUBPD            \_\_m128d \_mm\_sub\_pd (m128d a, m128d b)

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

#UD                    If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.  
                           If CR0.EM[bit 2] = 1.  
                           If CR4.OSFXSR[bit 9] = 1.  
                           If CPUID.01H:EDX.SSE2[bit 26] = 0.

### Real-Address Mode Exceptions

#GP(0)                If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  
                           If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM                    If CR0.TS[bit 3] = 1.

#XM                    If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

#UD                    If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.  
                           If CR0.EM[bit 2] = 1.  
                           If CR4.OSFXSR[bit 9] = 0.  
                           If CPUID.01H:EDX.SSE2[bit 26] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)      For a page fault.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.



## SUBPS—Subtract Packed Single-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 5C /r	SUBPS <i>xmm1</i> <i>xmm2/mem128</i>	Valid	Valid	Subtract packed single-precision floating-point values in <i>xmm2/mem</i> from <i>xmm1</i> .

### Description

Performs an SIMD subtract of the four packed single-precision floating-point values in the source operand (second operand) from the four packed single-precision floating-point values in the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of an SIMD double-precision floating-point operation.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[31:0] ← DEST[31:0] – SRC[31:0];
DEST[63:32] ← DEST[63:32] – SRC[63:32];
DEST[95:64] ← DEST[95:64] – SRC[95:64];
DEST[127:96] ← DEST[127:96] – SRC[127:96];
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
SUBPS      __m128 _mm_sub_ps(__m128 a, __m128 b)
```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.

## INSTRUCTION SET REFERENCE, N-Z

#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

### Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

## SUBSD—Subtract Scalar Double-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F 5C /r	SUBSD <i>xmm1</i> , <i>xmm2/mem64</i>	Valid	Valid	Subtracts the low double-precision floating-point values in <i>xmm2/mem64</i> from <i>xmm1</i> .

### Description

Subtracts the low double-precision floating-point value in the source operand (second operand) from the low double-precision floating-point value in the destination operand (first operand), and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of a scalar double-precision floating-point operation.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[63:0] ← DEST[63:0] – SRC[63:0];  
(\* DEST[127:64] unchanged \*)

### Intel C/C++ Compiler Intrinsic Equivalent

SUBSD            \_\_m128d \_mm\_sub\_sd (m128d a, m128d b)

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

#UD	<p>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</p> <p>If CR0.EM[bit 2] = 1.</p> <p>If CR4.OSFXSR[bit 9] = 0.</p> <p>If CPUID.01H:EDX.SSE2[bit 26] = 0.</p>
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	<p>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</p> <p>If CR0.EM[bit 2] = 1.</p> <p>If CR4.OSFXSR[bit 9] = 0.</p> <p>If CPUID.01H:EDX.SSE2[bit 26] = 0.</p>

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If $CR0.TS[\text{bit } 3] = 1$ .
#XM	If an unmasked SIMD floating-point exception and $CR4.OSXMMEXCPT[\text{bit } 10] = 1$ .
#UD	If an unmasked SIMD floating-point exception and $CR4.OSXMMEXCPT[\text{bit } 10] = 0$ . If $CR0.EM[\text{bit } 2] = 1$ . If $CR4.OSFXSR[\text{bit } 9] = 0$ . If $CPUID.01H:EDX.SSE2[\text{bit } 26] = 0$ .
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SUBSS—Subtract Scalar Single-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 5C /r	SUBSS <i>xmm1</i> , <i>xmm2/m32</i>	Valid	Valid	Subtract the lower single-precision floating-point values in <i>xmm2/m32</i> from <i>xmm1</i> .

### Description

Subtracts the low single-precision floating-point value in the source operand (second operand) from the low single-precision floating-point value in the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[31:0] ← DEST[31:0] – SRC[31:0];  
(\* DEST[127:96] unchanged \*)

### Intel C/C++ Compiler Intrinsic Equivalent

SUBSS            \_\_m128 \_mm\_sub\_ss(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

## INSTRUCTION SET REFERENCE, N-Z

#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.



**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## SWAPGS—Swap GS Base Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 7	SWAPGS	Valid	Invalid	Exchanges the current GS base register value with the value contained in MSR address C0000102H.

### Description

SWAPGS exchanges the current GS base register value with the value contained in MSR address C0000102H (MSR\_KERNELGSbase). KernelGSbase is guaranteed to be canonical; so SWAPGS does not perform a canonical check. The SWAPGS instruction is a privileged instruction intended for use by system software.

When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel can't save general purpose registers or reference memory.

By design, SWAPGS does not require any general purpose registers or memory operands. No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the KernelGSbase MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures. Similarly, when the OS kernel is entered using an interrupt or exception (where the kernel stack is already set up), SWAPGS can be used to quickly get a pointer to the kernel data structures.

The KernelGSbase MSR itself is only accessible using RDMSR/WRMSR instructions. Those instructions are only accessible at privilege level 0. WRMSR will cause a #GP(0) if the value to be written to KernelGSbase MSR is non-canonical.

See Table 4-5.

**Table 4-5. SWAPGS Operation Parameters**

Opcode	ModR/M Byte			Instruction	
	MOD	REG	R/M	Not 64-bit Mode	64-bit Mode
0F 01	MOD ≠ 11	111	xxx	INVLPG	INVLPG
	11	111	000	#UD	SWPGS
	11	111	≠ 000	#UD	#UD

**Operation**

```

IF CS.L ≠ 1 (* Not in 64-Bit Mode *)
  THEN
    #UD; FI;
IF CPL ≠ 0
  THEN #GP(0); FI;
tmp ← GS(BASE);
GS(BASE) ← KERNELGSbase;
KERNELGSbase ← tmp;

```

**Flags Affected**

None

**Protected Mode Exceptions**

#UD If Mode ≠ 64-Bit

**Real-Address Mode Exceptions**

#UD Instruction not recognized.

**Virtual-8086 Mode Exceptions**

#UD Instruction not recognized.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#GP(0) If CPL ≠ 0.

**SYSCALL—Fast System Call**

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 05	SYSCALL	Valid	Invalid	Fast call to privilege level 0 system procedures.

**Description**

SYSCALL saves the RIP of the instruction following SYSCALL to RCX and loads a new RIP from the IA32\_LSTAR (64-bit mode). Upon return, SYSRET copies the value saved in RCX to the RIP.

SYSCALL saves RFLAGS (lower 32 bit only) in R11. It then masks RFLAGS with an OS-defined value using the IA32\_FMASK (MSR C000\_0084). The actual mask value used by the OS is the complement of the value written to the IA32\_FMASK MSR. None of the bits in RFLAGS are automatically cleared (except for RF). SYSRET restores RFLAGS from R11 (the lower 32 bits only).

Software should not alter the CS or SS descriptors in a manner that violates the following assumptions made by SYSCALL/SYSRET:

- The CS and SS base and limit remain the same for all processes, including the operating system (the base is 0H and the limit is 0FFFFFFFH).
- The CS of the SYSCALL target has a privilege level of 0.
- The CS of the SYSRET target has a privilege level of 3.

SYSCALL/SYSRET do not check for violations of these assumptions.

**Operation**

```
IF ((CS ≠ 1 ) or (IA32_EFER.SCE ≠ 1))
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
  THEN #UD; FI;
RCX ← RIP;
RIP ← LSTAR_MSR;
R11 ← EFLAGS;
EFLAGS ← (EFLAGS MASKED BY IA32_FMASK);
CPL ← 0;
CS(SEL) ← IA32_STAR_MSR[47:32];
CS(DPL) ← 0;
CS(BASE) ← 0;
CS(LIMIT) ← 0xFFFFF;
CS(GRANULAR) ← 1;
SS(SEL) ← IA32_STAR_MSR[47:32] + 8;
SS(DPL) ← 0;
SS(BASE) ← 0;
```

SS(LIMIT)  $\leftarrow$  0xFFFFF;  
SS(GRANULAR)  $\leftarrow$  1;

**Flags Affected**

All.

**Protected Mode Exceptions**

#UD                    If Mode  $\neq$  64-bit.

**Real-Address Mode Exceptions**

#UD                    Instruction is not recognized in this mode.

**Virtual-8086 Mode Exceptions**

#UD                    Instruction is not recognized in this mode.

**Compatibility Mode Exceptions**

#UD                    Instruction is not recognized in this mode.

**64-Bit Mode Exceptions**

#UD                    If IA32\_EFER.SCE = 0.

## SYSENTER—Fast System Call

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 34	SYSENTER	Valid	Valid	Fast call to privilege level 0 system procedures.

### Description

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- IA32\_SYSENTER\_CS — Contains a 32-bit value, of which the lower 16 bits are the segment selector for the privilege level 0 code segment. This value is also used to compute the segment selector of the privilege level 0 stack segment.
- IA32\_SYSENTER\_EIP — Contains the 32-bit offset into the privilege level 0 code segment to the first instruction of the selected operating procedure or routine.
- IA32\_SYSENTER\_ESP — Contains the 32-bit stack pointer for the privilege level 0 stack.

These MSRs can be read from and written to using RDMSR/WRMSR. Register addresses are listed in Table 4-6. The addresses are defined to remain fixed for future IA-32 processors.

**Table 4-6. MSRs Used By the SYSENTER and SYSEXIT Instructions**

MSR	Address
IA32_SYSENTER_CS	174H
IA32_SYSENTER_ESP	175H
IA32_SYSENTER_EIP	176H

When SYSENTER is executed, the processor:

1. Loads the segment selector from the IA32\_SYSENTER\_CS into the CS register.
2. Loads the instruction pointer from the IA32\_SYSENTER\_EIP into the EIP register.
3. Adds 8 to the value in IA32\_SYSENTER\_CS and loads it into the SS register.
4. Loads the stack pointer from the IA32\_SYSENTER\_ESP into the ESP register.
5. Switches to privilege level 0.

6. Clears the VM flag in the EFLAGS register, if the flag is set.
7. Begins executing the selected system procedure.

The processor does not save a return IP or other state information for the calling procedure.

The SYSENTER instruction always transfers program control to a protected-mode code segment with a DPL of 0. The instruction requires that the following conditions are met by the operating system:

- The segment descriptor for the selected system code segment selects a flat, 32-bit code segment of up to 4 GBytes, with execute, read, accessed, and non-conforming permissions.
- The segment descriptor for selected system stack segment selects a flat 32-bit stack segment of up to 4 GBytes, with read, write, accessed, and expand-up permissions.

The SYSENTER can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code, and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in the global descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER\_CS\_MSR MSR.
- The fast system call “stub” routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
  THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
    THEN
      SYSENTER/SYSEXIT_Not_Supported; FI;
    ELSE
      SYSENTER/SYSEXIT_Supported; FI;
  FI;
```

## INSTRUCTION SET REFERENCE, N-Z

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

### Operation

```
IF CR0.PE = 0 THEN #GP(0); FI;
IF SYSENTER_CS_MSR[15:2] = 0 THEN #GP(0); FI;
EFLAGS.VM ← 0; (* Insures protected mode execution *)
EFLAGS.IF ← 0; (* Mask interrupts *)
EFLAGS.RF ← 0;

CS.SEL ← SYSENTER_CS_MSR (* Operating system provides CS *)
(* Set rest of CS to a fixed value *)
CS.BASE ← 0; (* Flat segment *)
CS.LIMIT ← FFFFFFFH; (* 4-GByte limit *)
CS.ARbyte.G ← 1; (* 4-KByte granularity *)
CS.ARbyte.S ← 1;
CS.ARbyte.TYPE ← 1011B; (* Execute + Read, Accessed *)
CS.ARbyte.D ← 1; (* 32-bit code segment*)
CS.ARbyte.DPL ← 0;
CS.SEL.RPL ← 0;
CS.ARbyte.P ← 1;
CPL ← 0;

SS.SEL ← CS.SEL + 8;
(* Set rest of SS to a fixed value *)
SS.BASE ← 0; (* Flat segment *)
SS.LIMIT ← FFFFFFFH; (* 4-GByte limit *)
SS.ARbyte.G ← 1; (* 4-KByte granularity *)
SS.ARbyte.S ← 1;
SS.ARbyte.TYPE ← 0011B; (* Read/Write, Accessed *)
SS.ARbyte.D ← 1; (* 32-bit stack segment*)
SS.ARbyte.DPL ← 0;
SS.SEL.RPL ← 0;
SS.ARbyte.P ← 1;

ESP ← SYSENTER_ESP_MSR;
EIP ← SYSENTER_EIP_MSR;
```

### IA-32e Mode Operation

In IA-32e mode, SYSENTER executes a fast system calls from user code running at privilege level 3 (in compatibility mode or 64-bit mode) to 64-bit executive procedures running at privilege level 0. This instruction is a companion instruction to the SYSEXIT instruction.

In IA-32e mode, the IA32\_SYSENTER\_EIP and IA32\_SYSENTER\_ESP MSRs hold 64-bit addresses and must be in canonical form; IA32\_SYSENTER\_CS must not contain a NULL selector.



When SYSENTER transfers control, the following fields are generated and bits set:

- **Target code segment** — Reads non-NULL selector from IA32\_SYSENTER\_CS.
- **New CS attributes** — L-bit = 1 (go to 64-bit mode); CS base = 0, CS limit = FFFFFFFFH.
- **Target instruction** — Reads 64-bit canonical address from IA32\_SYSENTER\_EIP.
- **Stack segment** — Computed by adding 8 to the value from IA32\_SYSENTER\_CS.
- **Stack pointer** — Reads 64-bit canonical address from IA32\_SYSENTER\_ESP.
- **New SS attributes** — SS base = 0, SS limit = FFFFFFFFH.

### Flags Affected

VM, IF, RF (see Operation above)

### Protected Mode Exceptions

#GP(0) If IA32\_SYSENTER\_CS[15:2] = 0.

### Real-Address Mode Exceptions

#GP(0) If protected mode is not enabled.

### Virtual-8086 Mode Exceptions

Same exceptions as in Protected Mode.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

Same exceptions as in Protected Mode.

## SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 35	SYSEXIT	Valid	Valid	Fast return to privilege level 3 user code.
REX.W + 0F 35	SYSEXIT	Valid	Valid	Fast return to 64-bit mode privilege level 3 user code.

### Description

Executes a fast return to privilege level 3 user code. SYSEXIT is a companion instruction to the SYSENTER instruction. The instruction is optimized to provide the maximum performance for returns from system procedures executing at protection levels 0 to user procedures executing at protection level 3. It must be executed from code executing at privilege level 0.

Prior to executing SYSEXIT, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- **IA32\_SYSENTER\_CS** — Contains a 32-bit value, of which the lower 16 bits are the segment selector for the privilege level 0 code segment in which the processor is currently executing. This value is used to compute the segment selectors for the privilege level 3 code and stack segments.
- **EDX** — Contains the 32-bit offset into the privilege level 3 code segment to the first instruction to be executed in the user code.
- **ECX** — Contains the 32-bit stack pointer for the privilege level 3 stack.

The IA32\_SYSENTER\_CS MSR can be read from and written to using RDMSR/WRMSR. The register address is listed in Table 4-6. This address is defined to remain fixed for future IA-32 processors.

When SYSEXIT is executed, the processor:

1. Adds 16 to the value in IA32\_SYSENTER\_CS and loads the sum into the CS selector register.
2. Loads the instruction pointer from the EDX register into the EIP register.
3. Adds 24 to the value in IA32\_SYSENTER\_CS and loads the sum into the SS selector register.
4. Loads the stack pointer from the ECX register into the ESP register.
5. Switches to privilege level 3.
6. Begins executing the user code at the EIP address.

See “SWAPGS—Swap GS Base Register” in this chapter for information about using the SYSENTER and SYSEXIT instructions as companion call and return instructions.

The SYSEXIT instruction always transfers program control to a protected-mode code segment with a DPL of 3. The instruction requires that the following conditions are met by the operating system:

- The segment descriptor for the selected user code segment selects a flat, 32-bit code segment of up to 4 GBytes, with execute, read, accessed, and non-conforming permissions.
- The segment descriptor for selected user stack segment selects a flat, 32-bit stack segment of up to 4 GBytes, with expand-up, read, write, and accessed permissions.

The SYSENTER can be invoked from all operating modes except real-address mode and virtual 8086 mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
  THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
    THEN
      SYSENTER/SYSEXIT_Not_Supported; FI;
    ELSE
      SYSENTER/SYSEXIT_Supported; FI;
  FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

## Operation

```
IF SYSENTER_CS_MSR[15:2] = 0 THEN #GP(0); FI;
IF CR0.PE = 0 THEN #GP(0); FI;
IF CPL ≠ 0 THEN #GP(0); FI;

CS.SEL ← (SYSENTER_CS_MSR + 16); (* Segment selector for return CS *)
(* Set rest of CS to a fixed value *)
CS.BASE ← 0; (* Flat segment *)
CS.LIMIT ← FFFFFFFH; (* 4-GByte limit *)
CS.ARbyte.G ← 1; (* 4-KByte granularity *)
CS.ARbyte.S ← 1;
CS.ARbyte.TYPE ← 1011B; (* Execute, Read, Non-Conforming Code *)
CS.ARbyte.D ← 1; (* 32-bit code segment *)
CS.ARbyte.DPL ← 3;
CS.SEL.RPL ← 3;
CS.ARbyte.P ← 1;
CPL ← 3;

SS.SEL ← (SYSENTER_CS_MSR + 24); (* Segment selector for return SS *)
(* Set rest of SS to a fixed value *);
```

## INSTRUCTION SET REFERENCE, N-Z

SS.BASE ← 0; (\* Flat segment \*)  
SS.LIMIT ← FFFFFFFH; (\* 4-GByte limit \*)  
SS.ARbyte.G ← 1; (\* 4-KByte granularity \*)  
SS.ARbyte.S ← ;  
SS.ARbyte.TYPE ← 0011B; (\* Expand Up, Read/Write, Data \*)  
SS.ARbyte.D ← 1; (\* 32-bit stack segment\*)  
SS.ARbyte.DPL ← 3;  
SS.SEL.RPL ← 3;  
SS.ARbyte.P ← 1;  
ESP ← ECX;  
EIP ← EDX;

### IA-32e Mode Operation

In IA-32e mode, SYSEXIT executes a fast system calls from a 64-bit executive procedures running at privilege level 0 to user code running at privilege level 3 (in compatibility mode or 64-bit mode). This instruction is a companion instruction to the SYSENTER instruction.

In IA-32e mode, the IA32\_SYSENTER\_EIP and IA32\_SYSENTER\_ESP MSRs hold 64-bit addresses and must be in canonical form; IA32\_SYSENTER\_CS must not contain a NULL selector.

When the SYSEXIT instruction transfers control to 64-bit mode user code using REX.W, the following fields are generated and bits set:

- **Target code segment** — Computed by adding 32 to the value in the IA32\_SYSENTER\_CS.
- **New CS attributes** — L-bit = 1 (go to 64-bit mode).
- **Target instruction** — Reads 64-bit canonical address in RDX.
- **Stack segment** — Computed by adding 8 to the value of CS selector.
- **Stack pointer** — Update RSP using 64-bit canonical address in RCX.

When SYSEXIT transfers control to compatibility mode user code when the operand size attribute is 32 bits, the following fields are generated and bits set:

- **Target code segment** — Computed by adding 16 to the value in IA32\_SYSENTER\_CS.
- **New CS attributes** — L-bit = 0 (go to compatibility mode).
- **Target instruction** — Fetch the target instruction from 32-bit address in EDX.
- **Stack segment** — Computed by adding 24 to the value in IA32\_SYSENTER\_CS.
- **Stack pointer** — Update ESP from 32-bit address in ECX.

### Flags Affected

None.

**Protected Mode Exceptions**

#GP(0)                    If IA32\_SYSENTER\_CS[15:2] = 0.  
                              If CPL ≠ 0.

**Real-Address Mode Exceptions**

#GP(0)                    If protected mode is not enabled.

**Virtual-8086 Mode Exceptions**

#GP(0)                    Always

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#GP(0)                    If IA32\_SYSENTER\_CS = 0.  
                              If CPL ≠ 0.  
                              If ECX or EDX contains a non-canonical address.

## SYSRET—Return From Fast System Call

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 07	SYSRET	Valid	Invalid	Return from fast system call

### Description

SYSCALL saves the RIP of the instruction following the SYSCALL into RCX and loads the new RIP from the LSTAR (64-bit mode only). Upon return, SYSRET copies the value saved in RCX to the RIP.

In a return to 64-bit mode using Osize 64, SYSRET sets the CS selector value to MSR IA32\_STAR[63:48] + 16. The SS is set to IA32\_STAR[63:48] + 8.

SYSRET transfer control to compatibility mode using Osize 32. The CS selector value is set to MSR IA32\_STAR[63:48]. The SS is set to IA32\_STAR[63:48] + 8.

It is the responsibility of the OS to keep descriptors in the GDT/LDT that correspond to selectors loaded by SYSCALL/SYSRET consistent with the base, limit and attribute values forced by the these instructions.

Software should not alter the CS or SS descriptors in a manner that violates the following assumptions made by SYSCALL/SYSRET:

- CS and SS base and limit remain the same for all processes, including the operating system.
- CS of the SYSCALL target has a privilege level of 0.
- CS of the SYSRET target has a privilege level of 3.

SYSCALL/SYSRET do not check for violations of these assumptions.

### Operation

```

IF (CS.L ≠ 1 ) or (IA32_EFER.SCE ≠ 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD; FI;
IF (CPL ≠ 0)
    THEN #GP(0); FI;
IF (RCX ≠ CANONICAL_ADDRESS)
    THEN #GP(0); FI;
IF (OPERAND_SIZE = 64)
    THEN (* Return to 64-Bit Mode *)
        EFLAGS ← R11;
        CPL ← 0x3;
        CS(SEL) ← IA32_STAR[63:48] + 16;
        CS(PL) ← 0x3;
        SS(SEL) ← IA32_STAR[63:48] + 8;
  
```

```

    SS(PL) ← 0x3;
    RIP ← RCX;
ELSE (* Return to Compatibility Mode *)
    EFLAGS ← R11;
    CPL ← 0x3;
    CS(SEL) ← IA32_STAR[63:48] ;
    CS(PL) ← 0x3;
    SS(SEL) ← IA32_STAR[63:48] + 8;
    SS(PL) ← 0x3;
    EIP ← ECX;

```

FI;

### Flags Affected

VM, IF, RF.

### Protected Mode Exceptions

#UD                    If Mode ≠ 64-Bit.

### Real-Address Mode Exceptions

#UD                    Instruction not recognized in this mode.

### Virtual-8086 Mode Exceptions

#UD                    Instruction not recognized in this mode.

### Compatibility Mode Exceptions

#UD                    Instruction not recognized in this mode.

### 64-Bit Mode Exceptions

#UD                    If IA32\_EFER.SCE bit = 0.

#GP(0)                If CPL ≠ 0.

                      If ECX contains a non-canonical address.

## TEST—Logical Compare

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
A8 <i>ib</i>	TEST AL, <i>imm8</i>	Valid	Valid	AND <i>imm8</i> with AL; set SF, ZF, PF according to result.
A9 <i>iw</i>	TEST AX, <i>imm16</i>	Valid	Valid	AND <i>imm16</i> with AX; set SF, ZF, PF according to result.
A9 <i>id</i>	TEST EAX, <i>imm32</i>	Valid	Valid	AND <i>imm32</i> with EAX; set SF, ZF, PF according to result.
REX.W + A9 <i>id</i>	TEST RAX, <i>imm32</i>	Valid	N.E.	AND <i>imm32</i> sign-extended to 64-bits with RAX; set SF, ZF, PF according to result.
F6 /0 <i>ib</i>	TEST <i>r/m8</i> , <i>imm8</i>	Valid	Valid	AND <i>imm8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
REX + F6 /0 <i>ib</i>	TEST <i>r/m8*</i> , <i>imm8</i>	Valid	N.E.	AND <i>imm8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
F7 /0 <i>iw</i>	TEST <i>r/m16</i> , <i>imm16</i>	Valid	Valid	AND <i>imm16</i> with <i>r/m16</i> ; set SF, ZF, PF according to result.
F7 /0 <i>id</i>	TEST <i>r/m32</i> , <i>imm32</i>	Valid	Valid	AND <i>imm32</i> with <i>r/m32</i> ; set SF, ZF, PF according to result.
REX.W + F7 /0 <i>id</i>	TEST <i>r/m64</i> , <i>imm32</i>	Valid	N.E.	AND <i>imm32</i> sign-extended to 64-bits with <i>r/m64</i> ; set SF, ZF, PF according to result.
84 / <i>r</i>	TEST <i>r/m8</i> , <i>r8</i>	Valid	Valid	AND <i>r8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
REX + 84 / <i>r</i>	TEST <i>r/m8*</i> , <i>r8*</i>	Valid	N.E.	AND <i>r8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
85 / <i>r</i>	TEST <i>r/m16</i> , <i>r16</i>	Valid	Valid	AND <i>r16</i> with <i>r/m16</i> ; set SF, ZF, PF according to result.
85 / <i>r</i>	TEST <i>r/m32</i> , <i>r32</i>	Valid	Valid	AND <i>r32</i> with <i>r/m32</i> ; set SF, ZF, PF according to result.
REX.W + 85 / <i>r</i>	TEST <i>r/m64</i> , <i>r64</i>	Valid	N.E.	AND <i>r64</i> with <i>r/m64</i> ; set SF, ZF, PF according to result.

**NOTES:**

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.

**Description**

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

In 64-bit mode, using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using an REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.



**Operation**

TEMP ← SRC1 AND SRC2;

SF ← MSB(TEMP);

IF TEMP = 0

    THEN ZF ← 1;

    ELSE ZF ← 0;

FI:

PF ← BitwiseXNOR(TEMP[0:7]);

CF ← 0;

OF ← 0;

(\* AF is undefined \*)

**Flags Affected**

The OF and CF flags are set to 0. The SF, ZF, and PF flags are set according to the result (see the “Operation” section above). The state of the AF flag is undefined.

**Protected Mode Exceptions**

#GP(0)            If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

                  If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0)            If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)   If a page fault occurs.

#AC(0)            If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP                If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS                If a memory operand effective address is outside the SS segment limit.

**Virtual-8086 Mode Exceptions**

#GP(0)            If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0)            If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)   If a page fault occurs.

#AC(0)            If alignment checking is enabled and an unaligned memory reference is made.

## INSTRUCTION SET REFERENCE, N-Z

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2E /r	UCOMISD <i>xmm1</i> , <i>xmm2/m64</i>	Valid	Valid	Compares (unordered) the low double-precision floating-point values in <i>xmm1</i> and <i>xmm2/m64</i> and set the EFLAGS accordingly.

### Description

Performs an unordered compare of the double-precision floating-point values in the low quadwords of source operand 1 (first operand) and source operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Source operand 1 is an XMM register; source operand 2 can be an XMM register or a 64 bit memory location.

The UCOMISD instruction differs from the COMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid operation exception if a source operand is either a QNaN or an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```

RESULT ← UnorderedCompare(SRC1[63:0] <> SRC2[63:0]) {
(* Set EFLAGS *)
CASE (RESULT) OF
    UNORDERED:      ZF, PF, CF ← 111;
    GREATER_THAN:   ZF, PF, CF ← 000;
    LESS_THAN:      ZF, PF, CF ← 001;
    EQUAL:          ZF, PF, CF ← 100;
ESAC;
OF, AF, SF ← 0;

```

## INSTRUCTION SET REFERENCE, N-Z

### Intel C/C++ Compiler Intrinsic Equivalent

`int_mm_ucomieq_sd(__m128d a, __m128d b)`  
`int_mm_ucomilt_sd(__m128d a, __m128d b)`  
`int_mm_ucomile_sd(__m128d a, __m128d b)`  
`int_mm_ucomigt_sd(__m128d a, __m128d b)`  
`int_mm_ucomige_sd(__m128d a, __m128d b)`  
`int_mm_ucomineq_sd(__m128d a, __m128d b)`

### SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.

If CR0.EM[bit 2] = 1.

If CR4.OSFXSR[bit 9] = 0.

If CPUID.01H:EDX.SSE2[bit 26] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#PF(fault-code) For a page fault.

#NM If CR0.TS[bit 3] = 1.

#XM If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

#UD If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.

If CR0.EM[bit 2] = 1.

If CR4.OSFXSR[bit 9] = 0.

If CPUID.01H:EDX.SSE2[bit 26] = 0.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 2E /r	UCOMISS <i>xmm1</i> , <i>xmm2/m32</i>	Valid	Valid	Compare lower single-precision floating-point value in <i>xmm1</i> register with lower single-precision floating-point value in <i>xmm2/mem</i> and set the status flags accordingly.

### Description

Performs an unordered compare of the single-precision floating-point values in the low double-words of the source operand 1 (first operand) and the source operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). In The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Source operand 1 is an XMM register; source operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals an SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISS instruction signals an invalid operation exception if a source operand is either a QNaN or an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

RESULT ← UnorderedCompare(SRC1[63:0] <> SRC2[63:0]) {

(\* Set EFLAGS \*)

CASE (RESULT) OF

UNORDERED: ZF,PF,CF ← 111;

GREATER\_THAN: ZF,PF,CF ← 000;

LESS\_THAN: ZF,PF,CF ← 001;

EQUAL: ZF,PF,CF ← 100;

ESAC;

OF,AF,SF ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

`int_mm_ucomieq_ss(__m128 a, __m128 b)`

`int_mm_ucomilt_ss(__m128 a, __m128 b)`

`int_mm_ucomile_ss(__m128 a, __m128 b)`

`int_mm_ucomigt_ss(__m128 a, __m128 b)`

`int_mm_ucomige_ss(__m128 a, __m128 b)`

`int_mm_ucomineq_ss(__m128 a, __m128 b)`

**SIMD Floating-Point Exceptions**

Invalid (if SNaN operands), Denormal.

**Protected Mode Exceptions**

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#XM	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.
#UD	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.

## INSTRUCTION SET REFERENCE, N-Z

If CR0.EM[bit 2] = 1.

If CR4.OSFXSR[bit 9] = 0.

If CPUID.01H:EDX.SSE[bit 25] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#PF(fault-code) For a page fault.

#NM If CR0.TS[bit 3] = 1.

#XM If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

#UD If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.

If CR0.EM[bit 2] = 1.

If CR4.OSFXSR[bit 9] = 0.

If CPUID.01H:EDX.SSE[bit 25] = 0.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## UD2—Undefined Instruction

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 0B	UD2	Valid	Valid	Raise invalid opcode exception.

### Description

Generates an invalid opcode. This instruction is provided for software testing to explicitly generate an invalid opcode. The opcode for this instruction is reserved for this purpose.

Other than raising the invalid opcode exception, this instruction is the same as the NOP instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

#UD (\* Generates invalid opcode exception \*);

### Flags Affected

None.

### Exceptions (All Operating Modes)

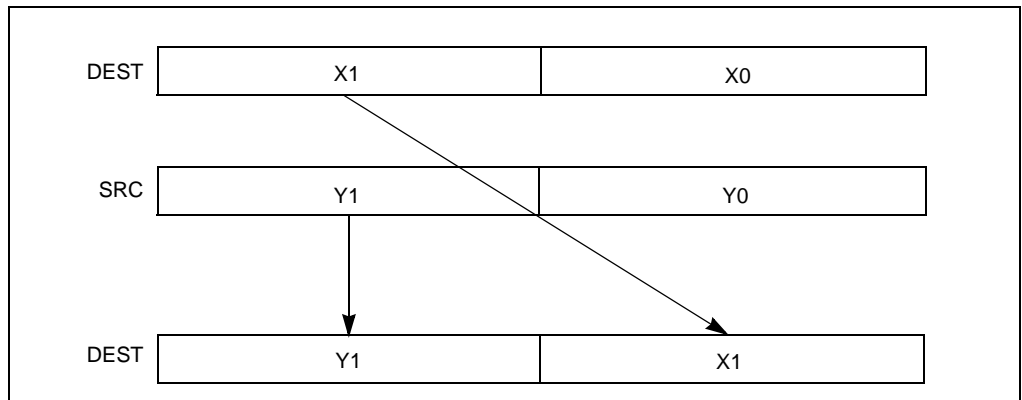
#UD                    Instruction is guaranteed to raise an invalid opcode exception in all operating modes.

## UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 15 /r	UNPCKHPD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Unpacks and Interleaves double-precision floating-point values from high quadwords of <i>xmm1</i> and <i>xmm2/m128</i> .

### Description

Performs an interleaved unpack of the high double-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-14. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.



**Figure 4-14. UNPCKHPD Instruction High Unpack and Interleave Operation**

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Operation**

DEST[63:0] ← DEST[127:64];  
 DEST[127:64] ← SRC[127:64];

**Intel C/C++ Compiler Intrinsic Equivalent**

UNPCKHPD    \_\_m128d \_mm\_unpackhi\_pd(\_\_m128d a, \_\_m128d b)

**SIMD Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.

**Real-Address Mode Exceptions**

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

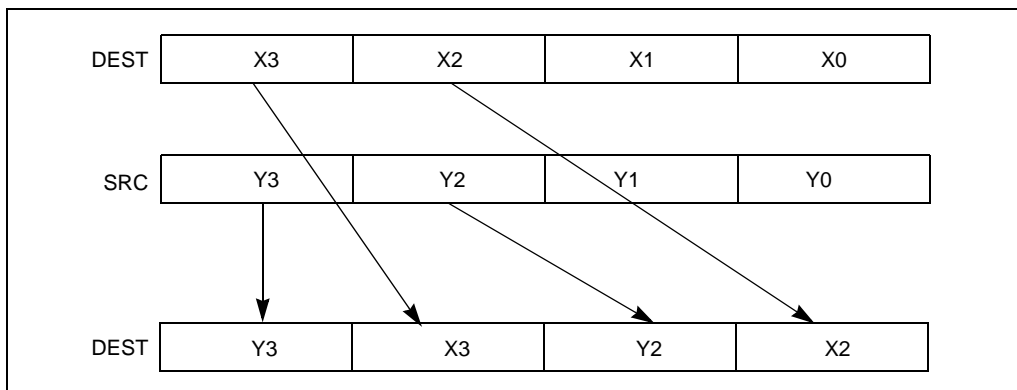
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.

## UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 15 /r	UNPCKHPS <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Unpacks and Interleaves single-precision floating-point values from high quadwords of <i>xmm1</i> and <i>xmm2/mem</i> into <i>xmm1</i> .

### Description

Performs an interleaved unpack of the high-order single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-15. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.



**Figure 4-15. UNPCKHPS Instruction High Unpack and Interleave Operation**

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```

DEST[31:0] ← DEST[95:64];
DEST[63:32] ← SRC[95:64];
DEST[95:64] ← DEST[127:96];
DEST[127:96] ← SRC[127:96];

```

## INSTRUCTION SET REFERENCE, N-Z

### Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPS \_\_m128 \_mm\_unpackhi\_ps(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE[bit 25] = 0.

### Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE[bit 25] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

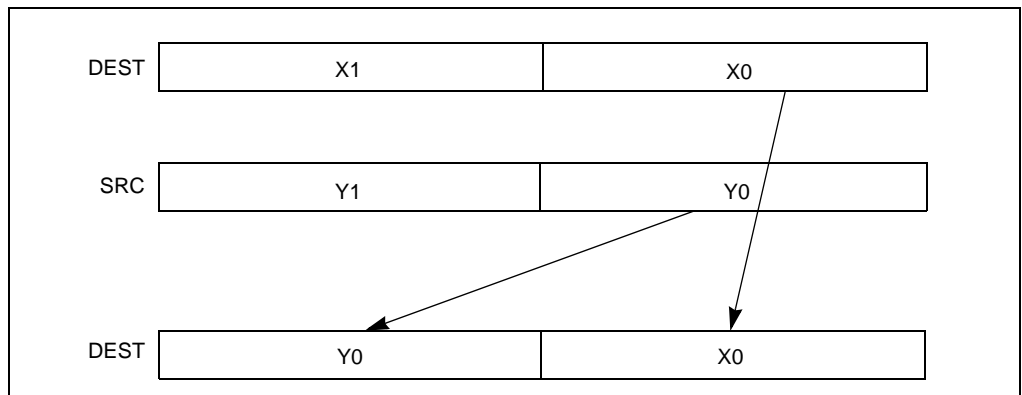
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

## UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 14 <i>/r</i>	UNPCKLPD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Unpacks and Interleaves double-precision floating-point values from low quadwords of <i>xmm1</i> and <i>xmm2/m128</i> .

### Description

Performs an interleaved unpack of the low double-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-16. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.



**Figure 4-16. UNPCKLPD Instruction Low Unpack and Interleave Operation**

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[63:0] ← DEST[63:0];  
DEST[127:64] ← SRC[63:0];



**Intel C/C++ Compiler Intrinsic Equivalent**

UNPCKHPD    \_\_m128d \_\_mm\_unpacklo\_pd(\_\_m128d a, \_\_m128d b)

**SIMD Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.

**Real-Address Mode Exceptions**

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

## INSTRUCTION SET REFERENCE, N-Z

### 64-Bit Mode Exceptions

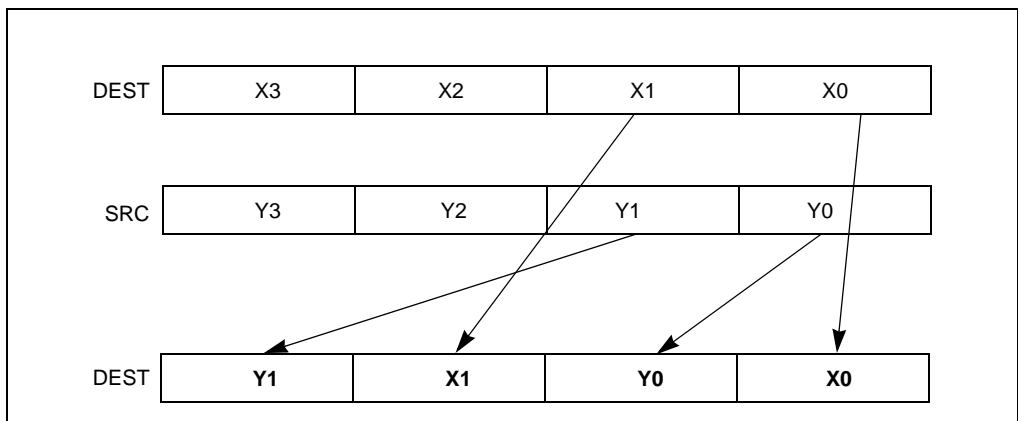
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.

## UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
0F 14 /r	UNPCKLPS <i>xmm1</i> , <i>xmm2/mem128</i>	Valid	Valid	Unpacks and interleaves single-precision floating-point values from low quadwords of <i>xmm1</i> and <i>xmm2/mem</i> into <i>xmm1</i> .

### Description

Performs an interleaved unpack of the low-order single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-17. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.



**Figure 4-17. UNPCKLPS Instruction Low Unpack and Interleave Operation**

When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## INSTRUCTION SET REFERENCE, N-Z

### Operation

DEST[31:0] ← DEST[31:0];  
DEST[63:32] ← SRC[31:0];  
DEST[95:64] ← DEST[63:32];  
DEST[127:96] ← SRC[63:32];

### Intel C/C++ Compiler Intrinsic Equivalent

UNPCKLPS    \_\_m128 \_mm\_unpacklo\_ps(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE[bit 25] = 0.

### Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.  If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE[bit 25] = 0.

**Virtual-8086 Mode Exceptions**

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#PF(fault-code) For a page fault.

#NM If CR0.TS[bit 3] = 1.

#UD If CR0.EM[bit 2] = 1.

If CR4.OSFXSR[bit 9] = 0.

If CPUID.01H:EDX.SSE[bit 25] = 0.

## VERR/VERW—Verify a Segment for Reading or Writing

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /4	VERR <i>r/m16</i>	Valid	Valid	Set ZF=1 if segment specified with <i>r/m16</i> can be read.
0F 00 /5	VERW <i>r/m16</i>	Valid	Valid	Set ZF=1 if segment specified with <i>r/m16</i> can be written.

### Description

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not NULL.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable.
- For the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment's DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector's RPL.

The validation performed is the same as is performed when a segment selector is loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) is performed. The segment selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. The operand size is fixed at 16 bits.

**Operation**

IF SRC(Offset) > (GDTR(Limit) or (LDTR(Limit))  
 THEN ZF ← 0; FI;

Read segment descriptor;

IF SegmentDescriptor(DescriptorType) = 0 (\* System segment \*)  
 or (SegmentDescriptor(Type) ≠ conforming code segment)  
 and (CPL > DPL) or (RPL > DPL)

THEN

ZF ← 0;

ELSE

IF ((Instruction = VERR) and (Segment readable))

or ((Instruction = VERW) and (Segment writable))

THEN

ZF ← 1;

FI;

FI;

**Flags Affected**

The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is set to 0.

**Protected Mode Exceptions**

The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#UD	The VERR and VERW instructions are not recognized in real-address mode.
-----	---

**Virtual-8086 Mode Exceptions**

#UD	The VERR and VERW instructions are not recognized in virtual-8086 mode.
-----	---

## INSTRUCTION SET REFERENCE, N-Z

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## WAIT/FWAIT—Wait

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B	WAIT	Valid	Valid	Check pending unmasked floating-point exceptions.
9B	FWAIT	Valid	Valid	Check pending unmasked floating-point exceptions.

### Description

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction insures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results. See the section titled "Floating-Point Exception Synchronization" in Chapter 8 of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*, for more information on using the WAIT/FWAIT instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

CheckForPendingUnmaskedFloatingPointExceptions;

### FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.

### Real-Address Mode Exceptions

#NM If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.

### Virtual-8086 Mode Exceptions

#NM If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.

## **INSTRUCTION SET REFERENCE, N-Z**

### **Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

### **64-Bit Mode Exceptions**

Same exceptions as in Protected Mode.

## WBINVD—Write Back and Invalidate Cache

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 09	WBINVD	Valid	Valid	Write back and flush Internal caches; initiate writing-back and flushing of external caches.

### Description

Writes back all modified cache lines in the processor's internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special-function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see "Serializing Instructions" in Chapter 8 of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A*).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future IA-32 processors. The instruction is not supported on IA-32 processors earlier than the Intel486 processor.

### Operation

```
WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue; (* Continue execution *)
```

### Flags Affected

None.

## INSTRUCTION SET REFERENCE, N-Z

### Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

#GP(0) The WBINVD instruction cannot be executed at the virtual-8086 mode.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

Same exceptions as in Protected Mode.

## WRMSR—Write to Model Specific Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 30	WRMSR	Valid	Valid	Write the value in EDX:EAX to MSR specified by ECX.
REX.W + 0F 30	WRMSR	Valid	N.E.	Write the value in RDX[31:0]: RAX[31:0] to MSR specified by RCX.

### Description

In legacy and compatibility mode, writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified by the ECX register. The value loaded into the ECX register is the address of the MSR. The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see “Translation Lookaside Buffers (TLBs)” in Chapter 3 of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3A*).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Appendix B, “Model-Specific Registers (MSRs)”, in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B*, lists all MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see “Serializing Instructions” in Chapter 7 of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3A*).

The CPUID instruction should be used to determine whether MSRs are supported (EDX[5]=1) before using this instruction.

In 64-bit mode, operation is the same as legacy mode, except that targeted registers are updated by MSR[63:32] = RDX[31:0], MSR[31:0] = RAX[31:0].

### IA-32 Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

## INSTRUCTION SET REFERENCE, N-Z

### Operation

IF 64-Bit Mode and REX.W used  
THEN  
    MSR[RCX] ← RDX:RAX;  
ELSE IF (Non-64-Bit Modes or Default 64-Bit Mode)  
    MSR[ECX] ← EDX:EAX; FI;  
FI;

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)            If the current privilege level is not 0.  
                  If the value in ECX specifies a reserved or unimplemented MSR address.  
                  If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.

### Real-Address Mode Exceptions

#GP(0)            If the value in ECX specifies a reserved or unimplemented MSR address.  
                  If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.

### Virtual-8086 Mode Exceptions

#GP(0)            The WRMSR instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

Same exceptions as in Protected Mode.

## XADD—Exchange and Add

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F C0 /r	XADD r/m8, r8	Valid	Valid	Exchange r8 and r/m8; load sum into r/m8.
REX + 0F C0 /r	XADD r/m8*, r8*	Valid	N.E.	Exchange r8 and r/m8; load sum into r/m8.
0F C1 /r	XADD r/m16, r16	Valid	Valid	Exchange r16 and r/m16; load sum into r/m16.
0F C1 /r	XADD r/m32, r32	Valid	Valid	Exchange r32 and r/m32; load sum into r/m32.
REX.W + 0F C1 /r	XADD r/m64, r64	Valid	N.E.	Exchange r64 and r/m64; load sum into r/m64.

### NOTES:

- \* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.

### Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

In 64-bit mode, the instruction's default operation size is 32 bits. Using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using an REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

### IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

### Operation

```
TEMP ← SRC + DEST;
SRC ← DEST;
DEST ← TEMP;
```

### Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

## INSTRUCTION SET REFERENCE, N-Z

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## XCHG—Exchange Register/Memory with Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
90+rw	XCHG AX, <i>r16</i>	Valid	Valid	Exchange <i>r16</i> with AX.
90+rw	XCHG <i>r16</i> , AX	Valid	Valid	Exchange AX with <i>r16</i> .
90+rd	XCHG EAX, <i>r32</i>	Valid	Valid	Exchange <i>r32</i> with EAX.
REX.W + 90+rd	XCHG RAX, <i>r64</i>	Valid	N.E.	Exchange <i>r64</i> with RAX.
90+rd	XCHG <i>r32</i> , EAX	Valid	Valid	Exchange EAX with <i>r32</i> .
REX.W + 90+rd	XCHG <i>r64</i> , RAX	Valid	N.E.	Exchange RAX with <i>r64</i> .
86 /r	XCHG <i>r/m8</i> , <i>r8</i>	Valid	Valid	Exchange <i>r8</i> (byte register) with byte from <i>r/m8</i> .
REX + 86 /r	XCHG <i>r/m8*</i> , <i>r8*</i>	Valid	N.E.	Exchange <i>r8</i> (byte register) with byte from <i>r/m8</i> .
86 /r	XCHG <i>r8</i> , <i>r/m8</i>	Valid	Valid	Exchange byte from <i>r/m8</i> with <i>r8</i> (byte register).
REX + 86 /r	XCHG <i>r8*</i> , <i>r/m8*</i>	Valid	N.E.	Exchange byte from <i>r/m8</i> with <i>r8</i> (byte register).
87 /r	XCHG <i>r/m16</i> , <i>r16</i>	Valid	Valid	Exchange <i>r16</i> with word from <i>r/m16</i> .
87 /r	XCHG <i>r16</i> , <i>r/m16</i>	Valid	Valid	Exchange word from <i>r/m16</i> with <i>r16</i> .
87 /r	XCHG <i>r/m32</i> , <i>r32</i>	Valid	Valid	Exchange <i>r32</i> with doubleword from <i>r/m32</i> .
REX.W + 87 /r	XCHG <i>r/m64</i> , <i>r64</i>	Valid	N.E.	Exchange <i>r64</i> with quadword from <i>r/m64</i> .
87 /r	XCHG <i>r32</i> , <i>r/m32</i>	Valid	Valid	Exchange doubleword from <i>r/m32</i> with <i>r32</i> .
REX.W + 87 /r	XCHG <i>r64</i> , <i>r/m64</i>	Valid	N.E.	Exchange quadword from <i>r/m64</i> with <i>r64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.

### Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 7 of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A*, for more information on bus locking.)

## INSTRUCTION SET REFERENCE, N-Z

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

In 64-bit mode, the instruction's default operation size is 32 bits. Using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using an REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```
TEMP ← DEST;  
DEST ← SRC;  
SRC ← TEMP;
```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If either operand is in a non-writable segment.  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D7	XLAT <i>m8</i>	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
D7	XLATB	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
REX.W + D7	XLATB	Valid	N.E.	Set AL to memory byte [RBX + unsigned AL].

### Description

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as an unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from either the DS:EBX or the DS:BX registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.)

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operand” form and the “no-operand” form. The explicit-operand form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS:(E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operands form (XLATB) provides a “short form” of the XLAT instructions. Here also the processor assumes that the DS:(E)BX registers contain the base address of the table.

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table’s base address. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```
IF AddressSize = 16
  THEN
    AL ← (DS:BX + ZeroExtend(AL));
  ELSE IF (AddressSize = 32)
    AL ← (DS:EBX + ZeroExtend(AL)); FI;
  ELSE (AddressSize = 64)
    AL ← (RBX + ZeroExtend(AL));
  FI;
```

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.

## XOR—Logical Exclusive OR

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
34 <i>ib</i>	XOR AL, <i>imm8</i>	Valid	Valid	AL XOR <i>imm8</i> .
35 <i>iw</i>	XOR AX, <i>imm16</i>	Valid	Valid	AX XOR <i>imm16</i> .
35 <i>id</i>	XOR EAX, <i>imm32</i>	Valid	Valid	EAX XOR <i>imm32</i> .
REX.W + 35 <i>id</i>	XOR RAX, <i>imm32</i>	Valid	N.E.	RAX XOR <i>imm32</i> ( <i>sign-extended</i> ).
80 /6 <i>ib</i>	XOR <i>r/m8</i> , <i>imm8</i>	Valid	Valid	<i>r/m8</i> XOR <i>imm8</i> .
REX + 80 /6 <i>ib</i>	XOR <i>r/m8*</i> , <i>imm8</i>	Valid	N.E.	<i>r/m8</i> XOR <i>imm8</i> .
81 /6 <i>iw</i>	XOR <i>r/m16</i> , <i>imm16</i>	Valid	Valid	<i>r/m16</i> XOR <i>imm16</i> .
81 /6 <i>id</i>	XOR <i>r/m32</i> , <i>imm32</i>	Valid	Valid	<i>r/m32</i> XOR <i>imm32</i> .
REX.W + 81 /6 <i>id</i>	XOR <i>r/m64</i> , <i>imm32</i>	Valid	N.E.	<i>r/m64</i> XOR <i>imm32</i> ( <i>sign-extended</i> ).
83 /6 <i>ib</i>	XOR <i>r/m16</i> , <i>imm8</i>	Valid	Valid	<i>r/m16</i> XOR <i>imm8</i> ( <i>sign-extended</i> ).
83 /6 <i>ib</i>	XOR <i>r/m32</i> , <i>imm8</i>	Valid	Valid	<i>r/m32</i> XOR <i>imm8</i> ( <i>sign-extended</i> ).
REX.W + 83 /6 <i>ib</i>	XOR <i>r/m64</i> , <i>imm8</i>	Valid	N.E.	<i>r/m64</i> XOR <i>imm8</i> ( <i>sign-extended</i> ).
30 /r	XOR <i>r/m8</i> , <i>r8</i>	Valid	Valid	<i>r/m8</i> XOR <i>r8</i> .
REX + 30 /r	XOR <i>r/m8*</i> , <i>r8*</i>	Valid	N.E.	<i>r/m8</i> XOR <i>r8</i> .
31 /r	XOR <i>r/m16</i> , <i>r16</i>	Valid	Valid	<i>r/m16</i> XOR <i>r16</i> .
31 /r	XOR <i>r/m32</i> , <i>r32</i>	Valid	Valid	<i>r/m32</i> XOR <i>r32</i> .
REX.W + 31 /r	XOR <i>r/m64</i> , <i>r64</i>	Valid	N.E.	<i>r/m64</i> XOR <i>r64</i> .
32 /r	XOR <i>r8</i> , <i>r/m8</i>	Valid	Valid	<i>r8</i> XOR <i>r/m8</i> .
REX + 32 /r	XOR <i>r8*</i> , <i>r/m8*</i>	Valid	N.E.	<i>r8</i> XOR <i>r/m8</i> .
33 /r	XOR <i>r16</i> , <i>r/m16</i>	Valid	Valid	<i>r16</i> XOR <i>r/m16</i> .
33 /r	XOR <i>r32</i> , <i>r/m32</i>	Valid	Valid	<i>r32</i> XOR <i>r/m32</i> .
REX.W + 33 /r	XOR <i>r64</i> , <i>r/m64</i>	Valid	N.E.	<i>r64</i> XOR <i>r/m64</i> .

**NOTES:**

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.

**Description**

Performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, using an REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using an REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

DEST ← DEST XOR SRC;

### Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

### Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## INSTRUCTION SET REFERENCE, N-Z

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



## XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 57 /r	XORPD <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Bitwise exclusive-OR of <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a bitwise logical exclusive-OR of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[127:0] ← DEST[127:0] BitwiseXOR SRC[127:0];

### Intel C/C++ Compiler Intrinsic Equivalent

XORPD            \_\_m128d \_mm\_xor\_pd(\_\_m128d a, \_\_m128d b)

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE2[bit 26] = 0.

## INSTRUCTION SET REFERENCE, N-Z

### Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE2[bit 26] = 0.

## XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 57 /r	XORPS <i>xmm1</i> , <i>xmm2/m128</i>	Valid	Valid	Bitwise exclusive-OR of <i>xmm2/m128</i> and <i>xmm1</i> .

### Description

Performs a bitwise logical exclusive-OR of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[127:0] ← DEST[127:0] BitwiseXOR SRC[127:0];

### Intel C/C++ Compiler Intrinsic Equivalent

XORPS            \_\_m128 \_mm\_xor\_ps(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.  If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.  If CR4.OSFXSR[bit 9] = 0.  If CPUID.01H:EDX.SSE[bit 25] = 0.

## INSTRUCTION SET REFERENCE, N-Z

### Real-Address Mode Exceptions

#GP(0)	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

### Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
-----------------	-------------------

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0. If CPUID.01H:EDX.SSE[bit 25] = 0.

# 5

## **VMX Instruction Reference title**



# CHAPTER 5

## VMX INSTRUCTION REFERENCE

### 5.1 OVERVIEW

This chapter describes the IA-32 instructions that support the virtual-machine extensions (VMX). VMX is intended to support virtualization of processor hardware and a system software layer acting as a host to multiple guest software environments. The virtual-machine extensions (VMX) includes five instructions that manage the virtual-machine control structure (VMCS) and five instruction that manage VMX operation. Additional details of VMX are described in *IA-32 Intel Architecture Software Developer's Manual, Volume 3B*.

The behavior of the VMCS-maintenance instructions is summarized below:

- **VMPTRLD** — This instruction takes a single 64-bit source operand that is in memory. It makes the referenced VMCS active and current, loading the current-VMCS pointer with this operand and establishes the current VMCS based on the contents of VMCS-data area in the referenced VMCS region. Because this makes the referenced VMCS active, a logical processor may start maintaining on the processor some of the VMCS data for the VMCS.
- **VMPTRST** — This instruction takes a single 64-bit destination operand that is in memory. The current-VMCS pointer is stored into the destination operand.
- **VMCLEAR** — This instruction takes a single 64-bit operand that is in memory. The instruction sets the launch state of the VMCS referenced by the operand to “clear”, renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCS-data area in the referenced VMCS region. If the operand is the same as the current-VMCS pointer, that pointer is made invalid.
- **VMREAD** — This instruction reads a component from the VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand that may be a register or in memory.
- **VMWRITE** — This instruction writes a component to the VMCS (the encoding of that field is given in a register operand) from a source operand that may be a register or in memory.

The behavior of the VMX management instructions is summarized below:

- **VMCALL** — This instruction allows a guest in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.
- **VMLAUNCH** — This instruction launches a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
- **VMRESUME** — This instruction resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
- **VMXOFF** — This instruction causes the processor to leave VMX operation.

## VMX INSTRUCTION REFERENCE

- **VMXON** — This instruction takes a single 64-bit source operand that is in memory. It causes a logical processor to enter VMX root operation and to use the memory referenced by the operand to support VMX operation.

Only VMCALL can be executed in compatibility mode (causing a VM exit). The other VMX instructions generate invalid-opcode exceptions if executed in compatibility mode.

## 5.2 CONVENTIONS

The operation sections for the VMX instructions in Section 5.3 use the pseudo-function VMexit, which indicates that the logical processor performs a VM exit.

The operation sections also use the pseudo-functions VMsucceed, VMfail, VMfailInvalid, and VMfailValid. These pseudo-functions signal instruction success or failure by setting or clearing bits in RFLAGS and, in some cases, by writing the VM-instruction error field. The following pseudocode fragments detail these functions:

VMsucceed:

```
CF ← 0;
PF ← 0;
AF ← 0;
ZF ← 0;
SF ← 0;
OF ← 0;
```

VMfail(ErrorNumber):

```
IF VMCS pointer is valid
    THEN VMfailValid(ErrorNumber);
    ELSE VMfailInvalid;
FI;
```

VMfailInvalid:

```
CF ← 1;
PF ← 0;
AF ← 0;
ZF ← 0;
SF ← 0;
OF ← 0;
```

VMfailValid(ErrorNumber):// executed only if there is a current VMCS

```
CF ← 0;
PF ← 0;
AF ← 0;
ZF ← 1;
SF ← 0;
OF ← 0;
Set the VM-instruction error field to ErrorNumber;
```

The different VM-instruction error numbers are enumerated in Appendix J, “VM Instruction Error Numbers,” in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B*.



## 5.3 VMX INSTRUCTIONS

This section provides detailed descriptions of the VMX instructions.

## VMCALL—Call to VM Monitor

Opcode	Instruction	Description
0F 01 C1	VMCALL	Call to VM monitor by causing VM exit.

### Description

This instruction allows guest software can make a call for service into an underlying VM monitor. The details of the programming interface for such calls are VMM-specific; this instruction does nothing more than cause a VM exit, registering the appropriate exit reason.

Use of this instruction in VMX root operation invokes an SMM monitor (see Section 24.16.2 in *IA-32 Intel Architecture Software Developer’s Manual, Volume 3B*). This invocation will activate the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM) if it is not already active (see Section 24.16.6 in *IA-32 Intel Architecture Software Developer’s Manual, Volume 3B*).

### Operation

```

IF not in VMX operation
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF in SMM or if the valid bit in the IA32_SMM_MONITOR_CTL MSR is clear
    THEN VMfail(VMCALL executed in VMX root operation);
ELSIF (RFLAGS.VM = 1) OR (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF dual-monitor treatment of SMIs and SMM is active
    THEN perform an SMM VM exit (see Section 24.16.2
        of the IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF launch state of current VMCS is not clear
    THEN VMfailValid(VMCALL with non-clear VMCS);
ELSIF VM-exit control fields pertinent to saving state are not valid1
    THEN VMfailValid(VMCALL with invalid VM-exit control fields);
ELSE
    enter SMM;
    read revision identifier in MSEG;
    IF revision identifier does not match that supported by processor
        THEN

```

---

1. This includes the “save” VM-exit controls and the VM-exit MSR-store address and count fields.

```

    leave SMM;
    VMfailValid(VMCALL with incorrect MSEG revision identifier);
ELSE
    read SMM-monitor features field, MSEG (see Section 24.16.6.2,
    in the IA-32 Intel® Architecture Software Developer's Manual, Volume 3B);
    IF features field is invalid
        THEN
            leave SMM;
            VMfailValid(VMCALL with invalid SMM-monitor features);
        ELSE activate dual-monitor treatment of SMIs and SMM (see Section 24.16.6
        in the IA-32 Intel® Architecture Software Developer's Manual, Volume 3B);
FI;
FI;
FI;
```

### Flags Affected

See the operation section and Section 5.2.

### Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPZ and REP/REPE/REPZ)
Segment overrides	Ignored
Operand size	Causes #UD
Address size	Ignored
REX	Ignored

### Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the logical processor is in VMX root operation.
#UD	If executed outside VMX operation.

### Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the VMCALL instruction is not recognized outside VMX operation.
-----	---

### Virtual-8086 Mode Exceptions

#UD	If executed outside VMX non-root operation.
-----	---

## VMX INSTRUCTION REFERENCE

### Compatibility Mode Exceptions

#UD                      If executed outside VMX non-root operation.

### 64-Bit Mode Exceptions

#UD                      If executed outside VMX operation.

## VMCLEAR—Clear Virtual-Machine Control Structure

Opcode	Instruction	Description
66 0F C7 /6	VMCLEAR m64	Copy VMCS data to VMCS region in memory.

### Description

This instruction applies to the VMCS whose VMCS region resides at the physical address contained in the instruction operand. The instruction ensures that VMCS data for that VMCS (some of these data may be currently maintained on the processor) are copied to the VMCS region in memory. It also initializes parts of the VMCS region (for example, it sets the launch state of that VMCS to clear). See Chapter 20, “Virtual-Machine Control Structures,” in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B*.

The operand of this instruction is always 64 bits and is always in memory. If the operand is the current-VMCS pointer, then that pointer is made invalid (set to FFFFFFFF\_FFFFFFFFH).

Note that the VMCLEAR instruction might not explicitly write any VMCS data to memory; the data may be already resident in memory before the VMCLEAR is executed.

### Operation

```

IF (register operand) or (not in VMX operation) or (RFLAGS.VM = 1) or
(IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF CPL > 0
    THEN #GP(0);
ELSE
    addr ← contents of 64-bit in-memory operand;
    IF addr is not 4KB-aligned OR
    (processor supports Intel EM64T and
    addr sets any bits beyond the physical-address width) OR
    (processor does not support Intel EM64T and addr sets any bits in the range 63:32)
        THEN VMfail(VMCLEAR with invalid physical address);
    ELSIF addr = VMXON pointer
        THEN VMfail(VMCLEAR with VMXON pointer);
    ELSE
        ensure that data for VMCS referenced by the operand is in memory;
        initialize implementation-specific data in VMCS region;
        launch state of VMCS referenced by the operand ← “clear”
        IF operand addr = current-VMCS pointer
            THEN current-VMCS pointer ← FFFFFFFF_FFFFFFFFH;
        FI;
        VMsucceed;
    
```

## VMX INSTRUCTION REFERENCE

FI;  
FI;

### Flags Affected

See the operation section and Section 5.2.

### Use of Prefixes

LOCK	Causes #UD.
REP*	Reserved and may cause unpredictable behavior (applies to both REPNE/REPZ and REP/REPE/REPZ).
Segment overrides	Treated normally
Operand size	Ignored
Address size	Treated normally
REX	Register extensions treated normally; operand-size overrides ignored

### Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.  If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  If the DS, ES, FS, or GS register contains an unusable segment.  If the operand is located in an execute-only code segment.
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory operand effective address is outside the SS segment limit.  If the SS register contains an unusable segment.
#UD	If operand is a register.  If not in VMX operation.

### Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the VMCLEAR instruction is not recognized outside VMX operation.
-----	--

### Virtual-8086 Mode Exceptions

#UD	The VMCLEAR instruction is not recognized in virtual-8086 mode.
-----	---

**Compatibility Mode Exceptions**

#UD                   The VMCLEAR instruction is not recognized in compatibility mode.

**64-Bit Mode Exceptions**

#GP(0)                If the current privilege level is not 0.  
                      If the source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.

#PF(fault-code)      If a page fault occurs in accessing the memory operand.

#SS(0)                If the source operand is in the SS segment and the memory address is in a non-canonical form.

#UD                    If operand is a register.  
                      If not in VMX operation.

## VMLAUNCH/VMRESUME—Launch/Resume Virtual Machine

Opcode	Instruction	Description
0F 01 C2	VMLAUNCH	Launch virtual machine managed by current VMCS.
0F 01 C3	VMRESUME	Resume virtual machine managed by current VMCS.

### Description

Effects a VM entry managed by the current VMCS.

- VMLAUNCH fails if the launch state of current VMCS is not “clear”. If the instruction is successful, it sets the launch state to “launched.”
- VMRESUME fails if the launch state of the current VMCS is not “launched.”

If VM entry is attempted, the logical processor performs a series of consistency checks as detailed in Chapter 22, “VM Entries,” in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B*. Failure to pass checks on the VMX controls or on the host-state area passes control to the instruction following the VMLAUNCH or VMRESUME instruction. If these pass but checks on the guest-state area fail, the logical processor loads state from the host-state area of the VMCS, passing control to the instruction referenced by the RIP field in the host-state area.

VM entry is not allowed when events are blocked by MOV SS or POP SS. Neither VMLAUNCH nor VMRESUME should be used immediately after either MOV to SS or POP to SS.

### Operation

```

IF (not in VMX operation) or (RFLAGS.VM = 1) or
(IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF events are being blocked by MOV SS
    THEN VMfailValid(VM entry with events blocked by MOV SS);
ELSIF (VMLAUNCH and launch state of current VMCS is not “clear”)
    THEN VMfailValid(VMLAUNCH with non-clear VMCS);
ELSIF (VMRESUME and launch state of current VMCS is not “launched”)
    THEN VMfailValid(VMRESUME with non-launched VMCS);
ELSE
    Check settings of VMX controls and host-state area;
    IF invalid settings
        THEN VMfailValid(VM entry with invalid VMX-control field(s)) or
    
```



VMfailValid(VM entry with invalid host-state field(s)) or  
 VMfailValid(VM entry with invalid executive-VMCS pointer) or  
 VMfailValid(VM entry with non-launched executive VMCS) or  
 VMfailValid(VM entry with executive-VMCS pointer not VMXON pointer)  
 or  
 VMfailValid(VM entry with invalid VM-execution control fields in executive  
 VMCS)  
 as appropriate;

ELSE

Attempt to load guest state and PDPTRs as appropriate;  
 clear address-range monitoring;

IF failure in checking guest state or PDPTRs

THEN VM entry fails (see Section 22.7, in the  
*IA-32 Intel® Architecture Software Developer's Manual, Volume 3B*);

ELSE

Attempt to load MSR from VM-entry MSR-load area;

IF failure

THEN VM entry fails (see Section 22.7, in the *IA-32 Intel®  
 Architecture Software Developer's Manual, Volume 3B*);

ELSE

IF VMLAUNCH

THEN launch state of VMCS ← “launched”;

FI;

IF in SMM and “entry to SMM” VM-entry control is 0

THEN

IF “deactivate dual-monitor treatment” VM-entry  
 control is 0

THEN SMM-transfer VMCS pointer ←  
 current-VMCS pointer;

FI;

IF executive-VMCS pointer is VMX pointer

THEN current-VMCS pointer ←  
 VMCS-link pointer;

ELSE current-VMCS pointer ←  
 executive-VMCS pointer;

FI;

leave SMM;

FI;

VM entry succeeds;

FI;

FI;

FI;

FI;

Further details of the operation of the VM-entry appear in Chapter 22 of *IA-32 Intel Architecture Software Developer's Manual, Volume 3B*.

## VMX INSTRUCTION REFERENCE

### Flags Affected

See the operation section and Section 5.2.

### Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ)
Segment overrides	Ignored
Operand size	Causes #UD
Address size	Ignored
REX	Ignored

### Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If executed outside VMX operation.

### Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the VMLAUNCH and VMRESUME instructions are not recognized outside VMX operation.
-----	--

### Virtual-8086 Mode Exceptions

#UD	The VMLAUNCH and VMRESUME instructions are not recognized in virtual-8086 mode.
-----	---

### Compatibility Mode Exceptions

#UD	The VMLAUNCH and VMRESUME instructions are not recognized in compatibility mode.
-----	--

### 64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If executed outside VMX operation.

## VMPTRLD—Load Pointer to Virtual-Machine Control Structure

Opcode	Instruction	Description
0F C7 /6	VMPTRLD m64	Loads the current VMCS pointer from memory.

### Description

Marks the current-VMCS pointer valid and loads it with the physical address in the instruction operand. The instruction fails if its operand is not properly aligned, sets unsupported physical-address bits, or is equal to the VMXON pointer. In addition, the instruction fails if the 32 bits in memory referenced by the operand do not match the VMCS revision identifier supported by this processor.<sup>2</sup>

The operand of this instruction is always 64 bits and is always in memory.

### Operation

```

IF (register operand) or (not in VMX operation) or (RFLAGS.VM = 1) or
(IA32_EFER.LMA = 1 and CS.L = 0)
  THEN #UD;
ELSIF in VMX non-root operation
  THEN VMexit;
ELSIF CPL > 0
  THEN #GP(0);
ELSE
  addr ← contents of 64-bit in-memory source operand;
  IF addr is not 4KB-aligned OR
  (processor supports Intel EM64T and
  addr sets any bits beyond the processor's physical-address width) OR
  processor does not support Intel EM64T and addr sets any bits in the range 63:32
    THEN VMfail(VMPTRLD with invalid physical address);
  ELSIF addr = VMXON pointer
    THEN VMfail(VMPTRLD with VMXON pointer);
  ELSE
    rev ← 32 bits located at physical address addr;
    IF rev ≠ VMCS revision identifier supported by processor
      THEN VMfail(VMPTRLD with incorrect VMCS revision identifier);
    ELSE
      current-VMCS pointer ← addr;
      VMSucceed;
    FI;
  FI;
FI;

```

2. Software should consult the VMX capability MSR VMX\_BASIC to discover the VMCS revision identifier supported by this processor (see Appendix G, “VMX Capability Reporting Facility,” in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3B*).

## VMX INSTRUCTION REFERENCE

### Flags Affected

See the operation section and Section 5.2.

### Use of Prefixes

LOCK	Causes #UD
REPNE/REPZ	Causes #UD
REP/REPE/REPZ	Changes encoding to that of VMXON; see “VMXON—Enter VMX Operation” for operation and interactions with other prefixes.
Segment overrides	Treated normally
Operand size	Changes encoding to that of VMCLEAR; see “VMCLEAR—Clear Virtual-Machine Control Structure” for operation and interactions with other prefixes.
Address size	Treated normally
REX	Register extensions treated normally; operand-size overrides ignored

### Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0. If the memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains an unusable segment. If the source operand is located in an execute-only code segment.
#PF(fault-code)	If a page fault occurs in accessing the memory source operand.
#SS(0)	If the memory source operand effective address is outside the SS segment limit. If the SS register contains an unusable segment.
#UD	If operand is a register. If not in VMX operation.

### Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the VMPTRLD instruction is not recognized outside VMX operation.
-----	--

### Virtual-8086 Mode Exceptions

#UD	The VMPTRLD instruction is not recognized in virtual-8086 mode.
-----	---

**Compatibility Mode Exceptions**

#UD                   The VMPTRLD instruction is not recognized in compatibility mode.

**64-Bit Mode Exceptions**

#GP(0)                If the current privilege level is not 0.  
                      If the source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.

#PF(fault-code)     If a page fault occurs in accessing the memory source operand.

#SS(0)                If the source operand is in the SS segment and the memory address is in a non-canonical form.

#UD                   If operand is a register.

                      If not in VMX operation.

## VMPTRST—Store Pointer to Virtual-Machine Control Structure

Opcode	Instruction	Description
0F C7 77	VMPTRST	Stores the current VMCS pointer into memory.

### Description

Stores the current-VMCS pointer into a specified memory address. The operand of this instruction is always 64 bits and is always in memory.

### Operation

```

IF (register operand) or (not in VMX operation) or (RFLAGS.VM = 1) or
(IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSE
    64-bit in-memory destination operand ← current-VMCS pointer;
    VMsucceed;
FI;
    
```

### Flags Affected

See the operation section and Section 5.2.

### Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ)
Segment overrides	Treated normally
Operand size	Causes #UD
Address size	Treated normally
REX	Register extensions treated normally; operand-size overrides ignored

### Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.  If the memory destination operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
--------	---

If the DS, ES, FS, or GS register contains an unusable segment.

If the destination operand is located in a read-only data segment or any code segment.

#PF(fault-code) If a page fault occurs in accessing the memory destination operand.

#SS(0) If the memory destination operand effective address is outside the SS segment limit.

If the SS register contains an unusable segment.

#UD If operand is a register.

If not in VMX operation.

### Real-Address Mode Exceptions

#UD A logical processor cannot be in real-address mode while in VMX operation and the VMPTRST instruction is not recognized outside VMX operation.

### Virtual-8086 Mode Exceptions

#UD The VMPTRST instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD The VMPTRST instruction is not recognized in compatibility mode.

### 64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0.

If the destination operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.

#PF(fault-code) If a page fault occurs in accessing the memory destination operand.

#SS(0) If the destination operand is in the SS segment and the memory address is in a non-canonical form.

#UD If operand is a register.

If not in VMX operation.

## VMREAD—Read Field from Virtual-Machine Control Structure

Opcode	Instruction	Description
0F 78	VMREAD r/m64, r64	Reads a specified VMCS field (in 64-bit mode).
0F 78	VMREAD r/m32, r32	Reads a specified VMCS field (outside 64-bit mode).

### Description

Reads a specified field from the VMCS and stores it into a specified destination operand (register or memory).

The specific VMCS field is identified by the VMCS-field encoding contained in the register source operand. Outside IA-32e mode, the source operand has 32 bits, regardless of the value of CS.D. In 64-bit mode, the source operand has 64 bits; however, if bits 63:32 of the source operand are not zero, VMREAD will fail due to an attempt to access an unsupported VMCS component (see operation section).

The effective size of the primary source operand, which may be a register or in memory, is always 32 bits outside IA-32e mode (the setting of CS.D is ignored with respect to operand size) and 64 bits in 64-bit mode. If the VMCS field specified by the secondary source operand is shorter than this effective operand size, the high bits of the primary source operand are ignored. If the VMCS field is longer, then the high bits of the field are cleared to 0.

Note that any faults resulting from accessing a memory destination operand can occur only after determining, in the operation section below, that the VMCS pointer is valid and that the specified VMCS field is supported.

### Operation

```

IF (not in VMX operation) or (RFLAGS.VM = 1) or
(IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF register source operand does not correspond to any VMCS field
    THEN VMfailValid(VMREAD/VMWRITE from/to unsupported VMCS component);
ELSE
    DEST ← contents of VMCS field indexed by register source operand;
    VMSucceed;
FI;
    
```



## Flags Affected

See the operation section and Section 5.2.

## Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPZ and REP/REPE/REPZ)
Segment overrides	Treated normally
Operand size	Causes #UD
Address size	Treated normally
REX	Register extensions treated normally; operand-size overrides ignored

## Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If a memory destination operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains an unusable segment.</p> <p>If the destination operand is located in a read-only data segment or any code segment.</p>
#PF(fault-code)	If a page fault occurs in accessing a memory destination operand.
#SS(0)	<p>If a memory destination operand effective address is outside the SS segment limit.</p> <p>If the SS register contains an unusable segment.</p>
#UD	If not in VMX operation.

## Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the VMREAD instruction is not recognized outside VMX operation.
-----	---

## Virtual-8086 Mode Exceptions

#UD	The VMREAD instruction is not recognized in virtual-8086 mode.
-----	--

## Compatibility Mode Exceptions

#UD	The VMREAD instruction is not recognized in compatibility mode.
-----	---

## VMX INSTRUCTION REFERENCE

### 64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0. If the memory destination operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing a memory destination operand.
#SS(0)	If the memory destination operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If not in VMX operation.

## **VMRESUME—Resume Virtual Machine**

See **VMLAUNCH/VMRESUME—Launch/Resume Virtual Machine**.

## VMWRITE—Write Field to Virtual-Machine Control Structure

Opcode	Instruction	Description
0F 79	VMWRITE r64, r/m64	Writes a specified VMCS field (in 64-bit mode)
0F 79	VMWRITE r32, r/m32	Writes a specified VMCS field (outside 64-bit mode)

### Description

Writes to a specified field in the VMCS specified by a secondary source operand (register only) using the contents of a primary source operand (register or memory).

The VMCS field is identified by the VMCS-field encoding contained in the register secondary source operand. Outside IA-32e mode, the secondary source operand is always 32 bits, regardless of the value of CS.D. In 64-bit mode, the secondary source operand has 64 bits; however, if bits 63:32 of the secondary source operand are not zero, VMWRITE will fail due to an attempt to access an unsupported VMCS component (see operation section).

The effective size of the primary source operand, which may be a register or in memory, is always 32 bits outside IA-32e mode (the setting of CS.D is ignored with respect to operand size) and 64 bits in 64-bit mode. If the VMCS field specified by the secondary source operand is shorter than this effective operand size, the high bits of the primary source operand are ignored. If the VMCS field is longer, then the high bits of the field are cleared to 0.

Note that any faults resulting from accessing a memory source operand occur after determining, in the operation section below, that the VMCS pointer is valid but before determining if the destination VMCS field is supported.

### Operation

```

IF (not in VMX operation) or (RFLAGS.VM = 1) or
(IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF register destination operand does not correspond to any VMCS field
    THEN VMfailValid(VMREAD/VMWRITE from/to unsupported VMCS component);
ELSIF VMCS field indexed by register destination operand is read-only)
    THEN VMfailValid(VMWRITE to read-only VMCS component);
ELSE
    VMCS field indexed by register destination operand ← SRC;
    VMsucceed;
FI;
    
```

## Flags Affected

See the operation section and Section 5.2.

## Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPZ and REP/REPE/REPZ)
Segment overrides	Treated normally
Operand size	Causes #UD
Address size	Treated normally
REX	Register extensions treated normally; operand-size overrides ignored

## Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If a memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains an unusable segment.</p> <p>If the source operand is located in an execute-only code segment.</p>
#PF(fault-code)	If a page fault occurs in accessing a memory source operand.
#SS(0)	<p>If a memory source operand effective address is outside the SS segment limit.</p> <p>If the SS register contains an unusable segment.</p>
#UD	If not in VMX operation.

## Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the VMWRITE instruction is not recognized outside VMX operation.
-----	--

## Virtual-8086 Mode Exceptions

#UD	The VMWRITE instruction is not recognized in virtual-8086 mode.
-----	---

## Compatibility Mode Exceptions

#UD	The VMWRITE instruction is not recognized in compatibility mode.
-----	--

## VMX INSTRUCTION REFERENCE

### 64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0. If the memory source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing a memory source operand.
#SS(0)	If the memory source operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If not in VMX operation.

## VMXOFF—Leave VMX Operation

Opcode	Instruction	Description
0F 01 C4	VMXOFF	Leaves VMX operation.

### Description

Takes the logical processor out of VMX operation, unblocks INIT signals, re-enables A20M, and clears any address-range monitoring.<sup>3</sup>

### Operation

```

IF (not in VMX operation) or (RFLAGS.VM = 1) or
(IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF dual-monitor treatment of SMIs and SMM is active
    THEN VMfail(VMXOFF under dual-monitor treatment of SMIs and SMM);
ELSE
    leave VMX operation;
    unblock INIT;
    unblock and enable A20M;
    clear address-range monitoring;
    VMSucceed;
FI;

```

### Flags Affected

See the operation section and Section 5.2.

### Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPZ and REP/REPE/REPZ)
Segment overrides	Ignored
Operand size	Causes #UD

---

3. See the information on MONITOR/MWAIT in Chapter 7, “Multiple-Processor Management,” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3A*.

## VMX INSTRUCTION REFERENCE

Address size            Ignored

REX                      Ignored

### Protected Mode Exceptions

#GP(0)                 If executed in VMX root operation with CPL > 0.

#UD                     If executed outside VMX operation.

### Real-Address Mode Exceptions

#UD                     A logical processor cannot be in real-address mode while in VMX operation and the VMXOFF instruction is not recognized outside VMX operation.

### Virtual-8086 Mode Exceptions

#UD                     The VMXOFF instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD                     The VMXOFF instruction is not recognized in compatibility mode.

### 64-Bit Mode Exceptions

#GP(0)                 If executed in VMX root operation with CPL > 0.

#UD                     If executed outside VMX operation.



## VMXON—Enter VMX Operation

Opcode	Instruction	Description
0F 0FC7 /6	VMXON m64	Enter VMX root operation.

### Description

Puts the logical processor in VMX operation with no current VMCS, blocks INIT signals, disables A20M, and clears any address-range monitoring established by the MONITOR instruction.<sup>4</sup>

The operand of this instruction is a 4KB-aligned physical address (the VMXON pointer) that references the VMXON region, which the logical processor may use to support VMX operation. This operand is always 64 bits and is always in memory.

### Operation

```

IF (register operand) or (CR4.VMXE = 0) or (CR0.PE = 0) or (RFLAGS.VM = 1) or
(IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF not in VMX operation
    THEN
        IF (CPL > 0) or (in A20M mode) or
        (the values of CR0 and CR4 are supported in VMX operation5) or
        (bit 0 (lock bit) of IA32_FEATURE_CONTROL MSR is clear) or
        (bit 2 of IA32_FEATURE_CONTROL MSR is clear)
            THEN #GP(0);
        ELSE
            addr ← contents of 64-bit in-memory source operand;
            IF addr is not 4KB-aligned or
            (processor supports Intel EM64T and
            addr sets any bits beyond the VMX physical-address width) or
            (processor does not support Intel EM64T and
            addr sets any bits in the range 63:32)
                THEN VMfailInvalid;
            ELSE
                rev ← 32 bits located at physical address addr;
                IF rev ≠ VMCS revision identifier supported by processor
                    THEN VMfailInvalid;
                ELSE
                    current-VMCS pointer ← FFFFFFFF_FFFFFFFFH;

```

4. See the information on MONITOR/MWAIT in Chapter 7, “Multiple-Processor Management,” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3A*.

5. See Section 19.8 of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B*.

## VMX INSTRUCTION REFERENCE

```
                                enter VMX operation;
                                block INIT signals;
                                block and disable A20M;
                                clear address-range monitoring;
                                VMsucceed;
                                FI;
                                FI;
                                FI;
                                ELSIF in VMX non-root operation
                                THEN VMexit;
                                ELSIF CPL > 0
                                THEN #GP(0);
                                ELSE VMfail("VMXON executed in VMX root operation");
                                FI;
```

### Flags Affected

See the operation section and Section 5.2.

### Use of Prefixes

LOCK	Causes #UD
REP*	Ignored (includes REPNE/REPNZ and REP/REPE/REPZ)
Segment overrides	Treated normally
Operand size	Ignored
Address size	Treated normally
REX	Register extensions treated normally; operand-size overrides ignored

### Protected Mode Exceptions

#GP(0)	If executed outside VMX operation with CPL>0 or with invalid CR0 or CR4 fixed bits.  If executed in A20M mode.  If the memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  If the DS, ES, FS, or GS register contains an unusable segment.  If the source operand is located in an execute-only code segment.
#PF(fault-code)	If a page fault occurs in accessing the memory source operand.
#SS(0)	If the memory source operand effective address is outside the SS segment limit.  If the SS register contains an unusable segment.

#UD                    If operand is a register.  
                           If executed with CR4.VMXE = 0.

**Real-Address Mode Exceptions**

#UD                    The VMXON instruction is not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

#UD                    The VMXON instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

#UD                    The VMXON instruction is not recognized in compatibility mode.

**64-Bit Mode Exceptions**

#GP(0)                If executed outside VMX operation with CPL > 0 or with invalid CR0 or CR4 fixed bits.

                          If executed in A20M mode.

                          If the source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.

#PF(fault-code)    If a page fault occurs in accessing the memory source operand.

#SS(0)                If the source operand is in the SS segment and the memory address is in a non-canonical form.

#UD                    If operand is a register.  
                           If executed with CR4.VMXE = 0.

**VMX INSTRUCTION REFERENCE**

**A**

# **Opcode Map**



# APPENDIX A OPCODE MAP

Use the opcode tables in this chapter to interpret IA-32 architecture object code. Instructions are divided into encoding groups:

- 1-byte, 2-byte and 3-byte opcode encodings are used to encode integer, system, MMX technology, SSE/SSE2/SSE3, and VMX instructions. Maps for these instructions are given in Table A-2 through Table A-6.
- Escape opcodes (in the format: ESC character, opcode, ModR/M byte) are used for floating-point instructions. The maps for these instructions are provided in Table A-7 through Table A-22.

## NOTE

All blanks in opcode maps are reserved and must not be used. Do not depend on the operation of undefined or blank opcodes.

## A.1 USING OPCODE TABLES

Tables in this appendix list opcodes of instructions (including required instruction prefixes, opcode extensions in associated ModR/M byte). Blank cells in the tables indicate opcodes that are reserved or undefined.

The opcode map tables are organized by hex values of the upper and lower 4 bits of an opcode byte. For 1-byte encodings (Table A-2), use the four high-order bits of an opcode to index a row of the opcode table; use the four low-order bits to index a column of the table. For 2-byte opcodes beginning with 0FH (Table A-3), skip any instruction prefixes, the 0FH byte (0FH may be preceded by 66H, F2H, or F3H) and use the upper and lower 4-bit values of the next opcode byte to index table rows and columns. Similarly, for 3-byte opcodes beginning with 0F38H or 0F3AH (Table A-4), skip any instruction prefixes, 0F38H or 0F3AH and use the upper and lower 4-bit values of the third opcode byte to index table rows and columns. See Section A.2.4, “Opcode Look-up Examples for One, Two, and Three-Byte Opcodes.”

When a ModR/M byte provides opcode extensions, this information qualifies opcode execution. For information on how an opcode extension in the ModR/M byte modifies the opcode map in Table A-2 and Table A-3, see Section A.4.

The escape (ESC) opcode tables for floating point instructions identify the eight high order bits of opcodes at the top of each page. See Section A.5. If the accompanying ModR/M byte is in the range of 00H-BFH, bits 3-5 (the top row of the third table on each page) along with the reg bits of ModR/M determine the opcode. ModR/M bytes outside the range of 00H-BFH are mapped by the bottom two tables on each page of the section.

### A.2 KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form *Zz*. The first character, an uppercase letter, specifies the addressing method; the second character, a lowercase letter, specifies the type of operand.

#### A.2.1 Codes for Addressing Method

The following abbreviations are used to document addressing methods:

- A Direct address: the instruction has no ModR/M byte; the address of the operand is encoded in the instruction. No base register, index register, or scaling factor can be applied (for example, far JMP (EA)).
- C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).
- D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).
- E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.
- F EFLAGS/RFLAGS Register.
- G The reg field of the ModR/M byte selects a general register (for example, AX (000)).
- I Immediate data: the operand value is encoded in subsequent bytes of the instruction.
- J The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (0E9), LOOP).
- M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).
- N The R/M field of the ModR/M byte selects a packed-quadword, MMX technology register.
- O The instruction has no ModR/M byte. The offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).
- P The reg field of the ModR/M byte selects a packed quadword MMX technology register.
- Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.
- R The R/M field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F23)).



- S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).
- U The R/M field of the ModR/M byte selects a 128-bit XMM register.
- V The reg field of the ModR/M byte selects a 128-bit XMM register.
- W A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.
- X Memory addressed by the DS:rSI register pair (for example, MOVS, CMPS, OUTS, or LODS).
- Y Memory addressed by the ES:rDI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

## A.2.2 Codes for Operand Type

The following abbreviations are used to document operand types:

- a Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).
- b Byte, regardless of operand-size attribute.
- c Byte or word, depending on operand-size attribute.
- d Doubleword, regardless of operand-size attribute.
- dq Double-quadword, regardless of operand-size attribute.
- p 32-bit or 48-bit pointer, depending on operand-size attribute.
- pi Quadword MMX technology register (for example: mm0).
- ps 128-bit packed single-precision floating-point data.
- q Quadword, regardless of operand-size attribute.
- s 6-byte or 10-byte pseudo-descriptor.
- ss Scalar element of a 128-bit packed single-precision floating data.
- si Doubleword integer register (for example: eax).
- v Word, doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
- w Word, regardless of operand-size attribute.
- z Word for 16-bit operand-size or doubleword for 32 or 64-bit operand-size.

## OPCODE MAP

### A.2.3 Register Codes

When an opcode requires a specific register as an operand, the register is identified by name (for example, AX, CL, or ESI). The name indicates whether the register is 64, 32, 16, or 8 bits wide.

A register identifier of the form eXX or rXX is used when register width depends on the operand-size attribute. eXX is used when 16 or 32-bit sizes are possible; rXX is used when 16, 32, or 64-bit sizes are possible. For example: eAX indicates that the AX register is used when the operand-size attribute is 16 and the EAX register is used when the operand-size attribute is 32. rAX can indicate AX, EAX or RAX.

When the REX.B bit is used to modify the register specified in the reg field of the opcode, this fact is indicated by adding “/x” to the register name to indicate the additional possibility. For example, rCX/r9 is used to indicate that the register could either be rCX or r9. Note that the size of r9 in this case is determined by the operand size attribute (just as for rCX).

### A.2.4 Opcode Look-up Examples for One, Two, and Three-Byte Opcodes

This section provides examples that demonstrate how opcode maps are used.

#### A.2.4.1 One-Byte Opcode Instructions

The opcode map for 1-byte opcodes is shown in Table A-2. The opcode map for 1-byte opcodes is arranged by row (the least-significant 4 bits of the hexadecimal value) and column (the most-significant 4 bits of the hexadecimal value). Each entry in the table lists one of the following types of opcodes:

- Instruction mnemonics and operand types using the notations listed in Section A.2
- Opcodes used as an instruction prefix

For each entry in the opcode map that corresponds to an instruction, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, “Instruction Format,” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 2A*. Operand types are listed according to notations listed in Section A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction prefix or entries for instructions without operands that use ModR/M (for example: 60H, PUSHA; 06H, PUSH ES).

### Example A-1. Look-up Example for 1-Byte Opcodes

Opcode 030500000000H for an ADD instruction is interpreted using the 1-byte opcode map (Table A-2) as follows:

- The first digit (0) of the opcode indicates the table row and the second digit (3) indicates the table column. This locates an opcode for ADD with two operands.
- The first operand (type Gv) indicates a general register that is a word or doubleword depending on the operand-size attribute. The second operand (type Ev) indicates a ModR/M byte follows that specifies whether the operand is a word or doubleword general-purpose register or a memory address.
- The ModR/M byte for this instruction is 05H, indicating that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3-5) is 000, indicating the EAX register.

The instruction for this opcode is ADD EAX, mem\_op, and the offset of mem\_op is 00000000H.

Some 1- and 2-byte opcodes point to group numbers (shaded entries in the opcode map table). Group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section A.4).

#### A.2.4.2 Two-Byte Opcode Instructions

The two-byte opcode map shown in Table A-3 includes primary opcodes that are either two bytes or three bytes in length. Primary opcodes that are 2 bytes in length begin with an escape opcode 0FH. The upper and lower four bits of the second opcode byte are used to index a particular row and column in Table A-3.

Two-byte opcodes that are 3 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and the escape opcode (0FH). The upper and lower four bits of the third byte are used to index a particular row and column in Table A-3 (except when the second opcode byte is the 3-byte escape opcodes 38H or 3AH; in this situation refer to Section A.2.4.3).

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, “Instruction Format,” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 2A*. The operand types are listed according to notations listed in Section A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction without operands that are encoded using ModR/M (for example: 0F77H, EMMS).

## OPCODE MAP

### Example A-2. Look-up Example for 2-Byte Opcodes

Look-up opcode 0FA405000000003H for a SHLD instruction using Table A-3.

- The opcode is located in row A, column 4. The location indicates a SHLD instruction with operands Ev, Gv, and Ib. Interpret the operands as follows:
  - Ev: The ModR/M byte follows the opcode to specify a word or doubleword operand.
  - Gv: The reg field of the ModR/M byte selects a general-purpose register.
  - Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The third byte is the ModR/M byte (05H). The mod and opcode/reg fields of ModR/M indicate that a 32-bit displacement is used to locate the first operand in memory and eAX as the second operand.
- The next part of the opcode is the 32-bit displacement for the destination memory operand (0000000H). The last byte stores immediate byte that provides the count of the shift (03H).
- By this breakdown, it has been shown that this opcode represents the instruction: SHLD DS:0000000H, EAX, 3.

### A.2.4.3 Three-Byte Opcode Instructions

The three-byte opcode maps shown in Table A-4 and Table A-5 includes primary opcodes that are either 3 or 4 bytes in length. Primary opcodes that are 3 bytes in length begin with two escape bytes 0F38H or 0F3A. The upper and lower four bits of the third opcode byte are used to index a particular row and column in Table A-4 or Table A-5.

Three-byte opcodes that are 4 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and two escape bytes (0F38H or 0F3AH). The upper and lower four bits of the fourth byte are used to index a particular row and column in Table A-4 or Table A-5.

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into the following case:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, “Instruction Format,” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 2A*. The operand types are listed according to notations listed in Section A.2.

### Example A-3. Look-up Example for 3-Byte Opcodes

Look-up opcode 660F3A0FC108H for a PALIGNR instruction using Table A-5.

- 66H is a prefix and 0F3AH indicate to use Table A-5. The opcode is located in row 0, column F indicating a PALIGNR instruction with operands Vdq, Wdq, and Ib. Interpret the operands as follows:
  - Vdq: The reg field of the ModR/M byte selects a 128-bit XMM register.

- Wdq: The R/M field of the ModR/M byte selects either a 128-bit XMM register or memory location.
- Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The next byte is the ModR/M byte (C1H). The reg field indicates that the first operand is XMM0. The mod shows that the R/M field specifies a register and the R/M indicates that the second operand is XMM1.
- The last byte is the immediate byte (08H).
- By this breakdown, it has been shown that this opcode represents the instruction: PALIGNR XMM0, XMM1, 8.

### A.2.5 Superscripts Utilized in Opcode Tables

Table A-1 contains notes on particular encodings. These notes are indicated in the following opcode maps by superscripts. Gray cells indicate instruction groupings.

**Table A-1. Superscripts Utilized in Opcode Tables**

Superscript Symbol	Meaning of Symbol
1A	Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section A.4, “Opcode Extensions For One-Byte And Two-byte Opcodes”).
1B	Use the 0F0B opcode (UD2 instruction) or the 0FB9H opcode when deliberately trying to generate an invalid opcode exception (#UD).
1C	Some instructions added in the Pentium III processor may use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to decode the instruction, see Table A-6.  These instructions include SFENCE, STMXCSR, LDMXCSR, FXRSTOR, and FXSAVE, as well as PREFETCH and its variations.
i64	The instruction is invalid or not encodable in 64-bit mode. 40 through 4F (single-byte INC and DEC) are REX prefix combinations when in 64-bit mode (use FE/FF Grp 4 and 5 for INC and DEC).
o64	Instruction is only available when in 64-bit mode.
d64	When in 64-bit mode, instruction defaults to 64-bit operand size and cannot encode 32-bit operand size.
f64	The operand size is forced to a 64-bit operand size when in 64-bit mode (prefixes that change operand size are ignored for this instruction in 64-bit mode).

## **OPCODE MAP**

### **A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS**

See Table A-2 through Table A-5 below. The tables are multiple page presentations. Rows and columns with sequential relationships are placed on facing pages to make look-up tasks easier. Note that table footnotes are not presented on each page. Table footnotes for each table are presented on the last page of the table.

Table A-2. One-byte Opcode Map: (00H — F7H) \*

	0	1	2	3	4	5	6	7
0	ADD						PUSH ES <sup>64</sup>	POP ES <sup>64</sup>
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, Iz		
1	ADC						PUSH SS <sup>64</sup>	POP SS <sup>64</sup>
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, Iz		
2	AND						SEG=ES (Prefix)	DAA <sup>64</sup>
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, Iz		
3	XOR						SEG=SS (Prefix)	AAA <sup>64</sup>
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, lb	rAX, Iz		
4	INC <sup>64</sup> general register / REX <sup>64</sup> Prefixes							
	eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH <sup>64</sup> general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA <sup>64</sup> / PUSHAD <sup>64</sup>	POPA <sup>64</sup> / POPAD <sup>64</sup>	BOUND <sup>64</sup> Gv, Ma	ARPL <sup>64</sup> Ew, Gw MOVSD <sup>64</sup> Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc <sup>64</sup> , Jb - Short-displacement jump on condition							
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1 <sup>1A</sup>				TEST		XCHG	
	Eb, lb	Ev, Iz	Eb, Ib <sup>64</sup>	Ev, lb	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	XCHG word, double-word or quad-word register with rAX						
		rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	MOV				MOVS/B Xb, Yb	MOVS/W/D/Q Xv, Yv	CMPS/B Xb, Yb	CMPS/W/D Xv, Yv
	AL, Ob	rAX, Ov	Ob, AL	Ov, rAX				
B	MOV immediate byte into byte register							
	AL/R8L, lb	CL/R9L, lb	DL/R10L, lb	BL/R11L, lb	AH/R12L, lb	CH/R13L, lb	DH/R14L, lb	BH/R15L, lb
C	Shift Grp 2 <sup>1A</sup>		RETN <sup>64</sup> lw	RETN <sup>64</sup>	LES <sup>64</sup> Gz, Mp	LDS <sup>64</sup> Gz, Mp	Grp 11 <sup>1A</sup> - MOV	
	Eb, lb	Ev, lb					Eb, lb	Ev, Iz
D	Shift Grp 2 <sup>1A</sup>				AAM <sup>64</sup> lb	AAD <sup>64</sup> lb		XLAT/ XLATB
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	LOOPNE <sup>64</sup> / LOOPNZ <sup>64</sup> Jb	LOOPE <sup>64</sup> / LOOPZ <sup>64</sup> Jb	LOOP <sup>64</sup> Jb	Jrcxz <sup>64</sup> / Jb	IN		OUT	
					AL, lb	eAX, lb	lb, AL	lb, eAX
F	LOCK (Prefix)		REPNE (Prefix)	REP/ REPE (Prefix)	HLT	CMC	Unary Grp 3 <sup>1A</sup>	
							Eb	Ev

# OPCODE MAP

**Table A-2. One-byte Opcode Map: (08H — FFH) \***

	8	9	A	B	C	D	E	F
0	OR Eb, Gb   Ev, Gv   Gb, Eb   Gv, Ev   AL, lb   rAX, lz						PUSH CS <sup>64</sup>	2-byte escape (Table A-3)
1	SBB Eb, Gb   Ev, Gv   Gb, Eb   Gv, Ev   AL, lb   rAX, lz						PUSH DS <sup>64</sup>	POP DS <sup>64</sup>
2	SUB Eb, Gb   Ev, Gv   Gb, Eb   Gv, Ev   AL, lb   rAX, lz						SEG=CS (Prefix)	DAS <sup>64</sup>
3	CMP Eb, Gb   Ev, Gv   Gb, Eb   Gv, Ev   AL, lb   rAX, lz						SEG=DS (Prefix)	AAS <sup>64</sup>
4	DEC <sup>64</sup> general register / REX <sup>064</sup> Prefixes eAX REX.W   eCX REX.WB   eDX REX.WX   eBX REX.WXB   eSP REX.WR   eBP REX.WRB   eSI REX.WRX   eDI REX.WRXB							
5	POP <sup>d64</sup> into general register rAX/r8   rCX/r9   rDX/r10   rBX/r11   rSP/r12   rBP/r13   rSI/r14   rDI/r15							
6	PUSH <sup>d64</sup> lz	IMUL Gv, Ev, lz	PUSH <sup>d64</sup> lb	IMUL Gv, Ev, lb	INS/INSB Yb, DX	INS/INSD Yz, DX	OUTS/OUTSB DX, Xb	OUTS/OUTSD DX, Xz
7	Jcc <sup>64</sup> , Jb- Short displacement jump on condition S   NS   P/PE   NP/PO   L/NGE   NL/GE   LE/NG   NLE/G							
8	MOV Eb, Gb   Ev, Gv   Gb, Eb   Gv, Ev   MOV Ev, Sw   LEA Gv, M   MOV Sw, Ev						Grp 1A <sup>1A</sup> POP <sup>d64</sup> Ev	
9	CBW/CWDE/CDQE	CWD/CDQ/CQO	CALL <sup>f64</sup> Ap	FWAIT/WAIT	PUSHF/D/Q <sup>d64</sup> Fv	POPF/D/Q <sup>d64</sup> Fv	SAHF	LAHF
A	TEST AL, lb   rAX, lz		STOS/B Yb, AL	STOS/W/D/Q Yv, rAX	LODS/B AL, Xb	LODS/W/D/Q rAX, Xv	SCAS/B AL, Yb	SCAS/W/D/Q rAX, Xv
B	MOV immediate word or double into word, double, or quad register rAX/r8, lv   rCX/r9, lv   rDX/r10, lv   rBX/r11, lv   rSP/r12, lv   rBP/r13, lv   rSI/r14, lv   rDI/r15, lv							
C	ENTER lw, lb	LEAVE <sup>d64</sup>	RETF lw	RETF	INT 3	INT lb	INTO <sup>64</sup>	IRET/D/Q
D	ESC (Escape to coprocessor instruction set)							
E	CALL <sup>f64</sup> Jz	near <sup>f64</sup> Jz	JMP far <sup>f64</sup> AP	short <sup>f64</sup> Jb	IN AL, DX   eAX, DX		OUT DX, AL   DX, eAX	
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4 <sup>1A</sup>	INC/DEC Grp 5 <sup>1A</sup>

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.



**Table A-3. Two-byte Opcode Map: 00H — 77H (First Byte is 0FH) \***

	0	1	2	3	4	5	6	7
0	Grp 6 <sup>1A</sup>	Grp 7 <sup>1A</sup>	LAR Gv, Ew	LSL Gv, Ew		SYSCALL <sup>064</sup>	CLTS	SYSRET <sup>064</sup>
1	movups Vps, Wps movss (F3) Vss, Wss movupd (66) Vpd, Wpd movsd (F2) Vsd, Wsd	movups Wps, Vps movss (F3) Wss, Vss movupd (66) Wpd, Vpd movsd (F2) Vsd, Wsd	movlps Vq, Mq movlpd (66) Vq, Mq movhlps Vq, Uq movddup(F2) Vq, Wq movsldup(F3) Vq, Wq	movlps Mq, Vq movlpd (66) Mq, Vq	unpcklps Vps, Wq unpcklpd (66) Vpd, Wq	unpckhps Vps, Wq unpckhpd (66) Vpd, Wq	movhps Vq, Mq movhpd (66) Vq, Mq movhlps Vq, Uq movshdup(F3) Vq, Wq	movhps Mq, Vq movhpd(66) Mq, Vq
2	MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3	WRMSR	RDTSR	RDMSR	RDPMSR	SYSENTER	SYSEXIT		
4	CMOVcc, (Gv, Ev) - Conditional Move							
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5	movmskps Gd, Ups movmskpd (66) Gd, Upd	sqrtps Vps, Wps sqrts (F3) Vss, Wss sqrtpd (66) Vpd, Wpd sqrtsd (F2) Vsd, Wsd	rsqrtps Vps, Wps rsqrts (F3) Vss, Wss	rcpps Vps, Wps rcps (F3) Vss, Wss	andps Vps, Wps andpd (66) Vpd, Wpd	andnps Vps, Wps andnpd (66) Vpd, Wpd	orps Vps, Wps orpd (66) Vpd, Wpd	xorps Vps, Wps xorpd (66) Vpd, Wpd
6	punpcklbw Pq, Qd punpcklbw (66) Vdq, Wdq	punpcklwd Pq, Qd punpcklwd (66) Vdq, Wdq	punpckldq Pq, Qd punpckldq (66) Vdq, Wdq	packsswb Pq, Qq packsswb (66) Vdq, Wdq	pcmpgtb Pq, Qq pcmpgtb (66) Vdq, Wdq	pcmpgtw Pq, Qq pcmpgtw (66) Vdq, Wdq	pcmpgtd Pq, Qq pcmpgtd (66) Vdq, Wdq	packuswb Pq, Qq packuswb (66) Vdq, Wdq
7	pshufw Pq, Qq, Ib pshufd (66) Vdq, Wdq, Ib pshufhw(F3) Vdq, Wdq, Ib pshufll (F2) Vdq, Wdq, Ib	(Grp 12 <sup>1A</sup> )	(Grp 13 <sup>1A</sup> )	(Grp 14 <sup>1A</sup> )	pcmpeqb Pq, Qq pcmpeqb (66) Vdq, Wdq	pcmpeqw Pq, Qq pcmpeqw (66) Vdq, Wdq	pcmpeqd Pq, Qq pcmpeqd (66) Vdq, Wdq	emms

# OPCODE MAP

**Table A-3. Two-byte Opcode Map: 08H — 7FH (First Byte is 0FH) \***

	8	9	A	B	C	D	E	F
0	INVD	WBINVD		2-byte Illegal Opcodes UD2 <sup>1B</sup>		NOP Ev		
1	Prefetch <sup>1C</sup> (Grp 16 <sup>1A</sup> )							NOP Ev
2	movaps Vps, Wps movapd (66) Vpd, Wpd	movaps Wps, Vps movapd (66) Wpd, Vpd	cvtpi2ps Vps, Qq cvtsi2ss (F3) Vss, Ed/q cvtpi2pd (66) Vpd, Qq cvtsi2sd (F2) Vsd, Ed/q	movntps Mps, Vps movntpd (66) Mpd, Vpd	cvtps2pi Qq, Wps cvtss2si (F3) Gd, Wss cvtpd2pi (66) Qdq, Wpd cvtsd2si (F2) Gd, Wsd	cvtps2pi Qq, Wps cvtss2si (F3) Gd/q, Wss cvtpd2pi (66) Qdq, Wpd cvtsd2si (F2) Gd/q, Wsd	ucomiss Vss, Wss ucomisd (66) Vsd, Wsd	comiss Vps, Wps comisd (66) Vsd, Wsd
3	3-byte escape (Table A-4)		3-byte escape (Table A-5)					
4	CMOVcc(Gv, Ev) - Conditional Move							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5	addps Vps, Wps addss (F3) Vss, Wss addpd (66) Vpd, Wpd addsd (F2) Vsd, Wsd	mulps Vps, Wps mulss (F3) Vss, Wss mulpd (66) Vpd, Wpd mulsd (F2) Vsd, Wsd	cvtps2pd Vpd, Wps cvtss2sd (F3) Vss, Wss cvtpd2ps (66) Vps, Wpd cvtsd2ss (F2) Vsd, Wsd	cvtqd2ps Vps, Wdq cvtps2dq (66) Vdq, Wps cvtps2dq (F3) Vdq, Wps	subps Vps, Wps subss (F3) Vss, Wss subpd (66) Vpd, Wpd subsd (F2) Vsd, Wsd	minps Vps, Wps minss (F3) Vss, Wss minpd (66) Vpd, Wpd minsd (F2) Vsd, Wsd	divps Vps, Wps divss (F3) Vss, Wss divpd (66) Vpd, Wpd divsd (F2) Vsd, Wsd	maxps Vps, Wps maxss (F3) Vss, Wss maxpd (66) Vpd, Wpd maxsd (F2) Vsd, Wsd
6	punpckhbw Pq, Qd punpckhbw (66) Pdq, Qdq	punpckhwd Pq, Qd punpckhwd (66) Pdq, Qdq	punpckhdq Pq, Qd punpckhdq (66) Pdq, Qdq	packssdw Pq, Qd packssdw (66) Pdq, Qdq	punpcklqdq (66) Vdq, Wdq	punpckhqdq (66) Vdq, Wdq	movd/q Pd, Ed/q movd/q (66) Vdq, Ed/q	movq Pq, Qq movdqa (66) Vdq, Wdq movdqu (F3) Vdq, Wdq
7	VMREAD Ed/q, Gd/q	VMWRITE Gd/q, Ed/q			haddps(F2) Vps, Wps haddpd(66) Vpd, Wpd	hsubps(F2) Vps, Wps hsubpd(66) Vpd, Wpd	movd/q Ed/q, Pd movd/q (66) Ed/q, Vdq movq (F3) Vq, Wq	movq Qq, Pq movdqa (66) Wdq, Vdq movdqu (F3) Wdq, Vdq

**Table A-3. Two-byte Opcode Map: 80H — F7H (First Byte is 0FH) \***

	0	1	2	3	4	5	6	7
8	Jcc <sup>64</sup> , Jz - Long-displacement jump on condition							
	O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
9	SETcc, Eb - Byte Set on condition							
	O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A	PUSH <sup>d64</sup> FS	POP <sup>d64</sup> FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B	CMPXCHG Eb, Gb      Ev, Gv		LSS Gv, Mp	BTR Ev, Gv	LFS Gv, Mp	LGS Gv, Mp	MOVZX Gv, Eb      Gv, Ew	
C	XADD Eb, Gb	XADD Ev, Gv	cmpps Vps, Wps, Ib cmpss (F3) Vss, Wss, Ib cmppd (66) Vpd, Wpd, Ib cmprd (F2) Vsd, Wsd, Ib	movnti Md/q, Gd/q	pinsrw Pq, Ew, Ib pinsrw (66) Vdq, Ew, Ib	pextrw Gd, Nq, Ib pextrw (66) Gd, Udq, Ib	shufps Vps, Wps, Ib shufpd (66) Vpd, Wpd, Ib	Grp 9 <sup>1A</sup>
D	addsubps(F2) Vps, Wps addsubpd(66) Vpd, Wpd	psrlw Pq, Qq psrlw (66) Vdq, Wdq	psrld Pq, Qq psrld (66) Vdq, Wdq	psrlq Pq, Qq psrlq (66) Vdq, Wdq	paddq Pq, Qq paddq (66) Vdq, Wdq	pmullw Pq, Qq pmullw (66) Vdq, Wdq	movq (66) Wq, Vq movq2dq (F3) Vdq, Nq movdq2q (F2) Pq, Uq	pmovmskb Gd, Nq pmovmskb (66) Gd, Udq
E	pavgb Pq, Qq pavgb (66) Vdq, Wdq	psraw Pq, Qq psraw (66) Vdq, Wdq	psrad Pq, Qq psrad (66) Vdq, Wdq	pavgw Pq, Qq pavgw (66) Vdq, Wdq	pmulhw Pq, Qq pmulhw (66) Vdq, Wdq	pmulhw Pq, Qq pmulhw (66) Vdq, Wdq	cvtpd2dq (F2) Vdq, Wpd cvttd2dq (66) Vdq, Wpd cvttd2pd (F3) Vpd, Wdq	movntq Mq, Pq movntdq (66) Mdq, Vdq
F	laddq (F2) Vdq, Mdq	psllw Pq, Qq psllw (66) Vdq, Wdq	pslld Pq, Qq pslld (66) Vdq, Wdq	psllq Pq, Qq psllq (66) Vdq, Wdq	pmuludq Pq, Qq pmuludq (66) Vdq, Wdq	pmaddwd Pq, Qq pmaddwd (66) Vdq, Wdq	psadbw Pq, Qq psadbw (66) Vdq, Wdq	maskmovq Pq, Nq maskmovdqu (66) Vdq, Udq

# OPCODE MAP

**Table A-3. Two-byte Opcode Map: 88H — FFH (First Byte is 0FH) \***

	8	9	A	B	C	D	E	F
8	Jcc <sup>64</sup> , Jz - Long-displacement jump on condition							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
9	SETcc, Eb - Byte Set on condition							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
A	PUSH <sup>d64</sup> GS	POP <sup>d64</sup> GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15 <sup>1A</sup> ) <sup>1C</sup>	IMUL Gv, Ev
B		Grp 10 <sup>1A</sup> Invalid Opcode <sup>1B</sup>	Grp 8 <sup>1A</sup> Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSX Gv, Eb      Gv, Ew	
C	BSWAP							
	RAX/EAX/ R8/R8D	RCX/ECX/ R9/R9D	RDX/EDX/ R10/R10D	RBX/EBX/ R11/R11D	RSP/ESP/ R12/R12D	RBP/EBP/ R13/R13D	RSI/ESI/ R14/R14D	RDI/EDI/ R15/R15D
D	psubusb Pq, Qq psubusb (66) Vdq, Wdq	psubusw Pq, Qq psubusw (66) Vdq, Wdq	pminub Pq, Qq pminub (66) Vdq, Wdq	pand Pq, Qq pand (66) Vdq, Wdq	paddusb Pq, Qq paddusb (66) Vdq, Wdq	paddusw Pq, Qq paddusw (66) Vdq, Wdq	pmaxub Pq, Qq pmaxub (66) Vdq, Wdq	pandn Pq, Qq pandn (66) Vdq, Wdq
E	psubsb Pq, Qq psubsb (66) Vdq, Wdq	psubsw Pq, Qq psubsw (66) Vdq, Wdq	pminsw Pq, Qq pminsw (66) Vdq, Wdq	por Pq, Qq por (66) Vdq, Wdq	paddsb Pq, Qq paddsb (66) Vdq, Wdq	paddsw Pq, Qq paddsw (66) Vdq, Wdq	pmaxsw Pq, Qq pmaxsw (66) Vdq, Wdq	pxor Pq, Qq pxor (66) Vdq, Wdq
F	psubb Pq, Qq psubb (66) Vdq, Wdq	psubw Pq, Qq psubw (66) Vdq, Wdq	psubd Pq, Qq psubd (66) Vdq, Wdq	psubq Pq, Qq psubq (66) Vdq, Wdq	paddb Pq, Qq paddb (66) Vdq, Wdq	paddw Pq, Qq paddw (66) Vdq, Wdq	paddd Pq, Qq paddd (66) Vdq, Wdq	

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**Table A-4. Three-byte Opcode Map: 00H — F7H (First Two Bytes are 0F 38H) \***

	0	1	2	3	4	5	6	7
0	pshufb Pq, Qq pshufb (66) Vdq, Wdq	phaddw Pq, Qq phaddw (66) Vdq, Wdq	phadd Pq, Qq phadd (66) Vdq, Wdq	phaddsw Pq, Qq phaddsw (66) Vdq, Wdq	pmaddubsw Pq, Qq pmaddubsw (66) Vdq, Wdq	phsubw Pq, Qq phsubw (66) Vdq, Wdq	phsubd Pq, Qq phsubd (66) Vdq, Wdq	phsubsw Pq, Qq phsubsw (66) Vdq, Wdq
1								
2								
3								
4								
5								
6								
7								
8								
9								
A								
B								
C								
D								
E								
F								

# OPCODE MAP

**Table A-4. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 38H) \***

	8	9	A	B	C	D	E	F
0	psignb Pq, Qq psignb (66) Vdq, Wdq	psignw Pq, Qq psignw (66) Vdq, Wdq	psignd Pq, Qq psignd (66) Vdq, Wdq	pmulhrsw Pq, Qq pmulhrsw (66) Vdq, Wdq				
1					pabsb Pq, Qq pabsb (66) Vdq, Wdq	pabsw Pq, Qq pabsw (66) Vdq, Wdq	pabsd Pq, Qq pabsd (66) Vdq, Wdq	
2								
3								
4								
5								
6								
7								
8								
9								
A								
B								
C								
D								
E								
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**Table A-5. Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH) \***

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								
A								
B								
C								
D								
E								
F								

## OPCODE MAP

**Table A-5. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 3AH) \***

	8	9	A	B	C	D	E	F
0								palignr Pq, Qq, lb palignr(66) Vdq, Wdq, lb
1								
2								
3								
4								
5								
6								
7								
8								
9								
A								
B								
C								
D								
E								
F								

**NOTES:**

- \* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.



## A.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES

Some 1-byte and 2-byte opcodes use bits 3-5 of the ModR/M byte (the nnn field in Figure A-1) as an extension of the opcode.

mod	nnn	R/M
-----	-----	-----

**Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)**

Opcodes that have opcode extensions are indicated in Table A-6 and organized by group number. Group numbers (from 1 to 16, second column) provide a table entry point. The encoding for the r/m field for each instruction can be established using the third column of the table.

### A.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

#### Example A-3. Interpreting an ADD Instruction

An ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction:

- Table A-6 indicates that the opcode extension field encoded in the ModR/M byte for this instruction is 000B.
- The r/m field can be encoded to access a register (11B) or a memory address using a specified addressing mode (for example: mem = 00B, 01B, 10B).

#### Example A-4. Looking Up 0F01C3H

Look up opcode 0F01C3 for a VMRESUME instruction by using Table A-2, Table A-3 and Table A-6:

- 0F tells us that this instruction is in the 2-byte opcode map.
- 01 (row 0, column 1 in Table A-3) reveals that this opcode is in Group 7 of Table A-6.
- C3 is the ModR/M byte. The first two bits of C3 are 11B. This tells us to look at the second of the Group 7 rows in Table A-6.
- The Op/Reg bits [5,4,3] are 000B. This tells us to look in the 000 column for Group 7.
- Finally, the R/M bits [2,1,0] are 011B. This identifies the opcode as the VMRESUME instruction.

# OPCODE MAP

## A.4.2 Opcode Extension Tables

See Table A-6 below.

**Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number \***

Opcode	Group	Mod 7,6	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
			000	001	010	011	100	101	110	111
80-83	1	mem, 11B	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
8F	1A	mem, 11B	POP							
C0, C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem, 11B	ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR
F6, F7	3	mem, 11B	TEST Ib/Iz		NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX
FE	4	mem, 11B	INC Eb	DEC Eb						
FF	5	mem, 11B	INC Ev	DEC Ev	CALLN <sup>f64</sup> Ev	CALLF Ep	JMPN <sup>f64</sup> Ev	JMPF Ep	PUSH <sup>d64</sup> Ev	
0F 00	6	mem, 11B	SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
0F 01	7	mem	SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb
		11B	VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001)						
0F BA	8	mem, 11B					BT	BTS	BTR	BTC
0F C7	9	mem		CMPXCH8B Mq CMPXCHG16B Mdq					VMPTRLD Mq VMCLEAR (66) Mq VMXON (F3) Mq	VMPTRST Mq
		11B								
0F B9	10	mem								
		11B								
C6	11	mem, 11B	MOV Eb, Ib							
C7		mem	MOV Ev, Iz							
		11B								

**Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number \***

Opcode	Group	Mod 7,6	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
			000	001	010	011	100	101	110	111
0F 71	12	mem								
		11B			psrlw Nq, lb psrlw (66) Udq, lb		psraw Nq, lb psraw (66) Udq, lb		psllw Nq, lb psllw (66) Udq, lb	
0F 72	13	mem								
		11B			psrld Nq, lb psrld (66) Udq, lb		psrad Nq, lb psrad (66) Udq, lb		pslld Nq, lb pslld (66) Udq, lb	
0F 73	14	mem								
		11B			psrlq Nq, lb psrlq (66) Udq, lb	psrldq (66) Udq, lb			psllq Nq, lb psllq (66) Udq, lb	pslldq (66) Udq, lb
0F AE	15	mem	fxsave	fxrstor	ldmxcsr	stmxcsr				
		11B						lfence	mfence	sfence
0F 18	16	mem	prefetch NTA	prefetch T0	prefetch T1	prefetch T2				
		11B								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## OPCODE MAP

### A.5 ESCAPE OPCODE INSTRUCTIONS

Opcode maps for coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are in Table A-7 through Table A-22. These maps are grouped by the first byte of the opcode, from D8-DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H-BFH, bits 3-5 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1- and 2-byte opcodes (see Section A.4). If the ModR/M byte is outside the range of 00H through BFH, the entire ModR/M byte is used as an opcode extension.

#### A.5.1 Opcode Look-up Examples for Escape Instruction Opcodes

Examples are provided below.

##### Example A-5. Opcode with ModR/M Byte in the 00H through BFH Range

DD0504000000H can be interpreted as follows:

- The instruction encoded with this opcode can be located in Section . Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode for an FLD double-real instruction (see Table A-9).
- The double-real value to be loaded is at 00000004H (the 32-bit displacement that follows and belongs to this opcode).

##### Example A-6. Opcode with ModR/M Byte outside the 00H through BFH Range

D8C1H can be interpreted as follows:

- This example illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction can be located in Section A.4.
- In Table A-8, the ModR/M byte C1H indicates row C, column 1 (the FADD instruction using ST(0), ST(1) as operands).

#### A.5.2 Escape Opcode Instruction Tables

Tables are listed below.

### A.5.2.1 Escape Opcodes with D8 as First Byte

Table A-7 and A-8 contain maps for the escape instruction opcodes that begin with D8H. Table A-7 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-7. D8 Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte (refer to Figure A.4)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD single-real	FMUL single-real	FCOM single-real	FCOMP single-real	FSUB single-real	FSUBR single-real	FDIV single-real	FDIVR single-real

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-8 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-8. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOM							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIV							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOMP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUBR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIVR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## OPCODE MAP

### A.5.2.2 Escape Opcodes with D9 as First Byte

Table A-9 and A-10 contain maps for escape instruction opcodes that begin with D9H. Table A-9 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD single-real		FST single-real	FSTP single-real	FLDENV 14/28 bytes	FLDCW 2 bytes	FSTENV 14/28 bytes	FSTCW 2 bytes

**NOTES:**

- \* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-10 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FLD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FNOP							
E	FCHS	FABS			FTST	FXAM		
F	F2XM1	FYL2X	FPTAN	FPATAN	FXTRACT	FPREM1	FDECSTP	FINCSTP

	8	9	A	B	C	D	E	F
C	FXCH							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D								
E	FLD1	FLDL2T	FLDL2E	FLDPI	FLDLG2	FLDLN2	FLDZ	
F	FPREM	FYL2XP1	FSQRT	FSINCOS	FRNDINT	FSCALE	FSIN	FCOS

**NOTES:**

- \* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.3 Escape Opcodes with DA as First Byte

Table A-11 and A-12 contain maps for escape instruction opcodes that begin with DAH. Table A-11 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-11. DA Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD dword-integer	FIMUL dword-integer	FICOM dword-integer	FICOMP dword-integer	FISUB dword-integer	FISUBR dword-integer	FIDIV dword-integer	FIDIVR dword-integer

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-11 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-12. DA Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FCMOV B							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOV BE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E								
F								

	8	9	A	B	C	D	E	F
C	FCMOV E							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOV U							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E		FUCOMPP						
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## OPCODE MAP

### A.5.2.4 Escape Opcodes with DB as First Byte

Table A-13 and A-14 contain maps for escape instruction opcodes that begin with DBH. Table A-13 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-13. DB Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD dword-integer	FISTTP dword- integer	FIST dword-integer	FISTP dword-integer		FLD extended-real		FSTP extended-real

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-14 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FCMOVNB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E			FCLEX	FINIT				
F	FCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FCMOVNE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FUCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.



### A.5.2.5 Escape Opcodes with DC as First Byte

Table A-15 and A-16 contain maps for escape instruction opcodes that begin with DCH. Table A-15 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD double-real	FMUL double-real	FCOM double-real	FCOMP double-real	FSUB double-real	FSUBR double-real	FDIV double-real	FDIVR double-real

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-16 shows the map if the ModR/M byte is outside the range of 00H-BFH. In this case the first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUB							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIV							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## OPCODE MAP

### A.5.2.6 Escape Opcodes with DD as First Byte

Table A-17 and A-18 contain maps for escape instruction opcodes that begin with DDH. Table A-17 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-17. DD Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD double-real	FISTTP integer64	FST double-real	FSTP double-real	FRSTOR 98/108bytes		FSAVE 98/108bytes	FSTSW 2 bytes

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-18 shows the map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FFREE							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
D	FST							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOM							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F								

	8	9	A	B	C	D	E	F
C								
D	FSTP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOMP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
F								

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.7 Escape Opcodes with DE as First Byte

Table A-19 and A-20 contain opcode maps for escape instruction opcodes that begin with DEH. Table A-19 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. In this case, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-19. DE Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD word-integer	FIMUL word-integer	FICOM word-integer	FICOMP word-integer	FISUB word-integer	FISUBR word-integer	FIDIV word-integer	FIDIVR word-integer

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-20 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C	FADDP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMULP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D		FCOMP						
E	FSUBP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

**NOTES:**

\* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

## OPCODE MAP

### A.5.2.8 Escape Opcodes with DF As First Byte

Table A-21 and A-22 contain the opcode maps for escape instruction opcodes that begin with DFH. Table A-21 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

**Table A-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH \***

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD word-integer	FISTTP word-integer	FIST word-integer	FISTP word-integer	FBLD packed- BCD	FILD qword-integer	FBSTP packed- BCD	FISTP qword-integer

**NOTES:**

- \* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-22 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

**Table A-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH \***

	0	1	2	3	4	5	6	7
C								
D								
E	FSTSW AX							
F	FCOMP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C								
D								
E	FUCOMP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

**NOTES:**

- \* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

**B**

**Instruction Formats  
and Encodings**



# APPENDIX B

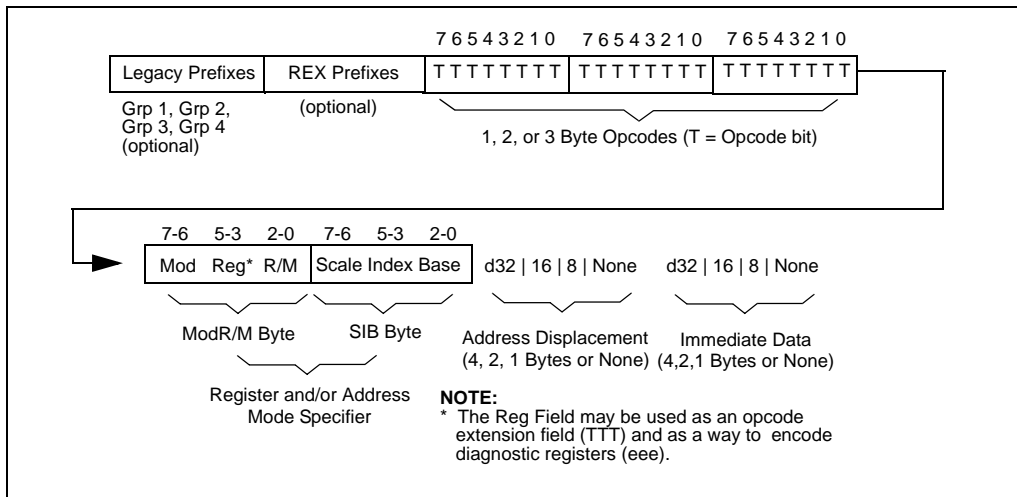
## INSTRUCTION FORMATS AND ENCODINGS

This appendix provides machine instruction formats and encodings of IA-32 architecture instructions. The first section describes the IA-32 architecture's machine instruction format. The remaining sections show the formats and encoding of general-purpose, MMX, P6 family, SSE/SSE2/SSE3, x87 FPU instructions, and VMX instructions. Instruction formats used in 64-bit mode are provided as supersets of the above.

### B.1 MACHINE INSTRUCTION FORMAT

All Intel Architecture instructions are encoded using subsets of the general machine instruction format shown in Figure B-1. Each instruction consists of:

- an opcode
- a register and/or address mode specifier consisting of the ModR/M byte and sometimes the scale-index-base (SIB) byte (if required)
- a displacement and an immediate data field (if required)



**Figure B-1. General Machine Instruction Format**

The following sections discuss this format.

## INSTRUCTION FORMATS AND ENCODINGS

### B.1.1 Legacy Prefixes

The legacy prefixes noted in Figure B-1 include 66H, 67H, F2H and F3H. They are optional, except when F2H, F3H and 66H are used in new instruction extensions. Legacy prefixes must be placed before REX prefixes.

Refer to Chapter 2, “Instruction Format,” in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 2A*, for more information on legacy prefixes.

### B.1.2 REX Prefixes

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in legacy IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

Refer to Chapter 2, “Instruction Format,” in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 2A*, for more information on REX prefixes.

### B.1.3 Opcode Fields

The primary opcode for an instruction is encoded in one to three bytes of the instruction. Within the primary opcode, smaller encoding fields may be defined. These fields vary according to the class of operation being performed.

Almost all instructions that refer to a register and/or memory operand have a register and/or address mode byte following the opcode. This byte, the ModR/M byte, consists of the mod field (3 bits), the reg field (3 bits; this field is sometimes an opcode extension), and the R/M field (2 bits). Certain encodings of the ModR/M byte indicate that a second address mode byte, the SIB byte, must be used.

If the addressing mode specifies a displacement, the displacement value is placed immediately following the ModR/M byte or SIB byte. Possible sizes are 8, 16, or 32 bits. If the instruction specifies an immediate value, the immediate value follows any displacement bytes. The immediate, if specified, is always the last field of the instruction.

Refer to Chapter 2, “Instruction Format,” in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 2A*, for more information on opcodes.

### B.1.4 Special Fields

Table B-1 lists bit fields that appear in certain instructions, sometimes within the opcode bytes. All of these fields (except the d bit) occur in the general-purpose instruction formats in Table B-13.



**Table B-1. Special Fields Within Instruction Encodings**

Field Name	Description	Number of Bits
reg	General-register specifier (see Table B-4 or B-5)	3
w	Specifies if data is byte or full-sized, where full-sized is 16 or 32 bits (see Table B-6)	1
s	Specifies sign extension of an immediate field (see Table B-7)	1
sreg2	Segment register specifier for CS, SS, DS, ES (see Table B-8)	2
sreg3	Segment register specifier for CS, SS, DS, ES, FS, GS (see Table B-8)	3
eee	Specifies a special-purpose (control or debug) register (see Table B-9)	3
ttn	For conditional instructions, specifies a condition asserted or negated (see Table B-12)	4
d	Specifies direction of data operation (see Table B-11)	1

#### B.1.4.1 Reg Field (reg) for Non-64-Bit Modes

The reg field in the ModR/M byte specifies a general-purpose register operand. The group of registers specified is modified by the presence and state of the w bit in an encoding (refer to Section B.1.4.3). Table B-2 shows the encoding of the reg field when the w bit is not present in an encoding; Table B-3 shows the encoding of the reg field when the w bit is present.

**Table B-2. Encoding of reg Field When w Field is Not Present in Instruction**

reg Field	Register Selected during 16-Bit Data Operations	Register Selected during 32-Bit Data Operations
000	AX	EAX
001	CX	ECX
010	DX	EDX
011	BX	EBX
100	SP	ESP
101	BP	EBP
110	SI	ESI
111	DI	EDI

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-3. Encoding of reg Field When w Field is Present in Instruction**

Register Specified by reg Field During 16-Bit Data Operations			Register Specified by reg Field During 32-Bit Data Operations		
reg	Function of w Field		reg	Function of w Field	
	When w = 0	When w = 1		When w = 0	When w = 1
000	AL	AX	000	AL	EAX
001	CL	CX	001	CL	ECX
010	DL	DX	010	DL	EDX
011	BL	BX	011	BL	EBX
100	AH	SP	100	AH	ESP
101	CH	BP	101	CH	EBP
110	DH	SI	110	DH	ESI
111	BH	DI	111	BH	EDI

### B.1.4.2 Reg Field (reg) for 64-Bit Mode

Just like in non-64-bit modes, the reg field in the ModR/M byte specifies a general-purpose register operand. The group of registers specified is modified by the presence of and state of the w bit in an encoding (refer to Section B.1.4.3). Table B-4 shows the encoding of the reg field when the w bit is not present in an encoding; Table B-5 shows the encoding of the reg field when the w bit is present.

**Table B-4. Encoding of reg Field When w Field is Not Present in Instruction**

reg Field	Register Selected during 16-Bit Data Operations	Register Selected during 32-Bit Data Operations	Register Selected during 64-Bit Data Operations
000	AX	EAX	RAX
001	CX	ECX	RCX
010	DX	EDX	RDX
011	BX	EBX	RBX
100	SP	ESP	RSP
101	BP	EBP	RBP
110	SI	ESI	RSI
111	DI	EDI	RDI

**Table B-5. Encoding of reg Field When w Field is Present in Instruction**

Register Specified by reg Field During 16-Bit Data Operations			Register Specified by reg Field During 32-Bit Data Operations		
Function of w Field			Function of w Field		
reg	When w = 0	When w = 1	reg	When w = 0	When w = 1
000	AL	AX	000	AL	EAX
001	CL	CX	001	CL	ECX
010	DL	DX	010	DL	EDX
011	BL	BX	011	BL	EBX
100	AH <sup>1</sup>	SP	100	AH*	ESP
101	CH <sup>1</sup>	BP	101	CH*	EBP
110	DH <sup>1</sup>	SI	110	DH*	ESI
111	BH <sup>1</sup>	DI	111	BH*	EDI

**NOTES:**

1. AH, CH, DH, BH can not be encoded when REX prefix is used. Such an expression defaults to the low byte.

**B.1.4.3 Encoding of Operand Size (w) Bit**

The current operand-size attribute determines whether the processor is performing 16-bit, 32-bit or 64-bit operations. Within the constraints of the current operand-size attribute, the operand-size bit (w) can be used to indicate operations on 8-bit operands or the full operand size specified with the operand-size attribute. Table B-6 shows the encoding of the w bit depending on the current operand-size attribute.

**Table B-6. Encoding of Operand Size (w) Bit**

w Bit	Operand Size When Operand-Size Attribute is 16 Bits	Operand Size When Operand-Size Attribute is 32 Bits
0	8 Bits	8 Bits
1	16 Bits	32 Bits

**B.1.4.4 Sign-Extend (s) Bit**

The sign-extend (s) bit occurs in instructions with immediate data fields that are being extended from 8 bits to 16 or 32 bits. See Table B-7.

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-7. Encoding of Sign-Extend (s) Bit**

s	Effect on 8-Bit Immediate Data	Effect on 16- or 32-Bit Immediate Data
0	None	None
1	Sign-extend to fill 16-bit or 32-bit destination	None

### B.1.4.5 Segment Register (sreg) Field

When an instruction operates on a segment register, the reg field in the ModR/M byte is called the sreg field and is used to specify the segment register. Table B-8 shows the encoding of the sreg field. This field is sometimes a 2-bit field (sreg2) and other times a 3-bit field (sreg3).

**Table B-8. Encoding of the Segment Register (sreg) Field**

2-Bit sreg2 Field	Segment Register Selected	3-Bit sreg3 Field	Segment Register Selected
00	ES	000	ES
01	CS	001	CS
10	SS	010	SS
11	DS	011	DS
		100	FS
		101	GS
		110	Reserved <sup>1</sup>
		111	Reserved

**NOTES:**

1. Do not use reserved encodings.

**B.1.4.6 Special-Purpose Register (eee) Field**

When control or debug registers are referenced in an instruction they are encoded in the eee field, located in bits 5 though 3 of the ModR/M byte (an alternate encoding of the sreg field). See Table B-9.

**Table B-9. Encoding of Special-Purpose Register (eee) Field**

eee	Control Register	Debug Register
000	CR0	DR0
001	Reserved <sup>1</sup>	DR1
010	CR2	DR2
011	CR3	DR3
100	CR4	Reserved
101	Reserved	Reserved
110	Reserved	DR6
111	Reserved	DR7

**NOTES:**

- 1. Do not use reserved encodings.

**B.1.4.7 Condition Test (ttn) Field**

For conditional instructions (such as conditional jumps and set on condition), the condition test field (ttn) is encoded for the condition being tested. The ttt part of the field gives the condition to test and the n part indicates whether to use the condition (n = 0) or its negation (n = 1).

- For 1-byte primary opcodes, the ttn field is located in bits 3, 2, 1, and 0 of the opcode byte.
- For 2-byte primary opcodes, the ttn field is located in bits 3, 2, 1, and 0 of the second opcode byte.

Table B-10 shows the encoding of the ttn field.

**Table B-10. Encoding of Conditional Test (ttn) Field**

t t t n	Mnemonic	Condition
0000	O	Overflow
0001	NO	No overflow
0010	B, NAE	Below, Not above or equal
0011	NB, AE	Not below, Above or equal
0100	E, Z	Equal, Zero
0101	NE, NZ	Not equal, Not zero
0110	BE, NA	Below or equal, Not above
0111	NBE, A	Not below or equal, Above

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-10. Encoding of Conditional Test (ttn) Field (Contd.)**

t t n	Mnemonic	Condition
1000	S	Sign
1001	NS	Not sign
1010	P, PE	Parity, Parity Even
1011	NP, PO	Not parity, Parity Odd
1100	L, NGE	Less than, Not greater than or equal to
1101	NL, GE	Not less than, Greater than or equal to
1110	LE, NG	Less than or equal to, Not greater than
1111	NLE, G	Not less than or equal to, Greater than

### B.1.4.8 Direction (d) Bit

In many two-operand instructions, a direction bit (d) indicates which operand is considered the source and which is the destination. See Table B-11.

- When used for integer instructions, the d bit is located at bit 1 of a 1-byte primary opcode. Note that this bit does not appear as the symbol “d” in Table B-13; the actual encoding of the bit as 1 or 0 is given.
- When used for floating-point instructions (in Table B-16), the d bit is shown as bit 2 of the first byte of the primary opcode.

**Table B-11. Encoding of Operation Direction (d) Bit**

d	Source	Destination
0	reg Field	ModR/M or SIB Byte
1	ModR/M or SIB Byte	reg Field

### B.1.5 Other Notes

Table B-12 contains notes on particular encodings. These notes are indicated in the tables shown in the following sections by superscripts.

**Table B-12. Notes on Instruction Encoding**

Symbol	Note
A	A value of 11B in bits 7 and 6 of the ModR/M byte is reserved.

## B.2 GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS FOR NON-64-BIT MODES

Table B-13 shows machine instruction formats and encodings for general purpose instructions in non-64-bit modes.

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes**

Instruction and Format	Encoding
<b>AAA – ASCII Adjust after Addition</b>	0011 0111
<b>AAD – ASCII Adjust AX before Division</b>	1101 0101 : 0000 1010
<b>AAM – ASCII Adjust AX after Multiply</b>	1101 0100 : 0000 1010
<b>AAS – ASCII Adjust AL after Subtraction</b>	0011 1111
<b>ADC – ADD with Carry</b>	
register1 to register2	0001 000w : 11 reg1 reg2
register2 to register1	0001 001w : 11 reg1 reg2
memory to register	0001 001w : mod reg r/m
register to memory	0001 000w : mod reg r/m
immediate to register	1000 00sw : 11 010 reg : immediate data
immediate to AL, AX, or EAX	0001 010w : immediate data
immediate to memory	1000 00sw : mod 010 r/m : immediate data
<b>ADD – Add</b>	
register1 to register2	0000 000w : 11 reg1 reg2
register2 to register1	0000 001w : 11 reg1 reg2
memory to register	0000 001w : mod reg r/m
register to memory	0000 000w : mod reg r/m
immediate to register	1000 00sw : 11 000 reg : immediate data
immediate to AL, AX, or EAX	0000 010w : immediate data
immediate to memory	1000 00sw : mod 000 r/m : immediate data
<b>AND – Logical AND</b>	
register1 to register2	0010 000w : 11 reg1 reg2
register2 to register1	0010 001w : 11 reg1 reg2
memory to register	0010 001w : mod reg r/m
register to memory	0010 000w : mod reg r/m
immediate to register	1000 00sw : 11 100 reg : immediate data
immediate to AL, AX, or EAX	0010 010w : immediate data
immediate to memory	1000 00sw : mod 100 r/m : immediate data

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>ARPL – Adjust RPL Field of Selector</b> from register	0110 0011 : 11 reg1 reg2
from memory	0110 0011 : mod reg r/m
<b>BOUND – Check Array Against Bounds</b>	0110 0010 : mod <sup>A</sup> reg r/m
<b>BSF – Bit Scan Forward</b>	
register1, register2	0000 1111 : 1011 1100 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1100 : mod reg r/m
<b>BSR – Bit Scan Reverse</b>	
register1, register2	0000 1111 : 1011 1101 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1101 : mod reg r/m
<b>BSWAP – Byte Swap</b>	0000 1111 : 1100 1 reg
<b>BT – Bit Test</b>	
register, immediate	0000 1111 : 1011 1010 : 11 100 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 100 r/m : imm8 data
register1, register2	0000 1111 : 1010 0011 : 11 reg2 reg1
memory, reg	0000 1111 : 1010 0011 : mod reg r/m
<b>BTC – Bit Test and Complement</b>	
register, immediate	0000 1111 : 1011 1010 : 11 111 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 111 r/m : imm8 data
register1, register2	0000 1111 : 1011 1011 : 11 reg2 reg1
memory, reg	0000 1111 : 1011 1011 : mod reg r/m
<b>BTR – Bit Test and Reset</b>	
register, immediate	0000 1111 : 1011 1010 : 11 110 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 110 r/m : imm8 data
register1, register2	0000 1111 : 1011 0011 : 11 reg2 reg1
memory, reg	0000 1111 : 1011 0011 : mod reg r/m
<b>BTS – Bit Test and Set</b>	
register, immediate	0000 1111 : 1011 1010 : 11 101 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 101 r/m : imm8 data
register1, register2	0000 1111 : 1010 1011 : 11 reg2 reg1
memory, reg	0000 1111 : 1010 1011 : mod reg r/m



## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>CALL – Call Procedure (in same segment)</b>	
direct	1110 1000 : full displacement
register indirect	1111 1111 : 11 010 reg
memory indirect	1111 1111 : mod 010 r/m
<b>CALL – Call Procedure (in other segment)</b>	
direct	1001 1010 : unsigned full offset, selector
indirect	1111 1111 : mod 011 r/m
<b>CBW – Convert Byte to Word</b>	1001 1000
<b>CDQ – Convert Doubleword to Qword</b>	1001 1001
<b>CLC – Clear Carry Flag</b>	1111 1000
<b>CLD – Clear Direction Flag</b>	1111 1100
<b>CLI – Clear Interrupt Flag</b>	1111 1010
<b>CLTS – Clear Task-Switched Flag in CR0</b>	0000 1111 : 0000 0110
<b>CMC – Complement Carry Flag</b>	1111 0101
<b>CMP – Compare Two Operands</b>	
register1 with register2	0011 100w : 11 reg1 reg2
register2 with register1	0011 101w : 11 reg1 reg2
memory with register	0011 100w : mod reg r/m
register with memory	0011 101w : mod reg r/m
immediate with register	1000 00sw : 11 111 reg : immediate data
immediate with AL, AX, or EAX	0011 110w : immediate data
immediate with memory	1000 00sw : mod 111 r/m : immediate data
<b>CMPS/CMPSB/CMPSW/CMPSD – Compare String Operands</b>	1010 011w
<b>CMPXCHG – Compare and Exchange</b>	
register1, register2	0000 1111 : 1011 000w : 11 reg2 reg1
memory, register	0000 1111 : 1011 000w : mod reg r/m
<b>CPUID – CPU Identification</b>	0000 1111 : 1010 0010
<b>CWD – Convert Word to Doubleword</b>	1001 1001
<b>CWDE – Convert Word to Doubleword</b>	1001 1000
<b>DAA – Decimal Adjust AL after Addition</b>	0010 0111
<b>DAS – Decimal Adjust AL after Subtraction</b>	0010 1111

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>DEC – Decrement by 1</b> register register (alternate encoding) memory	1111 111w : 11 001 reg 0100 1 reg 1111 111w : mod 001 r/m
<b>DIV – Unsigned Divide</b> AL, AX, or EAX by register AL, AX, or EAX by memory	1111 011w : 11 110 reg 1111 011w : mod 110 r/m
<b>ENTER – Make Stack Frame for High Level Procedure</b>	1100 1000 : 16-bit displacement : 8-bit level (L)
<b>HLT – Halt</b>	1111 0100
<b>IDIV – Signed Divide</b> AL, AX, or EAX by register AL, AX, or EAX by memory	1111 011w : 11 111 reg 1111 011w : mod 111 r/m
<b>IMUL – Signed Multiply</b> AL, AX, or EAX with register AL, AX, or EAX with memory register1 with register2 register with memory register1 with immediate to register2 memory with immediate to register	1111 011w : 11 101 reg 1111 011w : mod 101 reg 0000 1111 : 1010 1111 : 11 : reg1 reg2 0000 1111 : 1010 1111 : mod reg r/m 0110 10s1 : 11 reg1 reg2 : immediate data 0110 10s1 : mod reg r/m : immediate data
<b>IN – Input From Port</b> fixed port variable port	1110 010w : port number 1110 110w
<b>INC – Increment by 1</b> reg reg (alternate encoding) memory	1111 111w : 11 000 reg 0100 0 reg 1111 111w : mod 000 r/m
<b>INS – Input from DX Port</b>	0110 110w
<b>INT n – Interrupt Type n</b>	1100 1101 : type
<b>INT – Single-Step Interrupt 3</b>	1100 1100
<b>INTO – Interrupt 4 on Overflow</b>	1100 1110
<b>INVD – Invalidate Cache</b>	0000 1111 : 0000 1000

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>INVLPG – Invalidate TLB Entry</b>	0000 1111 : 0000 0001 : mod 111 r/m
<b>IRET/IRETD – Interrupt Return</b>	1100 1111
<b>Jcc – Jump if Condition is Met</b>	
8-bit displacement	0111 ttn : 8-bit displacement
full displacement	0000 1111 : 1000 ttn : full displacement
<b>JCXZ/JECXZ – Jump on CX/ECX Zero</b> Address-size prefix differentiates JCXZ and JECXZ	1110 0011 : 8-bit displacement
<b>JMP – Unconditional Jump (to same segment)</b>	
short	1110 1011 : 8-bit displacement
direct	1110 1001 : full displacement
register indirect	1111 1111 : 11 100 reg
memory indirect	1111 1111 : mod 100 r/m
<b>JMP – Unconditional Jump (to other segment)</b>	
direct intersegment	1110 1010 : unsigned full offset, selector
indirect intersegment	1111 1111 : mod 101 r/m
<b>LAHF – Load Flags into AH Register</b>	1001 1111
<b>LAR – Load Access Rights Byte</b>	
from register	0000 1111 : 0000 0010 : 11 reg1 reg2
from memory	0000 1111 : 0000 0010 : mod reg r/m
<b>LDS – Load Pointer to DS</b>	1100 0101 : mod <sup>A</sup> reg r/m
<b>LEA – Load Effective Address</b>	1000 1101 : mod <sup>A</sup> reg r/m
<b>LEAVE – High Level Procedure Exit</b>	1100 1001
<b>LES – Load Pointer to ES</b>	1100 0100 : mod <sup>A</sup> reg r/m
<b>LFS – Load Pointer to FS</b>	0000 1111 : 1011 0100 : mod <sup>A</sup> reg r/m
<b>LGDT – Load Global Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 010 r/m
<b>LGS – Load Pointer to GS</b>	0000 1111 : 1011 0101 : mod <sup>A</sup> reg r/m
<b>LIDT – Load Interrupt Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 011 r/m
<b>LLDT – Load Local Descriptor Table Register</b>	
LDTR from register	0000 1111 : 0000 0000 : 11 010 reg
LDTR from memory	0000 1111 : 0000 0000 : mod 010 r/m

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>LMSW – Load Machine Status Word</b> from register from memory	0000 1111 : 0000 0001 : 11 110 reg 0000 1111 : 0000 0001 : mod 110 r/m
<b>LOCK – Assert LOCK# Signal Prefix</b>	1111 0000
<b>LODS/LODSB/LODSW/LODSD – Load String Operand</b>	1010 110w
<b>LOOP – Loop Count</b>	1110 0010 : 8-bit displacement
<b>LOOPZ/LOOPE – Loop Count while Zero/Equal</b>	1110 0001 : 8-bit displacement
<b>LOOPNZ/LOOPNE – Loop Count while not Zero/Equal</b>	1110 0000 : 8-bit displacement
<b>LSL – Load Segment Limit</b> from register from memory	0000 1111 : 0000 0011 : 11 reg1 reg2 0000 1111 : 0000 0011 : mod reg r/m
<b>LSS – Load Pointer to SS</b>	0000 1111 : 1011 0010 : mod <sup>A</sup> reg r/m
<b>LTR – Load Task Register</b> from register from memory	0000 1111 : 0000 0000 : 11 011 reg 0000 1111 : 0000 0000 : mod 011 r/m
<b>MOV – Move Data</b> register1 to register2 register2 to register1 memory to reg reg to memory immediate to register immediate to register (alternate encoding) immediate to memory memory to AL, AX, or EAX AL, AX, or EAX to memory	1000 100w : 11 reg1 reg2 1000 101w : 11 reg1 reg2 1000 101w : mod reg r/m 1000 100w : mod reg r/m 1100 011w : 11 000 reg : immediate data 1011 w reg : immediate data 1100 011w : mod 000 r/m : immediate data 1010 000w : full displacement 1010 001w : full displacement
<b>MOV – Move to/from Control Registers</b> CR0 from register CR2 from register CR3 from register CR4 from register register from CR0-CR4	0000 1111 : 0010 0010 : 11 000 reg 0000 1111 : 0010 0010 : 11 010reg 0000 1111 : 0010 0010 : 11 011 reg 0000 1111 : 0010 0010 : 11 100 reg 0000 1111 : 0010 0000 : 11 eee reg

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>MOV – Move to/from Debug Registers</b>	
DR0-DR3 from register	0000 1111 : 0010 0011 : 11 eee reg
DR4-DR5 from register	0000 1111 : 0010 0011 : 11 eee reg
DR6-DR7 from register	0000 1111 : 0010 0011 : 11 eee reg
register from DR6-DR7	0000 1111 : 0010 0001 : 11 eee reg
register from DR4-DR5	0000 1111 : 0010 0001 : 11 eee reg
register from DR0-DR3	0000 1111 : 0010 0001 : 11 eee reg
<b>MOV – Move to/from Segment Registers</b>	
register to segment register	1000 1110 : 11 sreg3 reg
register to SS	1000 1110 : 11 sreg3 reg
memory to segment reg	1000 1110 : mod sreg3 r/m
memory to SS	1000 1110 : mod sreg3 r/m
segment register to register	1000 1100 : 11 sreg3 reg
segment register to memory	1000 1100 : mod sreg3 r/m
<b>MOVS/MOVSMB/MOVSQB/MOVSQ – Move Data from String to String</b>	
	1010 010w
<b>MOVSX – Move with Sign-Extend</b>	
register2 to register1	0000 1111 : 1011 111w : 11 reg1 reg2
memory to reg	0000 1111 : 1011 111w : mod reg r/m
<b>MOVZX – Move with Zero-Extend</b>	
register2 to register1	0000 1111 : 1011 011w : 11 reg1 reg2
memory to register	0000 1111 : 1011 011w : mod reg r/m
<b>MUL – Unsigned Multiply</b>	
AL, AX, or EAX with register	1111 011w : 11 100 reg
AL, AX, or EAX with memory	1111 011w : mod 100 reg
<b>NEG – Two's Complement Negation</b>	
register	1111 011w : 11 011 reg
memory	1111 011w : mod 011 r/m
<b>NOP – No Operation</b>	
	1001 0000
<b>NOP – Multi-byte No Operation<sup>1</sup></b>	
register	0000 1111 0001 1111 : 11 000 reg
memory	0000 1111 0001 1111 : mod 000 r/m

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>NOT – One's Complement Negation</b>	
register	1111 011w : 11 010 reg
memory	1111 011w : mod 010 r/m
<b>OR – Logical Inclusive OR</b>	
register1 to register2	0000 100w : 11 reg1 reg2
register2 to register1	0000 101w : 11 reg1 reg2
memory to register	0000 101w : mod reg r/m
register to memory	0000 100w : mod reg r/m
immediate to register	1000 00sw : 11 001 reg : immediate data
immediate to AL, AX, or EAX	0000 110w : immediate data
immediate to memory	1000 00sw : mod 001 r/m : immediate data
<b>OUT – Output to Port</b>	
fixed port	1110 011w : port number
variable port	1110 111w
<b>OUTS – Output to DX Port</b>	
	0110 111w
<b>POP – Pop a Word from the Stack</b>	
register	1000 1111 : 11 000 reg
register (alternate encoding)	0101 1 reg
memory	1000 1111 : mod 000 r/m
<b>POP – Pop a Segment Register from the Stack</b> (Note: CS cannot be sreg2 in this usage.)	
segment register DS, ES	000 sreg2 111
segment register SS	000 sreg2 111
segment register FS, GS	0000 1111: 10 sreg3 001
<b>POPA/POPAD – Pop All General Registers</b>	
	0110 0001
<b>POPF/POPFD – Pop Stack into FLAGS or EFLAGS Register</b>	
	1001 1101
<b>PUSH – Push Operand onto the Stack</b>	
register	1111 1111 : 11 110 reg
register (alternate encoding)	0101 0 reg
memory	1111 1111 : mod 110 r/m
immediate	0110 10s0 : immediate data

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>PUSH – Push Segment Register onto the Stack</b>	
segment register CS,DS,ES,SS	000 sreg2 110
segment register FS,GS	0000 1111: 10 sreg3 000
<b>PUSHA/PUSHAD – Push All General Registers</b>	0110 0000
<b>PUSHF/PUSHFD – Push Flags Register onto the Stack</b>	1001 1100
<b>RCL – Rotate thru Carry Left</b>	
register by 1	1101 000w : 11 010 reg
memory by 1	1101 000w : mod 010 r/m
register by CL	1101 001w : 11 010 reg
memory by CL	1101 001w : mod 010 r/m
register by immediate count	1100 000w : 11 010 reg : imm8 data
memory by immediate count	1100 000w : mod 010 r/m : imm8 data
<b>RCR – Rotate thru Carry Right</b>	
register by 1	1101 000w : 11 011 reg
memory by 1	1101 000w : mod 011 r/m
register by CL	1101 001w : 11 011 reg
memory by CL	1101 001w : mod 011 r/m
register by immediate count	1100 000w : 11 011 reg : imm8 data
memory by immediate count	1100 000w : mod 011 r/m : imm8 data
<b>RDMSR – Read from Model-Specific Register</b>	0000 1111 : 0011 0010
<b>RDPMS – Read Performance Monitoring Counters</b>	0000 1111 : 0011 0011
<b>RDTSC – Read Time-Stamp Counter</b>	0000 1111 : 0011 0001
<b>REP INS – Input String</b>	1111 0011 : 0110 110w
<b>REP LODS – Load String</b>	1111 0011 : 1010 110w
<b>REP MOVS – Move String</b>	1111 0011 : 1010 010w
<b>REP OUTS – Output String</b>	1111 0011 : 0110 111w
<b>REP STOS – Store String</b>	1111 0011 : 1010 101w
<b>REPE CMPS – Compare String</b>	1111 0011 : 1010 011w
<b>REPE SCAS – Scan String</b>	1111 0011 : 1010 111w
<b>REPNE CMPS – Compare String</b>	1111 0010 : 1010 011w

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>REPNE SCAS – Scan String</b>	1111 0010 : 1010 111w
<b>RET – Return from Procedure (to same segment)</b>	
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
<b>RET – Return from Procedure (to other segment)</b>	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement
<b>ROL – Rotate Left</b>	
register by 1	1101 000w : 11 000 reg
memory by 1	1101 000w : mod 000 r/m
register by CL	1101 001w : 11 000 reg
memory by CL	1101 001w : mod 000 r/m
register by immediate count	1100 000w : 11 000 reg : imm8 data
memory by immediate count	1100 000w : mod 000 r/m : imm8 data
<b>ROR – Rotate Right</b>	
register by 1	1101 000w : 11 001 reg
memory by 1	1101 000w : mod 001 r/m
register by CL	1101 001w : 11 001 reg
memory by CL	1101 001w : mod 001 r/m
register by immediate count	1100 000w : 11 001 reg : imm8 data
memory by immediate count	1100 000w : mod 001 r/m : imm8 data
<b>RSM – Resume from System Management Mode</b>	0000 1111 : 1010 1010
<b>SAHF – Store AH into Flags</b>	1001 1110
<b>SAL – Shift Arithmetic Left</b>	same instruction as SHL
<b>SAR – Shift Arithmetic Right</b>	
register by 1	1101 000w : 11 111 reg
memory by 1	1101 000w : mod 111 r/m
register by CL	1101 001w : 11 111 reg
memory by CL	1101 001w : mod 111 r/m
register by immediate count	1100 000w : 11 111 reg : imm8 data
memory by immediate count	1100 000w : mod 111 r/m : imm8 data



## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>SBB – Integer Subtraction with Borrow</b>	
register1 to register2	0001 100w : 11 reg1 reg2
register2 to register1	0001 101w : 11 reg1 reg2
memory to register	0001 101w : mod reg r/m
register to memory	0001 100w : mod reg r/m
immediate to register	1000 00sw : 11 011 reg : immediate data
immediate to AL, AX, or EAX	0001 110w : immediate data
immediate to memory	1000 00sw : mod 011 r/m : immediate data
<b>SCAS/SCASB/SCASW/SCASD – Scan String</b>	
	1010 111w
<b>SETcc – Byte Set on Condition</b>	
register	0000 1111 : 1001 ttn : 11 000 reg
memory	0000 1111 : 1001 ttn : mod 000 r/m
<b>SGDT – Store Global Descriptor Table Register</b>	
	0000 1111 : 0000 0001 : mod <sup>A</sup> 000 r/m
<b>SHL – Shift Left</b>	
register by 1	1101 000w : 11 100 reg
memory by 1	1101 000w : mod 100 r/m
register by CL	1101 001w : 11 100 reg
memory by CL	1101 001w : mod 100 r/m
register by immediate count	1100 000w : 11 100 reg : imm8 data
memory by immediate count	1100 000w : mod 100 r/m : imm8 data
<b>SHLD – Double Precision Shift Left</b>	
register by immediate count	0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8
memory by immediate count	0000 1111 : 1010 0100 : mod reg r/m : imm8
register by CL	0000 1111 : 1010 0101 : 11 reg2 reg1
memory by CL	0000 1111 : 1010 0101 : mod reg r/m
<b>SHR – Shift Right</b>	
register by 1	1101 000w : 11 101 reg
memory by 1	1101 000w : mod 101 r/m
register by CL	1101 001w : 11 101 reg
memory by CL	1101 001w : mod 101 r/m
register by immediate count	1100 000w : 11 101 reg : imm8 data
memory by immediate count	1100 000w : mod 101 r/m : imm8 data

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
<b>SHRD – Double Precision Shift Right</b> register by immediate count memory by immediate count register by CL memory by CL	0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8 0000 1111 : 1010 1100 : mod reg r/m : imm8 0000 1111 : 1010 1101 : 11 reg2 reg1 0000 1111 : 1010 1101 : mod reg r/m
<b>SIDT – Store Interrupt Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 001 r/m
<b>SLDT – Store Local Descriptor Table Register</b> to register to memory	0000 1111 : 0000 0000 : 11 000 reg 0000 1111 : 0000 0000 : mod 000 r/m
<b>SMSW – Store Machine Status Word</b> to register to memory	0000 1111 : 0000 0001 : 11 100 reg 0000 1111 : 0000 0001 : mod 100 r/m
<b>STC – Set Carry Flag</b>	1111 1001
<b>STD – Set Direction Flag</b>	1111 1101
<b>STI – Set Interrupt Flag</b>	1111 1011
<b>STOS/STOSB/STOSW/STOSD – Store String Data</b>	1010 101w
<b>STR – Store Task Register</b> to register to memory	0000 1111 : 0000 0000 : 11 001 reg 0000 1111 : 0000 0000 : mod 001 r/m
<b>SUB – Integer Subtraction</b> register1 to register2 register2 to register1 memory to register register to memory immediate to register immediate to AL, AX, or EAX immediate to memory	0010 100w : 11 reg1 reg2 0010 101w : 11 reg1 reg2 0010 101w : mod reg r/m 0010 100w : mod reg r/m 1000 00sw : 11 101 reg : immediate data 0010 110w : immediate data 1000 00sw : mod 101 r/m : immediate data
<b>TEST – Logical Compare</b> register1 and register2 memory and register immediate and register immediate and AL, AX, or EAX	1000 010w : 11 reg1 reg2 1000 010w : mod reg r/m 1111 011w : 11 000 reg : immediate data 1010 100w : immediate data

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings  
for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
immediate and memory	1111 011w : mod 000 r/m : immediate data
<b>UD2 – Undefined instruction</b>	0000 FFFF : 0000 1011
<b>VERR – Verify a Segment for Reading</b>	
register	0000 1111 : 0000 0000 : 11 100 reg
memory	0000 1111 : 0000 0000 : mod 100 r/m
<b>VERW – Verify a Segment for Writing</b>	
register	0000 1111 : 0000 0000 : 11 101 reg
memory	0000 1111 : 0000 0000 : mod 101 r/m
<b>WAIT – Wait</b>	1001 1011
<b>WBINVD – Writeback and Invalidate Data Cache</b>	0000 1111 : 0000 1001
<b>WRMSR – Write to Model-Specific Register</b>	0000 1111 : 0011 0000
<b>XADD – Exchange and Add</b>	
register1, register2	0000 1111 : 1100 000w : 11 reg2 reg1
memory, reg	0000 1111 : 1100 000w : mod reg r/m
<b>XCHG – Exchange Register/Memory with Register</b>	
register1 with register2	1000 011w : 11 reg1 reg2
AX or EAX with reg	1001 0 reg
memory with reg	1000 011w : mod reg r/m
<b>XLAT/XLATB – Table Look-up Translation</b>	1101 0111
<b>XOR – Logical Exclusive OR</b>	
register1 to register2	0011 000w : 11 reg1 reg2
register2 to register1	0011 001w : 11 reg1 reg2
memory to register	0011 001w : mod reg r/m
register to memory	0011 000w : mod reg r/m
immediate to register	1000 00sw : 11 110 reg : immediate data
immediate to AL, AX, or EAX	0011 010w : immediate data
immediate to memory	1000 00sw : mod 110 r/m : immediate data
<b>Prefix Bytes</b>	
address size	0110 0111
LOCK	1111 0000
operand size	0110 0110

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)**

Instruction and Format	Encoding
CS segment override	0010 1110
DS segment override	0011 1110
ES segment override	0010 0110
FS segment override	0110 0100
GS segment override	0110 0101
SS segment override	0011 0110

### NOTES:

1. The multi-byte NOP instruction does not alter the content of the register and will not issue a memory operation.

## B.2.1 General Purpose Instruction Formats and Encodings for 64-Bit Mode

Table B-15 shows machine instruction formats and encodings for general purpose instructions in 64-bit mode.

**Table B-14. Special Symbols**

Symbol	Application
S	If the value of REX.W. is 1, it overrides the presence of 66H.
w	The value of bit W. in REX is has no effect.

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode**

Instruction and Format	Encoding
<b>ADC – ADD with Carry</b>	
register1 to register2	0100 0R0B : 0001 000w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B : 0001 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0R0B : 0001 001w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B : 0001 0011 : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB : 0001 001w : mod reg r/m
memory to qwordregister	0100 1RXB : 0001 0011 : mod qwordreg r/m
register to memory	0100 0RXB : 0001 000w : mod reg r/m

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
qwordregister to memory	0100 1RXB : 0001 0001 : mod qwordreg r/m
immediate to register	0100 000B : 1000 00sw : 11 010 reg : immediate
immediate to qwordregister	0100 100B : 1000 0001 : 11 010 qwordreg : imm32
immediate to qwordregister	0100 1R0B : 1000 0011 : 11 010 qwordreg : imm8
immediate to AL, AX, or EAX	0001 010w : immediate data
immediate to RAX	0100 1000 : 0000 0101 : imm32
immediate to memory	0100 00XB : 1000 00sw : mod 010 r/m : immediate
immediate32 to memory64	0100 10XB : 1000 0001 : mod 010 r/m : imm32
immediate8 to memory64	0100 10XB : 1000 0031 : mod 010 r/m : imm8
<b>ADD – Add</b>	
register1 to register2	0100 0R0B : 0000 000w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B 0000 0000 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0R0B : 0000 001w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B 0000 0010 : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB : 0000 001w : mod reg r/m
memory64 to qwordregister	0100 1RXB : 0000 0000 : mod qwordreg r/m
register to memory	0100 0RXB : 0000 000w : mod reg r/m
qwordregister to memory64	0100 1RXB : 0000 0011 : mod qwordreg r/m
immediate to register	0100 0000B : 1000 00sw : 11 000 reg : immediate data
immediate32 to qwordregister	0100 100B : 1000 0001 : 11 010 qwordreg : imm
immediate to AL, AX, or EAX	0000 010w : immediate8
immediate to RAX	0100 1000 : 0000 0101 : imm32
immediate to memory	0100 00XB : 1000 00sw : mod 000 r/m : immediate
immediate32 to memory64	0100 10XB : 1000 0001 : mod 010 r/m : imm32
immediate8 to memory64	0100 10XB : 1000 0011 : mod 010 r/m : imm8
<b>AND – Logical AND</b>	
register1 to register2	0100 0R0B 0010 000w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B 0010 0001 : 11 qwordreg1 qwordreg2
register2 to register1	0100 0R0B 0010 001w : 11 reg1 reg2
register1 to register2	0100 1R0B 0010 0011 : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB 0010 001w : mod reg r/m

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
memory64 to qwordregister	0100 1RXB : 0010 0011 : mod qwordreg r/m
register to memory	0100 0RXB : 0010 000w : mod reg r/m
qwordregister to memory64	0100 1RXB : 0010 0001 : mod qwordreg r/m
immediate to register	0100 000B : 1000 00sw : 11 100 reg : immediate
immediate32 to qwordregister	0100 100B 1000 0001 : 11 100 qwordreg : imm32
immediate to AL, AX, or EAX	0010 010w : immediate
immediate32 to RAX	0100 1000 0010 1001 : imm32
immediate to memory	0100 00XB : 1000 00sw : mod 100 r/m : immediate
immediate32 to memory64	0100 10XB : 1000 0001 : mod 100 r/m : immediate32
immediate8 to memory64	0100 10XB : 1000 0011 : mod 100 r/m : imm8
<b>BSF – Bit Scan Forward</b>	
register1, register2	0100 0R0B 0000 1111 : 1011 1100 : 11 reg1 reg2
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1011 1100 : 11 qwordreg1 qwordreg2
memory, register	0100 0RXB 0000 1111 : 1011 1100 : mod reg r/m
memory64, qwordregister	0100 1RXB 0000 1111 : 1011 1100 : mod qwordreg r/m
<b>BSR – Bit Scan Reverse</b>	
register1, register2	0100 0R0B 0000 1111 : 1011 1101 : 11 reg1 reg2
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1011 1101 : 11 qwordreg1 qwordreg2
memory, register	0100 0RXB 0000 1111 : 1011 1101 : mod reg r/m
memory64, qwordregister	0100 1RXB 0000 1111 : 1011 1101 : mod qwordreg r/m
<b>BSWAP – Byte Swap</b>	
BSWAP – Byte Swap	0000 1111 : 1100 1 reg
BSWAP – Byte Swap	0100 100B 0000 1111 : 1100 1 qwordreg
<b>BT – Bit Test</b>	
register, immediate	0100 000B 0000 1111 : 1011 1010 : 11 100 reg: imm8
qwordregister, immediate8	0100 100B 1111 : 1011 1010 : 11 100 qwordreg: imm8 data
memory, immediate	0100 00XB 0000 1111 : 1011 1010 : mod 100 r/m : imm8
memory64, immediate8	0100 10XB 0000 1111 : 1011 1010 : mod 100 r/m : imm8 data

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
register1, register2	0100 0R0B 0000 1111 : 1010 0011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1010 0011 : 11 qwordreg2 qwordreg1
memory, reg	0100 0RXB 0000 1111 : 1010 0011 : mod reg r/m
memory, qwordreg	0100 1RXB 0000 1111 : 1010 0011 : mod qwordreg r/m
<b>BTC – Bit Test and Complement</b>	
register, immediate	0100 000B 0000 1111 : 1011 1010 : 11 111 reg: imm8
qwordregister, immediate8	0100 100B 0000 1111 : 1011 1010 : 11 111 qwordreg: imm8
memory, immediate	0100 00XB 0000 1111 : 1011 1010 : mod 111 r/m : imm8
memory64, immediate8	0100 10XB 0000 1111 : 1011 1010 : mod 111 r/m : imm8
register1, register2	0100 0R0B 0000 1111 : 1011 1011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1011 1011 : 11 qwordreg2 qwordreg1
memory, register	0100 0RXB 0000 1111 : 1011 1011 : mod reg r/m
memory, qwordreg	0100 1RXB 0000 1111 : 1011 1011 : mod qwordreg r/m
<b>BTR – Bit Test and Reset</b>	
register, immediate	0100 000B 0000 1111 : 1011 1010 : 11 110 reg: imm8
qwordregister, immediate8	0100 100B 0000 1111 : 1011 1010 : 11 110 qwordreg: imm8
memory, immediate	0100 00XB 0000 1111 : 1011 1010 : mod 110 r/m : imm8
memory64, immediate8	0100 10XB 0000 1111 : 1011 1010 : mod 110 r/m : imm8
register1, register2	0100 0R0B 0000 1111 : 1011 0011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1011 0011 : 11 qwordreg2 qwordreg1
memory, register	0100 0RXB 0000 1111 : 1011 0011 : mod reg r/m
memory64, qwordreg	0100 1RXB 0000 1111 : 1011 0011 : mod qwordreg r/m

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>BTS – Bit Test and Set</b>	
register, immediate	0100 000B 0000 1111 : 1011 1010 : 11 101 reg: imm8
qwordregister, immediate8	0100 100B 0000 1111 : 1011 1010 : 11 101 qwordreg: imm8
memory, immediate	0100 00XB 0000 1111 : 1011 1010 : mod 101 r/m : imm8
memory64, immediate8	0100 10XB 0000 1111 : 1011 1010 : mod 101 r/m : imm8
register1, register2	0100 0R0B 0000 1111 : 1010 1011 : 11 reg2 reg1
qwordregister1, qwordregister2	0100 1R0B 0000 1111 : 1010 1011 : 11 qwordreg2 qwordreg1
memory, register	0100 0RXB 0000 1111 : 1010 1011 : mod reg r/m
memory64, qwordreg	0100 1RXB 0000 1111 : 1010 1011 : mod qwordreg r/m
<b>CALL – Call Procedure (in same segment)</b>	
direct	1110 1000 : displacement32
register indirect	0100 WR00 <sup>W</sup> 1111 1111 : 11 010 reg
memory indirect	0100 W0XB <sup>W</sup> 1111 1111 : mod 010 r/m
<b>CALL – Call Procedure (in other segment)</b>	
indirect	1111 1111 : mod 011 r/m
indirect	0100 10XB 0100 1000 1111 1111 : mod 011 r/m
<b>CBW – Convert Byte to Word</b>	1001 1000
<b>CDQ – Convert Doubleword to Qword+</b>	1001 1001
<b>CDQE – RAX, Sign-Extend of EAX</b>	0100 1000 1001 1001
<b>CLC – Clear Carry Flag</b>	1111 1000
<b>CLD – Clear Direction Flag</b>	1111 1100
<b>CLI – Clear Interrupt Flag</b>	1111 1010
<b>CLTS – Clear Task-Switched Flag in CR0</b>	0000 1111 : 0000 0110
<b>CMC – Complement Carry Flag</b>	1111 0101



## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>CMP – Compare Two Operands</b>	
register1 with register2	0100 0R0B 0011 100w : 11 reg1 reg2
qwordregister1 with qwordregister2	0100 1R0B 0011 1001 : 11 qwordreg1 qwordreg2
register2 with register1	0100 0R0B 0011 101w : 11 reg1 reg2
qwordregister2 with qwordregister1	0100 1R0B 0011 101w : 11 qwordreg1 qwordreg2
memory with register	0100 0RXB 0011 100w : mod reg r/m
memory64 with qwordregister	0100 1RXB 0011 1001 : mod qwordreg r/m
register with memory	0100 0RXB 0011 101w : mod reg r/m
qwordregister with memory64	0100 1RXB 0011 101w1 : mod qwordreg r/m
immediate with register	0100 000B 1000 00sw : 11 111 reg : imm
immediate32 with qwordregister	0100 100B 1000 0001 : 11 111 qwordreg : imm64
immediate with AL, AX, or EAX	0011 110w : imm
immediate32 with RAX	0100 1000 0011 1101 : imm32
immediate with memory	0100 00XB 1000 00sw : mod 111 r/m : imm
immediate32 with memory64	0100 1RXB 1000 0001 : mod 111 r/m : imm64
immediate8 with memory64	0100 1RXB 1000 0011 : mod 111 r/m : imm8
<b>CMPS/CMPSB/CMPSW/CMPSD/CMPSQ – Compare String Operands</b>	
compare string operands [ X at DS:(E)SI with Y at ES:(E)DI ]	1010 011w
qword at address RSI with qword at address RDI	0100 1000 1010 0111
<b>CMPXCHG – Compare and Exchange</b>	
register1, register2	0000 1111 : 1011 000w : 11 reg2 reg1
byteregister1, byteregister2	0100 000B 0000 1111 : 1011 0000 : 11 bytereg2 reg1
qwordregister1, qwordregister2	0100 100B 0000 1111 : 1011 0001 : 11 qwordreg2 reg1
memory, register	0000 1111 : 1011 000w : mod reg r/m
memory8, byteregister	0100 00XB 0000 1111 : 1011 0000 : mod bytereg r/m
memory64, qwordregister	0100 10XB 0000 1111 : 1011 0001 : mod qwordreg r/m

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>CPUID – CPU Identification</b>	0000 1111 : 1010 0010
<b>CQO – Sign-Extend RAX</b>	0100 1000 1001 1001
<b>CWD – Convert Word to Doubleword</b>	1001 1001
<b>CWDE – Convert Word to Doubleword</b>	1001 1000
<b>DEC – Decrement by 1</b>	
register	0100 000B 1111 111w : 11 001 reg
qwordregister	0100 100B 1111 1111 : 11 001 qwordreg
memory	0100 00XB 1111 111w : mod 001 r/m
memory64	0100 10XB 1111 1111 : mod 001 r/m
<b>DIV – Unsigned Divide</b>	
AL, AX, or EAX by register	0100 000B 1111 011w : 11 110 reg
Divide RDX:RAX by qwordregister	0100 100B 1111 0111 : 11 110 qwordreg
AL, AX, or EAX by memory	0100 00XB 1111 011w : mod 110 r/m
Divide RDX:RAX by memory64	0100 10XB 1111 0111 : mod 110 r/m
<b>ENTER – Make Stack Frame for High Level Procedure</b>	1100 1000 : 16-bit displacement : 8-bit level (L)
<b>HLT – Halt</b>	1111 0100
<b>IDIV – Signed Divide</b>	
AL, AX, or EAX by register	0100 000B 1111 011w : 11 111 reg
RDX:RAX by qwordregister	0100 100B 1111 0111 : 11 111 qwordreg
AL, AX, or EAX by memory	0100 00XB 1111 011w : mod 111 r/m
RDX:RAX by memory64	0100 10XB 1111 0111 : mod 111 r/m
<b>IMUL – Signed Multiply</b>	
AL, AX, or EAX with register	0100 000B 1111 011w : 11 101 reg
RDX:RAX <- RAX with qwordregister	0100 100B 1111 0111 : 11 101 qwordreg
AL, AX, or EAX with memory	0100 00XB 1111 011w : mod 101 r/m
RDX:RAX <- RAX with memory64	0100 10XB 1111 0111 : mod 101 r/m
register1 with register2	0000 1111 : 1010 1111 : 11 : reg1 reg2
qwordregister1 <- qwordregister1 with qwordregister2	0100 1R0B 0000 1111 : 1010 1111 : 11 : qwordreg1 qwordreg2
register with memory	0100 0RXB 0000 1111 : 1010 1111 : mod reg r/m
qwordregister <- qwordregister with memory64	0100 1RXB 0000 1111 : 1010 1111 : mod qwordreg r/m

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
register1 with immediate to register2	0100 0R0B 0110 10s1 : 11 reg1 reg2 : imm
qwordregister1 <- qwordregister2 with sign-extended immediate8	0100 1R0B 0110 1011 : 11 qwordreg1 qwordreg2 : imm8
qwordregister1 <- qwordregister2 with immediate32	0100 1R0B 0110 1001 : 11 qwordreg1 qwordreg2 : imm32
memory with immediate to register	0100 0RXB 0110 10s1 : mod reg r/m : imm
qwordregister <- memory64 with sign-extended immediate8	0100 1RXB 0110 1011 : mod qwordreg r/m : imm8
qwordregister <- memory64 with immediate32	0100 1RXB 0110 1001 : mod qwordreg r/m : imm32
<b>IN – Input From Port</b>	
fixed port	1110 010w : port number
variable port	1110 110w
<b>INC – Increment by 1</b>	
reg	0100 000B 1111 111w : 11 000 reg
qwordreg	0100 100B 1111 1111 : 11 000 qwordreg
memory	0100 00XB 1111 111w : mod 000 r/m
memory64	0100 10XB 1111 1111 : mod 000 r/m
<b>INS – Input from DX Port</b>	0110 110w
<b>INT n – Interrupt Type n</b>	1100 1101 : type
<b>INT – Single-Step Interrupt 3</b>	1100 1100
<b>INTO – Interrupt 4 on Overflow</b>	1100 1110
<b>INVD – Invalidate Cache</b>	0000 1111 : 0000 1000
<b>INVLPG – Invalidate TLB Entry</b>	0000 1111 : 0000 0001 : mod 111 r/m
<b>IRETO – Interrupt Return</b>	1100 1111
<b>Jcc – Jump if Condition is Met</b>	
8-bit displacement	0111 ttn : 8-bit displacement
displacements (excluding 16-bit relative offsets)	0000 1111 : 1000 ttn : displacement32
<b>JCXZ/JECXZ – Jump on CX/ECX Zero</b> Address-size prefix differentiates JCXZ and JECXZ	1110 0011 : 8-bit displacement
<b>JMP – Unconditional Jump (to same segment)</b>	
short	1110 1011 : 8-bit displacement
direct	1110 1001 : displacement32

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
register indirect	0100 W00B <sup>W</sup> : 1111 1111 : 11 100 reg
memory indirect	0100 W0XB <sup>W</sup> : 1111 1111 : mod 100 r/m
<b>JMP – Unconditional Jump (to other segment)</b>	
indirect intersegment	0100 00XB : 1111 1111 : mod 101 r/m
64-bit indirect intersegment	0100 10XB : 1111 1111 : mod 101 r/m
<b>LAR – Load Access Rights Byte</b>	
from register	0100 0R0B : 0000 1111 : 0000 0010 : 11 reg1 reg2
from dwordregister to qwordregister, masked by 00FxFF00H	0100 WR0B : 0000 1111 : 0000 0010 : 11 qwordreg1 dwordreg2
from memory	0100 0RXB : 0000 1111 : 0000 0010 : mod reg r/m
from memory32 to qwordregister, masked by 00FxFF00H	0100 WRXB 0000 1111 : 0000 0010 : mod r/m
<b>LEA – Load Effective Address</b>	
in wordregister/dwordregister	0100 0RXB : 1000 1101 : mod <sup>A</sup> reg r/m
in qwordregister	0100 1RXB : 1000 1101 : mod <sup>A</sup> qwordreg r/m
<b>LEAVE – High Level Procedure Exit</b>	
<b>LFS – Load Pointer to FS</b>	
FS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0100 : mod <sup>A</sup> reg r/m
FS:r64 with far pointer from memory	0100 1RXB : 0000 1111 : 1011 0100 : mod <sup>A</sup> qwordreg r/m
<b>LGDT – Load Global Descriptor Table Register</b>	
0100 10XB : 0000 1111 : 0000 0001 : mod <sup>A</sup> 010 r/m	
<b>LGS – Load Pointer to GS</b>	
GS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0101 : mod <sup>A</sup> reg r/m
GS:r64 with far pointer from memory	0100 1RXB : 0000 1111 : 1011 0101 : mod <sup>A</sup> qwordreg r/m
<b>LIDT – Load Interrupt Descriptor Table Register</b>	
0100 10XB : 0000 1111 : 0000 0001 : mod <sup>A</sup> 011 r/m	
<b>LLDT – Load Local Descriptor Table Register</b>	
LDTR from register	0100 000B : 0000 1111 : 0000 0000 : 11 010 reg
LDTR from memory	0100 00XB : 0000 1111 : 0000 0000 : mod 010 r/m

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<p><b>LMSW – Load Machine Status Word</b></p> <p>from register</p> <p>from memory</p>	<p>0100 000B : 0000 1111 : 0000 0001 : 11 110 reg</p> <p>0100 00XB : 0000 1111 : 0000 0001 : mod 110 r/m</p>
<p><b>LOCK – Assert LOCK# Signal Prefix</b></p>	<p>1111 0000</p>
<p><b>LODS/LODSB/LODSW/LODSD/LODSQ – Load String Operand</b></p> <p>at DS:(E)SI to AL/EAX/EAX</p> <p>at (R)SI to RAX</p>	<p>1010 110w</p> <p>0100 1000 1010 1101</p>
<p><b>LOOP – Loop Count</b></p> <p>if count != 0, 8-bit displacement</p> <p>if count !=0, RIP + 8-bit displacement sign-extended to 64-bits</p>	<p>1110 0010</p> <p>0100 1000 1110 0010</p>
<p><b>LOOPE – Loop Count while Zero/Equal</b></p> <p>if count != 0 &amp; ZF =1, 8-bit displacement</p> <p>if count !=0 &amp; ZF = 1, RIP + 8-bit displacement sign-extended to 64-bits</p>	<p>1110 0001</p> <p>0100 1000 1110 0001</p>
<p><b>LOOPNE/LOOPNZ – Loop Count while not Zero/Equal</b></p> <p>if count != 0 &amp; ZF = 0, 8-bit displacement</p> <p>if count !=0 &amp; ZF = 0, RIP + 8-bit displacement sign-extended to 64-bits</p>	<p>1110 0000</p> <p>0100 1000 1110 0000</p>
<p><b>LSL – Load Segment Limit</b></p> <p>from register</p> <p>from qwordregister</p> <p>from memory16</p> <p>from memory64</p>	<p>0000 1111 : 0000 0011 : 11 reg1 reg2</p> <p>0100 1R00 0000 1111 : 0000 0011 : 11 qwordreg1 reg2</p> <p>0000 1111 : 0000 0011 : mod reg r/m</p> <p>0100 1RXB 0000 1111 : 0000 0011 : mod qwordreg r/m</p>
<p><b>LSS – Load Pointer to SS</b></p> <p>SS:r16/r32 with far pointer from memory</p> <p>SS:r64 with far pointer from memory</p>	<p>0100 0RXB : 0000 1111 : 1011 0010 : mod<sup>A</sup> reg r/m</p> <p>0100 1WXB : 0000 1111 : 1011 0010 : mod<sup>A</sup> qwordreg r/m</p>

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>LTR – Load Task Register</b> from register from memory	0100 0R00 : 0000 1111 : 0000 0000 : 11 011 reg 0100 00XB : 0000 1111 : 0000 0000 : mod 011 r/m
<b>MOV – Move Data</b> register1 to register2 qwordregister1 to qwordregister2 register2 to register1 qwordregister2 to qwordregister1 memory to reg memory64 to qwordregister reg to memory qwordregister to memory64 immediate to register immediate32 to qwordregister (zero extend) immediate to register (alternate encoding) immediate64 to qwordregister (alternate encoding) immediate to memory immediate32 to memory64 (zero extend) memory to AL, AX, or EAX memory64 to RAX AL, AX, or EAX to memory RAX to memory64	0100 0R0B : 1000 100w : 11 reg1 reg2 0100 1R0B 1000 1001 : 11 qwordreg1 qwordreg2 0100 0R0B : 1000 101w : 11 reg1 reg2 0100 1R0B 1000 1011 : 11 qwordreg1 qwordreg2 0100 0RXB : 1000 101w : mod reg r/m 0100 1RXB 1000 1011 : mod qwordreg r/m 0100 0RXB : 1000 100w : mod reg r/m 0100 1RXB 1000 1001 : mod qwordreg r/m 0100 000B : 1100 011w : 11 000 reg : imm 0100 100B 1100 0111 : 11 000 qwordreg : imm32 0100 000B : 1011 w reg : imm 0100 100B 1011 1000 reg : imm64 0100 00XB : 1100 011w : mod 000 r/m : imm 0100 10XB 1100 0111 : mod 000 r/m : imm32 0100 0000 : 1010 000w : displacement 0100 1000 1010 0001 : displacement64 0100 0000 : 1010 001w : displacement 0100 1000 1010 0011 : displacement64
<b>MOV – Move to/from Control Registers</b> CR0-CR4 from register CRx from qwordregister register from CR0-CR4 qwordregister from CRx	0100 0R0B : 0000 1111 : 0010 0010 : 11 eee reg (eee = CR#) 0100 1R0B : 0000 1111 : 0010 0010 : 11 eee qwordreg (Reee = CR#) 0100 0R0B : 0000 1111 : 0010 0000 : 11 eee reg (eee = CR#) 0100 1R0B 0000 1111 : 0010 0000 : 11 eee qwordreg (Reee = CR#)

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>MOV – Move to/from Debug Registers</b>	
DR0-DR7 from register	0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)
DR0-DR7 from quadregister	0100 100B 0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)
register from DR0-DR7	0000 1111 : 0010 0001 : 11 eee reg (eee = DR#)
quadregister from DR0-DR7	0100 100B 0000 1111 : 0010 0001 : 11 eee quadreg (eee = DR#)
<b>MOV – Move to/from Segment Registers</b>	
register to segment register	0100 W00B <sup>w</sup> : 1000 1110 : 11 sreg reg
register to SS	0100 000B : 1000 1110 : 11 sreg reg
memory to segment register	0100 00XB : 1000 1110 : mod sreg r/m
memory64 to segment register (lower 16 bits)	0100 10XB 1000 1110 : mod sreg r/m
memory to SS	0100 00XB : 1000 1110 : mod sreg r/m
segment register to register	0100 000B : 1000 1100 : 11 sreg reg
segment register to qwordregister (zero extended)	0100 100B 1000 1100 : 11 sreg qwordreg
segment register to memory	0100 00XB : 1000 1100 : mod sreg r/m
segment register to memory64 (zero extended)	0100 10XB 1000 1100 : mod sreg3 r/m
<b>MOVS/MOVSb/MOVSsw/MOVSd/MOVSq – Move Data from String to String</b>	
Move data from string to string	1010 010w
Move data from string to string (qword)	0100 1000 1010 0101
<b>MOVsx/MOVsxd – Move with Sign-Extend</b>	
register2 to register1	0100 0R0B : 0000 1111 : 1011 111w : 11 reg1 reg2
byteregister2 to qwordregister1 (sign-extend)	0100 1R0B 0000 1111 : 1011 1110 : 11 quadreg1 bytereg2
wordregister2 to qwordregister1	0100 1R0B 0000 1111 : 1011 1111 : 11 quadreg1 wordreg2
dwordregister2 to qwordregister1	0100 1R0B 0110 0011 : 11 quadreg1 dwordreg2
memory to register	0100 0RXB : 0000 1111 : 1011 111w : mod reg r/m
memory8 to qwordregister (sign-extend)	0100 1RXB 0000 1111 : 1011 1110 : mod qwordreg r/m
memory16 to qwordregister	0100 1RXB 0000 1111 : 1011 1111 : mod qwordreg r/m
memory32 to qwordregister	0100 1RXB 0110 0011 : mod qwordreg r/m

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>MOVZX – Move with Zero-Extend</b>	
register2 to register1	0100 0R0B : 0000 1111 : 1011 011w : 11 reg1 reg2
dwordregister2 to qwordregister1	0100 1R0B 0000 1111 : 1011 0111 : 11 qwordreg1 dwordreg2
memory to register	0100 0R0B : 0000 1111 : 1011 011w : mod reg r/m
memory32 to qwordregister	0100 1R0B 0000 1111 : 1011 0111 : mod qwordreg r/m
<b>MUL – Unsigned Multiply</b>	
AL, AX, or EAX with register	0100 000B : 1111 011w : 11 100 reg
RAX with qwordregister (to RDX:RAX)	0100 100B 1111 0111 : 11 100 qwordreg
AL, AX, or EAX with memory	0100 00XB 1111 011w : mod 100 r/m
RAX with memory64 (to RDX:RAX)	0100 10XB 1111 0111 : mod 100 r/m
<b>NEG – Two's Complement Negation</b>	
register	0100 000B : 1111 011w : 11 011 reg
qwordregister	0100 100B 1111 0111 : 11 011 qwordreg
memory	0100 00XB : 1111 011w : mod 011 r/m
memory64	0100 10XB 1111 0111 : mod 011 r/m
<b>NOP – No Operation</b>	
	1001 0000
<b>NOT – One's Complement Negation</b>	
register	0100 000B : 1111 011w : 11 010 reg
qwordregister	0100 000B 1111 0111 : 11 010 qwordreg
memory	0100 00XB : 1111 011w : mod 010 r/m
memory64	0100 1RXB 1111 0111 : mod 010 r/m
<b>OR – Logical Inclusive OR</b>	
register1 to register2	0000 100w : 11 reg1 reg2
byteregister1 to byteregister2	0100 0R0B 0000 1000 : 11 bytereg1 bytereg2
qwordregister1 to qwordregister2	0100 1R0B 0000 1001 : 11 qwordreg1 qwordreg2
register2 to register1	0000 101w : 11 reg1 reg2
byteregister2 to byteregister1	0100 0R0B 0000 1010 : 11 bytereg1 bytereg2
qwordregister2 to qwordregister1	0100 0R0B 0000 1011 : 11 qwordreg1 qwordreg2
memory to register	0000 101w : mod reg r/m
memory8 to byteregister	0100 0RXB 0000 1010 : mod bytereg r/m
memory8 to qwordregister	0100 0RXB 0000 1011 : mod qwordreg r/m



## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
register to memory	0000 100w : mod reg r/m
byte register to memory8	0100 0RXB 0000 1000 : mod bytereg r/m
qword register to memory64	0100 1RXB 0000 1001 : mod qwordreg r/m
immediate to register	1000 00sw : 11 001 reg : imm
immediate8 to byte register	0100 000B 1000 0000 : 11 001 bytereg : imm8
immediate32 to qword register	0100 000B 1000 0001 : 11 001 qwordreg : imm32
immediate8 to qword register	0100 000B 1000 0011 : 11 001 qwordreg : imm8
immediate to AL, AX, or EAX	0000 110w : imm
immediate64 to RAX	0100 1000 0000 1101 : imm64
immediate to memory	1000 00sw : mod 001 r/m : imm
immediate8 to memory8	0100 00XB 1000 0000 : mod 001 r/m : imm8
immediate32 to memory64	0100 00XB 1000 0001 : mod 001 r/m : imm32
immediate8 to memory64	0100 00XB 1000 0011 : mod 001 r/m : imm8
<b>OUT – Output to Port</b>	
fixed port	1110 011w : port number
variable port	1110 111w
<b>OUTS – Output to DX Port</b>	
output to DX Port	0110 111w
<b>POP – Pop a Value from the Stack</b>	
word register	0101 0101 : 0100 000B : 1000 1111 : 11 000 reg16
qword register	0100 W00B <sup>S</sup> : 1000 1111 : 11 000 reg64
word register (alternate encoding)	0101 0101 : 0100 000B : 0101 1 reg16
qword register (alternate encoding)	0100 W00B : 0101 1 reg64
memory64	0100 W0XB <sup>S</sup> : 1000 1111 : mod 000 r/m
memory16	0101 0101 : 0100 00XB 1000 1111 : mod 000 r/m
<b>POP – Pop a Segment Register from the Stack</b> (Note: CS cannot be sreg2 in this usage.)	
segment register FS, GS	0000 1111: 10 sreg3 001
<b>POPF/POPFQ – Pop Stack into FLAGS/RFLAGS Register</b>	
pop stack to FLAGS register	0101 0101 : 1001 1101
pop Stack to RFLAGS register	0100 1000 1001 1101

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>PUSH – Push Operand onto the Stack</b>	
wordregister	0101 0101 : 0100 000B : 1111 1111 : 11 110 reg16
qwordregister	0100 W00B <sup>S</sup> : 1111 1111 : 11 110 reg64
wordregister (alternate encoding)	0101 0101 : 0100 000B : 0101 0 reg16
qwordregister (alternate encoding)	0100 W00B <sup>S</sup> : 0101 0 reg64
memory16	0101 0101 : 0100 000B : 1111 1111 : mod 110 r/m
memory64	0100 W00B <sup>S</sup> : 1111 1111 : mod 110 r/m
immediate8	0110 1010 : imm8
immediate16	0101 0101 : 0110 1000 : imm16
immediate64	0110 1000 : imm64
<b>PUSH – Push Segment Register onto the Stack</b>	
segment register FS,GS	0000 1111: 10 sreg3 000
<b>PUSHF/PUSHFD – Push Flags Register onto the Stack</b>	
	1001 1100
<b>RCL – Rotate thru Carry Left</b>	
register by 1	0100 000B : 1101 000w : 11 010 reg
qwordregister by 1	0100 100B 1101 0001 : 11 010 qwordreg
memory by 1	0100 00XB : 1101 000w : mod 010 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 010 r/m
register by CL	0100 000B : 1101 001w : 11 010 reg
qwordregister by CL	0100 100B 1101 0011 : 11 010 qwordreg
memory by CL	0100 00XB : 1101 001w : mod 010 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 010 r/m
register by immediate count	0100 000B : 1100 000w : 11 010 reg : imm
qwordregister by immediate count	0100 100B 1100 0001 : 11 010 qwordreg : imm8
memory by immediate count	0100 00XB : 1100 000w : mod 010 r/m : imm
memory64 by immediate count	0100 10XB 1100 0001 : mod 010 r/m : imm8
<b>RCR – Rotate thru Carry Right</b>	
register by 1	0100 000B : 1101 000w : 11 011 reg
qwordregister by 1	0100 100B 1101 0001 : 11 011 qwordreg
memory by 1	0100 00XB : 1101 000w : mod 011 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 011 r/m

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
register by CL	0100 000B : 1101 001w : 11 011 reg
qwordregister by CL	0100 000B 1101 0010 : 11 011 qwordreg
memory by CL	0100 00XB : 1101 001w : mod 011 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 011 r/m
register by immediate count	0100 000B : 1100 000w : 11 011 reg : imm8
qwordregister by immediate count	0100 100B 1100 0001 : 11 011 qwordreg : imm8
memory by immediate count	0100 00XB : 1100 000w : mod 011 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 011 r/m : imm8
<b>RDMSR – Read from Model-Specific Register</b>	
load ECX-specified register into EDX:EAX	0000 1111 : 0011 0010
<b>RDPNC – Read Performance Monitoring Counters</b>	
load ECX-specified performance counter into EDX:EAX	0000 1111 : 0011 0011
<b>RDTSC – Read Time-Stamp Counter</b>	
read time-stamp counter into EDX:EAX	0000 1111 : 0011 0001
<b>REP INS – Input String</b>	
<b>REP LODS – Load String</b>	
<b>REP MOVS – Move String</b>	
<b>REP OUTS – Output String</b>	
<b>REP STOS – Store String</b>	
<b>REPE CMPS – Compare String</b>	
<b>REPE SCAS – Scan String</b>	
<b>REPNE CMPS – Compare String</b>	
<b>REPNE SCAS – Scan String</b>	
<b>RET – Return from Procedure (to same segment)</b>	
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
<b>RET – Return from Procedure (to other segment)</b>	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>ROL – Rotate Left</b>	
register by 1	0100 000B 1101 000w : 11 000 reg
byteregister by 1	0100 000B 1101 0000 : 11 000 bytereg
qwordregister by 1	0100 100B 1101 0001 : 11 000 qwordreg
memory by 1	0100 00XB 1101 000w : mod 000 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 000 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 000 r/m
register by CL	0100 000B 1101 001w : 11 000 reg
byteregister by CL	0100 000B 1101 0010 : 11 000 bytereg
qwordregister by CL	0100 100B 1101 0011 : 11 000 qwordreg
memory by CL	0100 00XB 1101 001w : mod 000 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 000 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 000 r/m
register by immediate count	1100 000w : 11 000 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 000 bytereg : imm8
qwordregister by immediate count	0100 100B 1100 0001 : 11 000 bytereg : imm8
memory by immediate count	1100 000w : mod 000 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 000 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 000 r/m : imm8
<b>ROR – Rotate Right</b>	
register by 1	0100 000B 1101 000w : 11 001 reg
byteregister by 1	0100 000B 1101 0000 : 11 001 bytereg
qwordregister by 1	0100 100B 1101 0001 : 11 001 qwordreg
memory by 1	0100 00XB 1101 000w : mod 001 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 001 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 001 r/m
register by CL	0100 000B 1101 001w : 11 001 reg
byteregister by CL	0100 000B 1101 0010 : 11 001 bytereg
qwordregister by CL	0100 100B 1101 0011 : 11 001 qwordreg
memory by CL	0100 00XB 1101 001w : mod 001 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 001 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 001 r/m

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
register by immediate count	0100 000B 1100 000w : 11 001 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 001 reg : imm8
qwordregister by immediate count	0100 100B 1100 0001 : 11 001 qwordreg : imm8
memory by immediate count	0100 00XB 1100 000w : mod 001 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 001 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 001 r/m : imm8
<b>RSM – Resume from System Management Mode</b>	0000 1111 : 1010 1010
<b>SAL – Shift Arithmetic Left</b>	same instruction as SHL
<b>SAR – Shift Arithmetic Right</b>	
register by 1	0100 000B 1101 000w : 11 111 reg
byteregister by 1	0100 000B 1101 0000 : 11 111 bytereg
qwordregister by 1	0100 100B 1101 0001 : 11 111 qwordreg
memory by 1	0100 00XB 1101 000w : mod 111 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 111 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 111 r/m
register by CL	0100 000B 1101 001w : 11 111 reg
byteregister by CL	0100 000B 1101 0010 : 11 111 bytereg
qwordregister by CL	0100 100B 1101 0011 : 11 111 qwordreg
memory by CL	0100 00XB 1101 001w : mod 111 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 111 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 111 r/m
register by immediate count	0100 000B 1100 000w : 11 111 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 111 bytereg : imm8
qwordregister by immediate count	0100 100B 1100 0001 : 11 111 qwordreg : imm8
memory by immediate count	0100 00XB 1100 000w : mod 111 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 111 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 111 r/m : imm8
<b>SBB – Integer Subtraction with Borrow</b>	
register1 to register2	0100 0R0B 0001 100w : 11 reg1 reg2
byteregister1 to byteregister2	0100 0R0B 0001 1000 : 11 bytereg1 bytereg2
quadregister1 to quadregister2	0100 1R0B 0001 1001 : 11 quadreg1 quadreg2

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
register2 to register1	0100 0R0B 0001 101w : 11 reg1 reg2
byteregister2 to byteregister1	0100 0R0B 0001 1010 : 11 reg1 bytereg2
byteregister2 to byteregister1	0100 1R0B 0001 1011 : 11 reg1 bytereg2
memory to register	0100 0RXB 0001 101w : mod reg r/m
memory8 to byteregister	0100 0RXB 0001 1010 : mod bytereg r/m
memory64 to byteregister	0100 1RXB 0001 1011 : mod quadreg r/m
register to memory	0100 0RXB 0001 100w : mod reg r/m
byteregister to memory8	0100 0RXB 0001 1000 : mod reg r/m
quadregister to memory64	0100 1RXB 0001 1001 : mod reg r/m
immediate to register	0100 000B 1000 00sw : 11 011 reg : imm
immediate8 to byteregister	0100 000B 1000 0000 : 11 011 bytereg : imm8
immediate32 to qwordregister	0100 100B 1000 0001 : 11 011 qwordreg : imm32
immediate8 to qwordregister	0100 100B 1000 0011 : 11 011 qwordreg : imm8
immediate to AL, AX, or EAX	0100 000B 0001 110w : imm
immediate32 to RAL	0100 1000 0001 1101 : imm32
immediate to memory	0100 00XB 1000 00sw : mod 011 r/m : imm
immediate8 to memory8	0100 00XB 1000 0000 : mod 011 r/m : imm8
immediate32 to memory64	0100 10XB 1000 0001 : mod 011 r/m : imm32
immediate8 to memory64	0100 10XB 1000 0011 : mod 011 r/m : imm8
<b>SCAS/SCASB/SCASW/SCASD – Scan String</b>	
scan string	1010 111w
scan string (compare AL with byte at RDI)	0100 1000 1010 1110
scan string (compare RAX with qword at RDI)	0100 1000 1010 1111
<b>SETcc – Byte Set on Condition</b>	
register	0100 000B 0000 1111 : 1001 ttn : 11 000 reg
register	0100 0000 0000 1111 : 1001 ttn : 11 000 reg
memory	0100 00XB 0000 1111 : 1001 ttn : mod 000 r/m
memory	0100 0000 0000 1111 : 1001 ttn : mod 000 r/m
<b>SGDT – Store Global Descriptor Table Register</b>	
	0000 1111 : 0000 0001 : mod <sup>A</sup> 000 r/m
<b>SHL – Shift Left</b>	
register by 1	0100 000B 1101 000w : 11 100 reg
byteregister by 1	0100 000B 1101 0000 : 11 100 bytereg

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
qwordregister by 1	0100 100B 1101 0001 : 11 100 qwordreg
memory by 1	0100 00XB 1101 000w : mod 100 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 100 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 100 r/m
register by CL	0100 000B 1101 001w : 11 100 reg
byteregister by CL	0100 000B 1101 0010 : 11 100 bytereg
qwordregister by CL	0100 100B 1101 0011 : 11 100 qwordreg
memory by CL	0100 00XB 1101 001w : mod 100 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 100 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 100 r/m
register by immediate count	0100 000B 1100 000w : 11 100 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 100 bytereg : imm8
quadregister by immediate count	0100 100B 1100 0001 : 11 100 quadreg : imm8
memory by immediate count	0100 00XB 1100 000w : mod 100 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 100 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 100 r/m : imm8
<b>SHLD – Double Precision Shift Left</b>	
register by immediate count	0100 0R0B 0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8
qwordregister by immediate8	0100 1R0B 0000 1111 : 1010 0100 : 11 qwordreg2 qwordreg1 : imm8
memory by immediate count	0100 0RXB 0000 1111 : 1010 0100 : mod reg r/m : imm8
memory64 by immediate8	0100 1RXB 0000 1111 : 1010 0100 : mod qwordreg r/m : imm8
register by CL	0100 0R0B 0000 1111 : 1010 0101 : 11 reg2 reg1
quadregister by CL	0100 1R0B 0000 1111 : 1010 0101 : 11 quadreg2 quadreg1
memory by CL	0100 00XB 0000 1111 : 1010 0101 : mod reg r/m
memory64 by CL	0100 1RXB 0000 1111 : 1010 0101 : mod quadreg r/m
<b>SHR – Shift Right</b>	
register by 1	0100 000B 1101 000w : 11 101 reg
byteregister by 1	0100 000B 1101 0000 : 11 101 bytereg

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
qwordregister by 1	0100 100B 1101 0001 : 11 101 qwordreg
memory by 1	0100 00XB 1101 000w : mod 101 r/m
memory8 by 1	0100 00XB 1101 0000 : mod 101 r/m
memory64 by 1	0100 10XB 1101 0001 : mod 101 r/m
register by CL	0100 000B 1101 001w : 11 101 reg
byteregister by CL	0100 000B 1101 0010 : 11 101 bytereg
qwordregister by CL	0100 100B 1101 0011 : 11 101 qwordreg
memory by CL	0100 00XB 1101 001w : mod 101 r/m
memory8 by CL	0100 00XB 1101 0010 : mod 101 r/m
memory64 by CL	0100 10XB 1101 0011 : mod 101 r/m
register by immediate count	0100 000B 1100 000w : 11 101 reg : imm8
byteregister by immediate count	0100 000B 1100 0000 : 11 101 reg : imm8
qwordregister by immediate count	0100 100B 1100 0001 : 11 101 reg : imm8
memory by immediate count	0100 00XB 1100 000w : mod 101 r/m : imm8
memory8 by immediate count	0100 00XB 1100 0000 : mod 101 r/m : imm8
memory64 by immediate count	0100 10XB 1100 0001 : mod 101 r/m : imm8
<b>SHRD – Double Precision Shift Right</b>	
register by immediate count	0100 0R0B 0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8
qwordregister by immediate8	0100 1R0B 0000 1111 : 1010 1100 : 11 qwordreg2 qwordreg1 : imm8
memory by immediate count	0100 00XB 0000 1111 : 1010 1100 : mod reg r/m : imm8
memory64 by immediate8	0100 1RXB 0000 1111 : 1010 1100 : mod qwordreg r/m : imm8
register by CL	0100 000B 0000 1111 : 1010 1101 : 11 reg2 reg1
qwordregister by CL	0100 1R0B 0000 1111 : 1010 1101 : 11 qwordreg2 qwordreg1
memory by CL	0000 1111 : 1010 1101 : mod reg r/m
memory64 by CL	0100 1RXB 0000 1111 : 1010 1101 : mod qwordreg r/m
<b>SIDT – Store Interrupt Descriptor Table Register</b>	0000 1111 : 0000 0001 : mod <sup>A</sup> 001 r/m



## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>SLDT – Store Local Descriptor Table Register</b>	
to register	0100 000B 0000 1111 : 0000 0000 : 11 000 reg
to memory	0100 00XB 0000 1111 : 0000 0000 : mod 000 r/m
<b>SMSW – Store Machine Status Word</b>	
to register	0100 000B 0000 1111 : 0000 0001 : 11 100 reg
to memory	0100 00XB 0000 1111 : 0000 0001 : mod 100 r/m
<b>STC – Set Carry Flag</b>	1111 1001
<b>STD – Set Direction Flag</b>	1111 1101
<b>STI – Set Interrupt Flag</b>	1111 1011
<b>STOS/STOSB/STOSW/STOSD/STOSQ – Store String Data</b>	
store string data	1010 101w
store string data (RAX at address RDI)	0100 1000 1010 1011
<b>STR – Store Task Register</b>	
to register	0100 000B 0000 1111 : 0000 0000 : 11 001 reg
to memory	0100 00XB 0000 1111 : 0000 0000 : mod 001 r/m
<b>SUB – Integer Subtraction</b>	
register1 from register2	0100 0R0B 0010 100w : 11 reg1 reg2
byteregister1 from byteregister2	0100 0R0B 0010 1000 : 11 bytereg1 bytereg2
qwordregister1 from qwordregister2	0100 1R0B 0010 1000 : 11 qwordreg1 qwordreg2
register2 from register1	0100 0R0B 0010 101w : 11 reg1 reg2
byteregister2 from byteregister1	0100 0R0B 0010 1010 : 11 bytereg1 bytereg2
qwordregister2 from qwordregister1	0100 1R0B 0010 1011 : 11 qwordreg1 qwordreg2
memory from register	0100 00XB 0010 101w : mod reg r/m
memory8 from byteregister	0100 0RXB 0010 1010 : mod bytereg r/m
memory64 from qwordregister	0100 1RXB 0010 1011 : mod qwordreg r/m
register from memory	0100 0RXB 0010 100w : mod reg r/m
byteregister from memory8	0100 0RXB 0010 1000 : mod bytereg r/m
qwordregister from memory8	0100 1RXB 0010 1000 : mod qwordreg r/m
immediate from register	0100 000B 1000 00sw : 11 101 reg : imm
immediate8 from byteregister	0100 000B 1000 0000 : 11 101 bytereg : imm8

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
immediate32 from qwordregister	0100 100B 1000 0001 : 11 101 qwordreg : imm32
immediate8 from qwordregister	0100 100B 1000 0011 : 11 101 qwordreg : imm8
immediate from AL, AX, or EAX	0100 000B 0010 110w : imm
immediate32 from RAX	0100 1000 0010 1101 : imm32
immediate from memory	0100 00XB 1000 00sw : mod 101 r/m : imm
immediate8 from memory8	0100 00XB 1000 0000 : mod 101 r/m : imm8
immediate32 from memory64	0100 10XB 1000 0001 : mod 101 r/m : imm32
immediate8 from memory64	0100 10XB 1000 0011 : mod 101 r/m : imm8
<b>SWAPGS – Swap GS Base Register</b>	
GS base register value for value in MSR C0000102H	0000 1111 0000 0001 [this one incomplete]
<b>SYSCALL – Fast System Call</b>	
fast call to privilege level 0 system procedures	0000 1111 0000 0101
<b>SYSRET – Return From Fast System Call</b>	
return from fast system call	0000 1111 0000 0111
<b>TEST – Logical Compare</b>	
register1 and register2	0100 0R0B 1000 010w : 11 reg1 reg2
byteregister1 and byteregister2	0100 0R0B 1000 0100 : 11 bytereg1 bytereg2
qwordregister1 and qwordregister2	0100 1R0B 1000 0101 : 11 qwordreg1 qwordreg2
memory and register	0100 0R0B 1000 010w : mod reg r/m
memory8 and byteregister	0100 0RXB 1000 0100 : mod bytereg r/m
memory64 and qwordregister	0100 1RXB 1000 0101 : mod qwordreg r/m
immediate and register	0100 000B 1111 011w : 11 000 reg : imm
immediate8 and byteregister	0100 000B 1111 0110 : 11 000 bytereg : imm8
immediate32 and qwordregister	0100 100B 1111 0111 : 11 000 bytereg : imm8
immediate and AL, AX, or EAX	0100 000B 1010 100w : imm
immediate32 and RAX	0100 1000 1010 1001 : imm32
immediate and memory	0100 00XB 1111 011w : mod 000 r/m : imm
immediate8 and memory8	0100 1000 1111 0110 : mod 000 r/m : imm8
immediate32 and memory64	0100 1000 1111 0111 : mod 000 r/m : imm32
<b>UD2 – Undefined instruction</b>	
	0000 FFFF : 0000 1011

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
<b>VERR – Verify a Segment for Reading</b>	
register	0100 000B 0000 1111 : 0000 0000 : 11 100 reg
memory	0100 00XB 0000 1111 : 0000 0000 : mod 100 r/m
<b>VERW – Verify a Segment for Writing</b>	
register	0100 000B 0000 1111 : 0000 0000 : 11 101 reg
memory	0100 00XB 0000 1111 : 0000 0000 : mod 101 r/m
<b>WAIT – Wait</b>	1001 1011
<b>WBINVD – Writeback and Invalidate Data Cache</b>	0000 1111 : 0000 1001
<b>WRMSR – Write to Model-Specific Register</b>	
write EDX:EAX to ECX specified MSR	0000 1111 : 0011 0000
write RDX[31:0]:RAX[31:0] to RCX specified MSR	0100 1000 0000 1111 : 0011 0000
<b>XADD – Exchange and Add</b>	
register1, register2	0100 0R0B 0000 1111 : 1100 000w : 11 reg2 reg1
byteregister1, byteregister2	0100 0R0B 0000 1111 : 1100 0000 : 11 bytereg2 bytereg1
qwordregister1, qwordregister2	0100 0R0B 0000 1111 : 1100 0001 : 11 qwordreg2 qwordreg1
memory, register	0100 0RXB 0000 1111 : 1100 000w : mod reg r/m
memory8, bytereg	0100 1RXB 0000 1111 : 1100 0000 : mod bytereg r/m
memory64, qwordreg	0100 1RXB 0000 1111 : 1100 0001 : mod qwordreg r/m
<b>XCHG – Exchange Register/Memory with Register</b>	
register1 with register2	1000 011w : 11 reg1 reg2
AX or EAX with register	1001 0 reg
memory with register	1000 011w : mod reg r/m
<b>XLAT/XLATB – Table Look-up Translation</b>	
AL to byte DS:[(E)BX + unsigned AL]	1101 0111
AL to byte DS:[RBX + unsigned AL]	0100 1000 1101 0111
<b>XOR – Logical Exclusive OR</b>	
register1 to register2	0100 0RXB 0011 000w : 11 reg1 reg2
byteregister1 to byteregister2	0100 0R0B 0011 0000 : 11 bytereg1 bytereg2
qwordregister1 to qwordregister2	0100 1R0B 0011 0001 : 11 qwordreg1 qwordreg2

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-15. General Purpose Instruction Formats and Encodings  
for 64-Bit Mode (Contd.)**

Instruction and Format	Encoding
register2 to register1	0100 0R0B 0011 001w : 11 reg1 reg2
byteregister2 to byteregister1	0100 0R0B 0011 0010 : 11 bytereg1 bytereg2
qwordregister2 to qwordregister1	0100 1R0B 0011 0011 : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB 0011 001w : mod reg r/m
memory8 to byteregister	0100 0RXB 0011 0010 : mod bytereg r/m
memory64 to qwordregister	0100 1RXB 0011 0011 : mod qwordreg r/m
register to memory	0100 0RXB 0011 000w : mod reg r/m
byteregister to memory8	0100 0RXB 0011 0000 : mod bytereg r/m
qwordregister to memory8	0100 1RXB 0011 0001 : mod qwordreg r/m
immediate to register	0100 000B 1000 00sw : 11 110 reg : imm
immediate8 to byteregister	0100 000B 1000 0000 : 11 110 bytereg : imm8
immediate32 to qwordregister	0100 100B 1000 0001 : 11 110 qwordreg : imm32
immediate8 to qwordregister	0100 100B 1000 0011 : 11 110 qwordreg : imm8
immediate to AL, AX, or EAX	0100 000B 0011 010w : imm
immediate to RAX	0100 1000 0011 0101 : immediate data
immediate to memory	0100 00XB 1000 00sw : mod 110 r/m : imm
immediate8 to memory8	0100 00XB 1000 0000 : mod 110 r/m : imm8
immediate32 to memory64	0100 10XB 1000 0001 : mod 110 r/m : imm32
immediate8 to memory64	0100 10XB 1000 0011 : mod 110 r/m : imm8
<b>Prefix Bytes</b>	
address size	0110 0111
LOCK	1111 0000
operand size	0110 0110
CS segment override	0010 1110
DS segment override	0011 1110
ES segment override	0010 0110
FS segment override	0110 0100
GS segment override	0110 0101
SS segment override	0011 0110

### B.3 PENTIUM® PROCESSOR FAMILY INSTRUCTION FORMATS AND ENCODINGS

The following table shows formats and encodings introduced by the Pentium processor family.

**Table B-16. Pentium Processor Family Instruction Formats and Encodings, Non-64-Bit Modes**

Instruction and Format	Encoding
<b>CMPXCHG8B – Compare and Exchange 8 Bytes</b> EDX:EAX with memory64	0000 1111 : 1100 0111 : mod 001 r/m

**Table B-17. Pentium Processor Family Instruction Formats and Encodings, 64-Bit Mode**

Instruction and Format	Encoding
<b>CMPXCHG8B/CMPXCHG16B – Compare and Exchange Bytes</b> EDX:EAX with memory64 RDX:RAX with memory128	0000 1111 : 1100 0111 : mod 001 r/m 0100 10XB 0000 1111 : 1100 0111 : mod 001 r/m

### B.4 64-BIT MODE INSTRUCTION ENCODINGS FOR SIMD INSTRUCTION EXTENSIONS

Non-64-bit mode instruction encodings for MMX Technology, SSE, SSE2, and SSE3 are covered by applying these rules to Table B-19 through Table B-30. Table B-31 lists special encodings (instructions that do not follow the rules below).

- The REX instruction has no effect:
  - On immediates
  - If both operands are MMX registers
  - On MMX registers and XMM registers
  - If an MMX register is encoded in the reg field of the ModR/M byte
- If a memory operand is encoded in the r/m field of the ModR/M byte, REX.X and REX.B may be used for encoding the memory operand.
- If a general-purpose register is encoded in the r/m field of the ModR/M byte, REX.B may be used for register encoding and REX.W may be used to encode the 64-bit operand size.
- If an XMM register operand is encoded in the reg field of the ModR/M byte, REX.R may be used for register encoding. If an XMM register operand is encoded in the r/m field of the ModR/M byte, REX.B may be used for register encoding.

## INSTRUCTION FORMATS AND ENCODINGS

### B.5 MMX INSTRUCTION FORMATS AND ENCODINGS

MMX instructions, except the EMMS instruction, use a format similar to the 2-byte Intel Architecture integer format. Details of subfield encodings within these formats are presented below.

#### B.5.1 Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-18 shows the encoding of the gg field.

**Table B-18. Encoding of Granularity of Data Field (gg)**

<b>gg</b>	<b>Granularity of Data</b>
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

#### B.5.2 MMX Technology and General-Purpose Register Fields (mmxreg and reg)

When MMX technology registers (mmxreg) are used as operands, they are encoded in the ModR/M byte in the reg field (bits 5, 4, and 3) and/or the R/M field (bits 2, 1, and 0).

If an MMX instruction operates on a general-purpose register (reg), the register is encoded in the R/M field of the ModR/M byte.

### B.5.3 MMX Instruction Formats and Encodings Table

Table B-19 shows the formats and encodings of the integer instructions.

**Table B-19. MMX Instruction Formats and Encodings**

Instruction and Format	Encoding
<b>EMMS - Empty MMX technology state</b>	0000 1111:01110111
<b>MOVD - Move doubleword</b>	
reg to mmreg	0000 1111:0110 1110: 11 mmxreg reg
reg from mmxreg	0000 1111:0111 1110: 11 mmxreg reg
mem to mmxreg	0000 1111:0110 1110: mod mmxreg r/m
mem from mmxreg	0000 1111:0111 1110: mod mmxreg r/m
<b>MOVQ - Move quadword</b>	
mmxreg2 to mmxreg1	0000 1111:0110 1111: 11 mmxreg1 mmxreg2
mmxreg2 from mmxreg1	0000 1111:0111 1111: 11 mmxreg1 mmxreg2
mem to mmxreg	0000 1111:0110 1111: mod mmxreg r/m
mem from mmxreg	0000 1111:0111 1111: mod mmxreg r/m
<b>PACKSSDW<sup>1</sup> - Pack dword to word data (signed with saturation)</b>	
mmxreg2 to mmxreg1	0000 1111:0110 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 1011: mod mmxreg r/m
<b>PACKSSWB<sup>1</sup> - Pack word to byte data (signed with saturation)</b>	
mmxreg2 to mmxreg1	0000 1111:0110 0011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0011: mod mmxreg r/m
<b>PACKUSWB<sup>1</sup> - Pack word to byte data (unsigned with saturation)</b>	
mmxreg2 to mmxreg1	0000 1111:0110 0111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0111: mod mmxreg r/m

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-19. MMX Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
<b>PADD - Add with wrap-around</b> mmxreg2 to mmxreg1 memory to mmxreg	0000 1111: 1111 11gg: 11 mmxreg1 mmxreg2 0000 1111: 1111 11gg: mod mmxreg r/m
<b>PADDS - Add signed with saturation</b> mmxreg2 to mmxreg1 memory to mmxreg	0000 1111: 1110 11gg: 11 mmxreg1 mmxreg2 0000 1111: 1110 11gg: mod mmxreg r/m
<b>PADDUS - Add unsigned with saturation</b> mmxreg2 to mmxreg1 memory to mmxreg	0000 1111: 1101 11gg: 11 mmxreg1 mmxreg2 0000 1111: 1101 11gg: mod mmxreg r/m
<b>PAND - Bitwise And</b> mmxreg2 to mmxreg1 memory to mmxreg	0000 1111:1101 1011: 11 mmxreg1 mmxreg2 0000 1111:1101 1011: mod mmxreg r/m
<b>PANDN - Bitwise AndNot</b> mmxreg2 to mmxreg1 memory to mmxreg	0000 1111:1101 1111: 11 mmxreg1 mmxreg2 0000 1111:1101 1111: mod mmxreg r/m
<b>PCMPEQ - Packed compare for equality</b> mmxreg1 with mmxreg2 mmxreg with memory	0000 1111:0111 01gg: 11 mmxreg1 mmxreg2 0000 1111:0111 01gg: mod mmxreg r/m
<b>PCMPGT - Packed compare greater (signed)</b> mmxreg1 with mmxreg2 mmxreg with memory	0000 1111:0110 01gg: 11 mmxreg1 mmxreg2 0000 1111:0110 01gg: mod mmxreg r/m
<b>PMADDWD - Packed multiply add</b> mmxreg2 to mmxreg1 memory to mmxreg	0000 1111:1111 0101: 11 mmxreg1 mmxreg2 0000 1111:1111 0101: mod mmxreg r/m
<b>PMULHUW - Packed multiplication, store high word (unsigned)</b> mmxreg2 to mmxreg1 memory to mmxreg	0000 1111: 1110 0100: 11 mmxreg1 mmxreg2 0000 1111: 1110 0100: mod mmxreg r/m
<b>PMULHW - Packed multiplication, store high word</b> mmxreg2 to mmxreg1 memory to mmxreg	0000 1111:1110 0101: 11 mmxreg1 mmxreg2 0000 1111:1110 0101: mod mmxreg r/m



## INSTRUCTION FORMATS AND ENCODINGS

**Table B-19. MMX Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
<b>PMULLW - Packed multiplication, store low word</b>	
mmxreg2 to mmxreg1	0000 1111:1101 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 0101: mod mmxreg r/m
<b>POR - Bitwise Or</b>	
mmxreg2 to mmxreg1	0000 1111:1110 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 1011: mod mmxreg r/m
<b>PSLL<sup>2</sup> - Packed shift left logical</b>	
mmxreg1 by mmxreg2	0000 1111:1111 00gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:1111 00gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:0111 00gg: 11 110 mmxreg: imm8 data
<b>PSRA<sup>2</sup> - Packed shift right arithmetic</b>	
mmxreg1 by mmxreg2	0000 1111:1110 00gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:1110 00gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:0111 00gg: 11 100 mmxreg: imm8 data
<b>PSRL<sup>2</sup> - Packed shift right logical</b>	
mmxreg1 by mmxreg2	0000 1111:1101 00gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:1101 00gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:0111 00gg: 11 010 mmxreg: imm8 data
<b>PSUB - Subtract with wrap-around</b>	
mmxreg2 from mmxreg1	0000 1111:1111 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1111 10gg: mod mmxreg r/m
<b>PSUBS - Subtract signed with saturation</b>	
mmxreg2 from mmxreg1	0000 1111:1110 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1110 10gg: mod mmxreg r/m
<b>PSUBUS - Subtract unsigned with saturation</b>	
mmxreg2 from mmxreg1	0000 1111:1101 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1101 10gg: mod mmxreg r/m
<b>PUNPCKH - Unpack high data to next larger type</b>	
mmxreg2 to mmxreg1	0000 1111:0110 10gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 10gg: mod mmxreg r/m

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-19. MMX Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
<b>PUNPCKL - Unpack low data to next larger type</b>	
mmxreg2 to mmxreg1	0000 1111:0110 00gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 00gg: mod mmxreg r/m
<b>PXOR - Bitwise Xor</b>	
mmxreg2 to mmxreg1	0000 1111:1110 1111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 1111: mod mmxreg r/m

**NOTES:**

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

## B.6 P6 FAMILY INSTRUCTION FORMATS AND ENCODINGS

Table B-20 shows the formats and encodings for several instructions that were introduced into the IA-32 architecture in the P6 family processors.

**Table B-20. Formats and Encodings of P6 Family Instructions**

Instruction and Format	Encoding
<b>CMOVcc – Conditional Move</b>	
register2 to register1	0000 1111: 0100 ttn : 11 reg1 reg2
memory to register	0000 1111 : 0100 ttn : mod reg r/m
<b>FCMOVcc – Conditional Move on EFLAG Register Condition Codes</b>	
move if below (B)	11011 010 : 11 000 ST(i)
move if equal (E)	11011 010 : 11 001 ST(i)
move if below or equal (BE)	11011 010 : 11 010 ST(i)
move if unordered (U)	11011 010 : 11 011 ST(i)
move if not below (NB)	11011 011 : 11 000 ST(i)
move if not equal (NE)	11011 011 : 11 001 ST(i)
move if not below or equal (NBE)	11011 011 : 11 010 ST(i)
move if not unordered (NU)	11011 011 : 11 011 ST(i)
<b>FCOMI – Compare Real and Set EFLAGS</b>	11011 011 : 11 110 ST(i)

**Table B-20. Formats and Encodings of P6 Family Instructions (Contd.)**

Instruction and Format	Encoding
<b>FXRSTOR</b> —Restore x87 FPU, MMX, SSE, and SSE2 State <sup>1</sup>	0000 1111:1010 1110: mod <sup>A</sup> 001 r/m
<b>FXSAVE</b> —Save x87 FPU, MMX, SSE, and SSE2 State <sup>1</sup>	0000 1111:1010 1110: mod <sup>A</sup> 000 r/m
<b>SYSENTER</b> —Fast System Call	0000 1111:0011 0100
<b>SYSEXIT</b> —Fast Return from Fast System Call	0000 1111:0011 0101

**NOTES:**

1. For FXSAVE and FXRSTOR, “mod = 11” is reserved.

## B.7 SSE INSTRUCTION FORMATS AND ENCODINGS

The SSE instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables (Tables B-21, B-22, and B-23) show the formats and encodings for the SSE SIMD floating-point, SIMD integer, and cacheability and memory ordering instructions, respectively. Some SSE instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. Mandatory prefixes are included in the tables.

**Table B-21. Formats and Encodings of SSE Floating-Point Instructions**

Instruction and Format	Encoding
<b>ADDPS</b> —Add Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1000: mod xmmreg r/m
<b>ADDSS</b> —Add Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg	1111 0011:0000 1111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:01011000: mod xmmreg r/m
<b>ANDNPS</b> —Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0101: mod xmmreg r/m
<b>ANDPS</b> —Bitwise Logical AND of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0100: mod xmmreg r/m

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-21. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>CMPPS—Compare Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg, imm8	0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0000 1111:1100 0010: mod xmmreg r/m: imm8
<b>CMPSS—Compare Scalar Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg, imm8	1111 0011:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0011:0000 1111:1100 0010: mod xmmreg r/m: imm8
<b>COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg to xmmreg	0000 1111:0010 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0010 1111: mod xmmreg r/m
<b>CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</b>	
mmreg to xmmreg	0000 1111:0010 1010:11 xmmreg1 mmreg1
mem to xmmreg	0000 1111:0010 1010: mod xmmreg r/m
<b>CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to mmreg	0000 1111:0010 1101:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1101: mod mmreg r/m
<b>CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value</b>	
r32 to xmmreg1	1111 0011:0000 1111:00101010:11 xmmreg r32
mem to xmmreg	1111 0011:0000 1111:00101010: mod xmmreg r/m
<b>CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	1111 0011:0000 1111:0010 1101:11 r32 xmmreg
mem to r32	1111 0011:0000 1111:0010 1101: mod r32 r/m
<b>CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to mmreg	0000 1111:0010 1100:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1100: mod mmreg r/m

**Table B-21. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	1111 0011:0000 1111:0010 1100:11 r32 xmmreg1
mem to r32	1111 0011:0000 1111:0010 1100: mod r32 r/m
<b>DIVPS—Divide Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1110: mod xmmreg r/m
<b>DIVSS—Divide Scalar Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	1111 0011:0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1110: mod xmmreg r/m
<b>LDMXCSR—Load MXCSR Register State</b>	
m32 to MXCSR	0000 1111:1010 1110:mod <sup>A</sup> 010 mem
<b>MAXPS—Return Maximum Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1111: mod xmmreg r/m
<b>MAXSS—Return Maximum Scalar Double-Precision Floating-Point Value</b>	
xmmreg to xmmreg	1111 0011:0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1111: mod xmmreg r/m
<b>MINPS—Return Minimum Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1101: mod xmmreg r/m
<b>MINSS—Return Minimum Scalar Double-Precision Floating-Point Value</b>	
xmmreg to xmmreg	1111 0011:0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1101: mod xmmreg r/m
<b>MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0010 1000:11 xmmreg2 xmmreg1
mem to xmmreg1	0000 1111:0010 1000: mod xmmreg r/m

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-21. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
xmmreg1 to xmmreg2	0000 1111:0010 1001:11 xmmreg1 xmmreg2
xmmreg1 to mem	0000 1111:0010 1001: mod xmmreg r/m
<b>MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low</b>	
xmmreg to xmmreg	0000 1111:0001 0010:11 xmmreg1 xmmreg2
<b>MOVHPS—Move High Packed Single-Precision Floating-Point Values</b>	
mem to xmmreg	0000 1111:0001 0110: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0111: mod xmmreg r/m
<b>MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High</b>	
xmmreg to xmmreg	0000 1111:00010110:11 xmmreg1 xmmreg2
<b>MOVLPS—Move Low Packed Single-Precision Floating-Point Values</b>	
mem to xmmreg	0000 1111:0001 0010: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0011: mod xmmreg r/m
<b>MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask</b>	
xmmreg to r32	0000 1111:0101 0000:11 r32 xmmreg
<b>MOVSS—Move Scalar Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0001 0000:11 xmmreg2 xmmreg1
mem to xmmreg1	1111 0011:0000 1111:0001 0000: mod xmmreg r/m
xmmreg1 to xmmreg2	1111 0011:0000 1111:0001 0001:11 xmmreg1 xmmreg2
xmmreg1 to mem	1111 0011:0000 1111:0001 0001: mod xmmreg r/m
<b>MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0000 1111:0001 0000:11 xmmreg2 xmmreg1
mem to xmmreg1	0000 1111:0001 0000: mod xmmreg r/m
xmmreg1 to xmmreg2	0000 1111:0001 0001:11 xmmreg1 xmmreg2
xmmreg1 to mem	0000 1111:0001 0001: mod xmmreg r/m
<b>MULPS—Multiply Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1001: mod xmmreg r/m

Table B-21. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
<b>MULSS—Multiply Scalar Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	1111 0011:0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1001: mod xmmreg r/m
<b>ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0000 1111:0101 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0110 mod xmmreg r/m
<b>RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0000 1111:0101 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0011: mod xmmreg r/m
<b>RCPSS—Compute Reciprocals of Scalar Single-Precision Floating-Point Value</b>	
xmmreg to xmmreg	1111 0011:0000 1111:01010011:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:01010011: mod xmmreg r/m
<b>RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0000 1111:0101 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0010 mode xmmreg r/m
<b>RSQRTSS—Compute Reciprocals of Square Roots of Scalar Single-Precision Floating-Point Value</b>	
xmmreg to xmmreg	1111 0011:0000 1111:0101 0010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 0010 mod xmmreg r/m
<b>SHUFPS—Shuffle Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg, imm8	0000 1111:1100 0110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0000 1111:1100 0110: mod xmmreg r/m: imm8
<b>SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0000 1111:0101 0001:11 xmmreg1 xmmreg 2
mem to xmmreg	0000 1111:0101 0001 mod xmmreg r/m

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-21. Formats and Encodings of SSE Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value</b> xmmreg to xmmreg mem to xmmreg	1111 0011:0000 1111:0101 0001:11 xmmreg1 xmmreg 2 1111 0011:0000 1111:0101 0001:mod xmmreg r/m
<b>STMXCSR—Store MXCSR Register State</b> MXCSR to mem	0000 1111:1010 1110:mod <sup>A</sup> 011 mem
<b>SUBPS—Subtract Packed Single-Precision Floating-Point Values</b> xmmreg to xmmreg mem to xmmreg	0000 1111:0101 1100:11 xmmreg1 xmmreg2 0000 1111:0101 1100:mod xmmreg r/m
<b>SUBSS—Subtract Scalar Single-Precision Floating-Point Values</b> xmmreg to xmmreg mem to xmmreg	1111 0011:0000 1111:0101 1100:11 xmmreg1 xmmreg2 1111 0011:0000 1111:0101 1100:mod xmmreg r/m
<b>UCOMISS—Unordered Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS</b> xmmreg to xmmreg mem to xmmreg	0000 1111:0010 1110:11 xmmreg1 xmmreg2 0000 1111:0010 1110 mod xmmreg r/m
<b>UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values</b> xmmreg to xmmreg mem to xmmreg	0000 1111:0001 0101:11 xmmreg1 xmmreg2 0000 1111:0001 0101 mod xmmreg r/m
<b>UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values</b> xmmreg to xmmreg mem to xmmreg	0000 1111:0001 0100:11 xmmreg1 xmmreg2 0000 1111:0001 0100 mod xmmreg r/m
<b>XORPS—Bitwise Logical XOR of Single-Precision Floating-Point Values</b> xmmreg to xmmreg mem to xmmreg	0000 1111:0101 0111:11 xmmreg1 xmmreg2 0000 1111:0101 0111 mod xmmreg r/m



**Table B-22. Formats and Encodings of SSE Integer Instructions**

Instruction and Format	Encoding
<p><b>PAVGB/PAVGW—Average Packed Integers</b></p> <p>mmreg to mmreg</p> <p>mem to mmreg</p>	<p>0000 1111:1110 0000:11 mmreg1 mmreg2</p> <p>0000 1111:1110 0011:11 mmreg1 mmreg2</p> <p>0000 1111:1110 0000 mod mmreg r/m</p> <p>0000 1111:1110 0011 mod mmreg r/m</p>
<p><b>PEXTRW—Extract Word</b></p> <p>mmreg to reg32, imm8</p>	<p>0000 1111:1100 0101:11 r32 mmreg: imm8</p>
<p><b>PINSRW - Insert Word</b></p> <p>reg32 to mmreg, imm8</p> <p>m16 to mmreg, imm8</p>	<p>0000 1111:1100 0100:11 mmreg r32: imm8</p> <p>0000 1111:1100 0100 mod mmreg r/m: imm8</p>
<p><b>PMAXSW—Maximum of Packed Signed Word Integers</b></p> <p>mmreg to mmreg</p> <p>mem to mmreg</p>	<p>0000 1111:1110 1110:11 mmreg1 mmreg2</p> <p>0000 1111:1110 1110 mod mmreg r/m</p>
<p><b>PMAXUB—Maximum of Packed Unsigned Byte Integers</b></p> <p>mmreg to mmreg</p> <p>mem to mmreg</p>	<p>0000 1111:1101 1110:11 mmreg1 mmreg2</p> <p>0000 1111:1101 1110 mod mmreg r/m</p>
<p><b>PMINSW—Minimum of Packed Signed Word Integers</b></p> <p>mmreg to mmreg</p> <p>mem to mmreg</p>	<p>0000 1111:1110 1010:11 mmreg1 mmreg2</p> <p>0000 1111:1110 1010 mod mmreg r/m</p>
<p><b>PMINUB—Minimum of Packed Unsigned Byte Integers</b></p> <p>mmreg to mmreg</p> <p>mem to mmreg</p>	<p>0000 1111:1101 1010:11 mmreg1 mmreg2</p> <p>0000 1111:1101 1010 mod mmreg r/m</p>
<p><b>PMOVBMSKB - Move Byte Mask To Integer</b></p> <p>mmreg to reg32</p>	<p>0000 1111:1101 0111:11 r32 mmreg</p>
<p><b>PMLHUW—Multiply Packed Unsigned Integers and Store High Result</b></p> <p>mmreg to mmreg</p> <p>mem to mmreg</p>	<p>0000 1111:1110 0100:11 mmreg1 mmreg2</p> <p>0000 1111:1110 0100 mod mmreg r/m</p>

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-22. Formats and Encodings of SSE Integer Instructions (Contd.)**

Instruction and Format	Encoding
<b>PSADBW—Compute Sum of Absolute Differences</b>	
mmreg to mmreg	0000 1111:1111 0110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 0110 mod mmreg r/m
<b>PSHUFW—Shuffle Packed Words</b>	
mmreg to mmreg, imm8	0000 1111:0111 0000:11 mmreg1 mmreg2: imm8
mem to mmreg, imm8	0000 1111:0111 0000:11 mod mmreg r/m: imm8

**Table B-23. Format and Encoding of SSE Cacheability & Memory Ordering Instructions**

Instruction and Format	Encoding
<b>MASKMOVQ—Store Selected Bytes of Quadword</b>	
mmreg to mmreg	0000 1111:1111 0111:11 mmreg1 mmreg2
<b>MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint</b>	
xmmreg to mem	0000 1111:0010 1011: mod xmmreg r/m
<b>MOVNTQ—Store Quadword Using Non-Temporal Hint</b>	
mmreg to mem	0000 1111:1110 0111: mod mmreg r/m
<b>PREFETCH0—Prefetch Temporal to All Cache Levels</b>	0000 1111:0001 1000:mod <sup>A</sup> 001 mem
<b>PREFETCH1—Prefetch Temporal to First Level Cache</b>	0000 1111:0001 1000:mod <sup>A</sup> 010 mem
<b>PREFETCH2—Prefetch Temporal to Second Level Cache</b>	0000 1111:0001 1000:mod <sup>A</sup> 011 mem
<b>PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels</b>	0000 1111:0001 1000:mod <sup>A</sup> 000 mem
<b>SFENCE—Store Fence</b>	0000 1111:1010 1110:11 111 000

## B.8 SSE2 INSTRUCTION FORMATS AND ENCODINGS

The SSE2 instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables show the formats and encodings for the SSE2 SIMD floating-point, SIMD integer, and cacheability instructions, respectively. Some SSE2 instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These prefixes are included in the tables.

### B.8.1 Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-24 shows the encoding of this gg field.

**Table B-24. Encoding of Granularity of Data Field (gg)**

gg	Granularity of Data
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

**Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions**

Instruction and Format	Encoding
<b>ADDPD - Add Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1000: mod xmmreg r/m
<b>ADDSD - Add Scalar Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	1111 0010:0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1000: mod xmmreg r/m
<b>ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0101 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0101: mod xmmreg r/m

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0100: mod xmmreg r/m
<b>CMPPD—Compare Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg, imm8	0110 0110:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:1100 0010: mod xmmreg r/m: imm8
<b>CMPSD—Compare Scalar Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg, imm8	1111 0010:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 010:0000 1111:1100 0010: mod xmmreg r/m: imm8
<b>COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0010 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0010 1111: mod xmmreg r/m
<b>CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values</b>	
mmreg to xmmreg	0110 0110:0000 1111:0010 1010:11 xmmreg1 mmreg1
mem to xmmreg	0110 0110:0000 1111:0010 1010: mod xmmreg r/m
<b>CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to mmreg	0110 0110:0000 1111:0010 1101:11 mmreg1 xmmreg1
mem to mmreg	0110 0110:0000 1111:0010 1101: mod mmreg r/m
<b>CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value</b>	
r32 to xmmreg1	1111 0010:0000 1111:0010 1010:11 xmmreg r32
mem to xmmreg1	1111 0010:0000 1111:0010 1010: mod xmmreg r/m

Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
<b>CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>  xmmreg to r32 mem to r32	1111 0010:0000 1111:0010 1101:11 r32 xmmreg 1111 0010:0000 1111:0010 1101: mod r32 r/m
<b>CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>  xmmreg to mmreg mem to mmreg	0110 0110:0000 1111:0010 1100:11 mmreg xmmreg 0110 0110:0000 1111:0010 1100: mod mmreg r/m
<b>CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>  xmmreg to r32 mem to r32	1111 0010:0000 1111:0010 1100:11 r32 xmmreg 1111 0010:0000 1111:0010 1100: mod r32 r/m
<b>CVTPD2PS—Covert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values</b>  xmmreg to xmmreg mem to xmmreg	0110 0110:0000 1111:0101 1010:11 xmmreg1 xmmreg2 0110 0110:0000 1111:0101 1010: mod xmmreg r/m
<b>CVTPS2PD—Covert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values</b>  xmmreg to xmmreg mem to xmmreg	0000 1111:0101 1010:11 xmmreg1 xmmreg2 0000 1111:0101 1010: mod xmmreg r/m
<b>CVTSD2SS—Covert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value</b>  xmmreg to xmmreg mem to xmmreg	1111 0010:0000 1111:0101 1010:11 xmmreg1 xmmreg2 1111 0010:0000 1111:0101 1010: mod xmmreg r/m
<b>CVTSS2SD—Covert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value</b>  xmmreg to xmmreg mem to xmmreg	1111 0011:0000 1111:0101 1010:11 xmmreg1 xmmreg2 1111 0011:0000 1111:0101 1010: mod xmmreg r/m

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to xmmreg	1111 0010:0000 1111:1110 0110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:1110 0110: mod xmmreg r/m
<b>CVTTPD2DQ—Convert With Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to xmmreg	0110 0110:0000 1111:1110 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1110 0110: mod xmmreg r/m
<b>CVTDQ2PD—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values</b>	
xmmreg to xmmreg	1111 0011:0000 1111:1110 0110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:1110 0110: mod xmmreg r/m
<b>CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1011: mod xmmreg r/m
<b>CVTTPS2DQ—Convert With Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers</b>	
xmmreg to xmmreg	1111 0011:0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1011: mod xmmreg r/m
<b>CVTDQ2PS—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1011: mod xmmreg r/m
<b>DIVPD—Divide Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1110: mod xmmreg r/m
<b>DIVSD—Divide Scalar Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	1111 0010:0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1110: mod xmmreg r/m

Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
<b>MAXPD—Return Maximum Packed Double-Precision Floating-Point Values</b> xmmreg to xmmreg mem to xmmreg	0110 0110:0000 1111:0101 1111:11 xmmreg1 xmmreg2 0110 0110:0000 1111:0101 1111: mod xmmreg r/m
<b>MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value</b> xmmreg to xmmreg mem to xmmreg	1111 0010:0000 1111:0101 1111:11 xmmreg1 xmmreg2 1111 0010:0000 1111:0101 1111: mod xmmreg r/m
<b>MINPD—Return Minimum Packed Double-Precision Floating-Point Values</b> xmmreg to xmmreg mem to xmmreg	0110 0110:0000 1111:0101 1101:11 xmmreg1 xmmreg2 0110 0110:0000 1111:0101 1101: mod xmmreg r/m
<b>MINSD—Return Minimum Scalar Double-Precision Floating-Point Value</b> xmmreg to xmmreg mem to xmmreg	1111 0010:0000 1111:0101 1101:11 xmmreg1 xmmreg2 1111 0010:0000 1111:0101 1101: mod xmmreg r/m
<b>MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values</b> xmmreg2 to xmmreg1 mem to xmmreg1 xmmreg1 to xmmreg2 xmmreg1 to mem	0110 0110:0000 1111:0010 1001:11 xmmreg2 xmmreg1 0110 0110:0000 1111:0010 1001: mod xmmreg r/m 0110 0110:0000 1111:0010 1000:11 xmmreg1 xmmreg2 0110 0110:0000 1111:0010 1000: mod xmmreg r/m
<b>MOVHPD—Move High Packed Double-Precision Floating-Point Values</b> mem to xmmreg xmmreg to mem	0110 0110:0000 1111:0001 0111: mod xmmreg r/m 0110 0110:0000 1111:0001 0110: mod xmmreg r/m
<b>MOVLPD—Move Low Packed Double-Precision Floating-Point Values</b> mem to xmmreg xmmreg to mem	0110 0110:0000 1111:0001 0011: mod xmmreg r/m 0110 0110:0000 1111:0001 0010: mod xmmreg r/m
<b>MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask</b> xmmreg to r32	0110 0110:0000 1111:0101 0000:11 r32 xmmreg

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>MOVSD—Move Scalar Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0001 0001:11 xmmreg2 xmmreg1
mem to xmmreg1	1111 0010:0000 1111:0001 0001: mod xmmreg r/m
xmmreg1 to xmmreg2	1111 0010:0000 1111:0001 0000:11 xmmreg1 xmmreg2
xmmreg1 to mem	1111 0010:0000 1111:0001 0000: mod xmmreg r/m
<b>MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0001 0001:11 xmmreg2 xmmreg1
mem to xmmreg1	0110 0110:0000 1111:0001 0001: mod xmmreg r/m
xmmreg1 to xmmreg2	0110 0110:0000 1111:0001 0000:11 xmmreg1 xmmreg2
xmmreg1 to mem	0110 0110:0000 1111:0001 0000: mod xmmreg r/m
<b>MULPD—Multiply Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1001: mod xmmreg rm
<b>MULSD—Multiply Scalar Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	1111 0010:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:00001111:01011001: mod xmmreg r/m
<b>ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0101 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0110: mod xmmreg r/m
<b>SHUFPD—Shuffle Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg, imm8	0110 0110:0000 1111:1100 0110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:1100 0110: mod xmmreg r/m: imm8
<b>SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0101 0001:11 xmmreg1 xmmreg 2
mem to xmmreg	0110 0110:0000 1111:0101 0001: mod xmmreg r/m



## INSTRUCTION FORMATS AND ENCODINGS

**Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)**

Instruction and Format	Encoding
<b>SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value</b>	
xmmreg to xmmreg	1111 0010:0000 1111:0101 0001:11 xmmreg1 xmmreg 2
mem to xmmreg	1111 0010:0000 1111:0101 0001: mod xmmreg r/m
<b>SUBPD—Subtract Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1100: mod xmmreg r/m
<b>SUBSD—Subtract Scalar Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	1111 0010:0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1100: mod xmmreg r/m
<b>UCOMISD—Unordered Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0010 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0010 1110: mod xmmreg r/m
<b>UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0001 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0001 0101: mod xmmreg r/m
<b>UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0001 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0001 0100: mod xmmreg r/m
<b>XORPD—Bitwise Logical OR of Double-Precision Floating-Point Values</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0101 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0111: mod xmmreg r/m

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-26. Formats and Encodings of SSE2 Integer Instructions**

Instruction and Format	Encoding
<b>MOVD - Move Doubleword</b> reg to xmmreg reg from xmmreg mem to xmmreg mem from xmmreg	0110 0110:0000 1111:0110 1110: 11 xmmreg reg 0110 0110:0000 1111:0111 1110: 11 xmmreg reg 0110 0110:0000 1111:0110 1110: mod xmmreg r/m 0110 0110:0000 1111:0111 1110: mod xmmreg r/m
<b>MOVDQA—Move Aligned Double Quadword</b> xmmreg to xmmreg mem to xmmreg mem from xmmreg	0110 0110:0000 1111:0110 1111:11 xmmreg1 xmmreg2 0110 0110:0000 1111:0111 1111:11 xmmreg1 xmmreg2 0110 0110:0000 1111:0110 1111: mod xmmreg r/m 0110 0110:0000 1111:0111 1111: mod xmmreg r/m
<b>MOVDQU—Move Unaligned Double Quadword</b> xmmreg to xmmreg mem to xmmreg mem from xmmreg	1111 0011:0000 1111:0110 1111:11 xmmreg1 xmmreg2 1111 0011:0000 1111:0111 1111:11 xmmreg1 xmmreg2 1111 0011:0000 1111:0110 1111: mod xmmreg r/m 1111 0011:0000 1111:0111 1111: mod xmmreg r/m
<b>MOVQ2DQ—Move Quadword from MMX to XMM Register</b> mmreg to xmmreg	1111 0011:0000 1111:1101 0110:11 mmreg1 mmreg2
<b>MOVDQ2Q—Move Quadword from XMM to MMX Register</b> xmmreg to mmreg	1111 0010:0000 1111:1101 0110:11 mmreg1 mmreg2
<b>MOVQ - Move Quadword</b> xmmreg2 to xmmreg1 xmmreg2 from xmmreg1 mem to xmmreg mem from xmmreg	1111 0011:0000 1111:0111 1110: 11 xmmreg1 xmmreg2 0110 0110:0000 1111:1101 0110: 11 xmmreg1 xmmreg2 1111 0011:0000 1111:0111 1110: mod xmmreg r/m 0110 0110:0000 1111:1101 0110: mod xmmreg r/m
<b>PACKSSDW<sup>1</sup> - Pack Dword To Word Data (signed with saturation)</b> xmmreg2 to xmmreg1 memory to xmmreg	0110 0110:0000 1111:0110 1011: 11 xmmreg1 xmmreg2 0110 0110:0000 1111:0110 1011: mod xmmreg r/m

Table B-26. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
<b>PACKSSWB - Pack Word To Byte Data (signed with saturation)</b> xmmreg2 to xmmreg1 memory to xmmreg	0110 0110:0000 1111:0110 0011: 11 xmmreg1 xmmreg2 0110 0110:0000 1111:0110 0011: mod xmmreg r/m
<b>PACKUSWB - Pack Word To Byte Data (unsigned with saturation)</b> xmmreg2 to xmmreg1 memory to xmmreg	0110 0110:0000 1111:0110 0111: 11 xmmreg1 xmmreg2 0110 0110:0000 1111:0110 0111: mod xmmreg r/m
<b>PADDQ—Add Packed Quadword Integers</b> mmreg to mmreg mem to mmreg xmmreg to xmmreg mem to xmmreg	0000 1111:1101 0100:11 mmreg1 mmreg2 0000 1111:1101 0100: mod mmreg r/m 0110 0110:0000 1111:1101 0100:11 xmmreg1 xmmreg2 0110 0110:0000 1111:1101 0100: mod xmmreg r/m
<b>PADD - Add With Wrap-around</b> xmmreg2 to xmmreg1 memory to xmmreg	0110 0110:0000 1111: 1111 11gg: 11 xmmreg1 xmmreg2 0110 0110:0000 1111: 1111 11gg: mod xmmreg r/m
<b>PADDs - Add Signed With Saturation</b> xmmreg2 to xmmreg1 memory to xmmreg	0110 0110:0000 1111: 1110 11gg: 11 xmmreg1 xmmreg2 0110 0110:0000 1111: 1110 11gg: mod xmmreg r/m
<b>PADDUS - Add Unsigned With Saturation</b> xmmreg2 to xmmreg1 memory to xmmreg	0110 0110:0000 1111: 1101 11gg: 11 xmmreg1 xmmreg2 0110 0110:0000 1111: 1101 11gg: mod xmmreg r/m
<b>PAND - Bitwise And</b> xmmreg2 to xmmreg1 memory to xmmreg	0110 0110:0000 1111:1101 1011: 11 xmmreg1 xmmreg2 0110 0110:0000 1111:1101 1011: mod xmmreg r/m
<b>PANDN - Bitwise AndNot</b> xmmreg2 to xmmreg1 memory to xmmreg	0110 0110:0000 1111:1101 1111: 11 xmmreg1 xmmreg2 0110 0110:0000 1111:1101 1111: mod xmmreg r/m
<b>PAVGB—Average Packed Integers</b> xmmreg to xmmreg mem to xmmreg	0110 0110:0000 1111:11100 000:11 xmmreg1 xmmreg2 01100110:00001111:11100000 mod xmmreg r/m

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-26. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

Instruction and Format	Encoding
<b>PAVGW—Average Packed Integers</b>	
xmmreg to xmmreg	0110 0110:0000 1111:1110 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1110 0011 mod xmmreg r/m
<b>PCMPEQ - Packed Compare For Equality</b>	
xmmreg1 with xmmreg2	0110 0110:0000 1111:0111 01gg: 11 xmmreg1 xmmreg2
xmmreg with memory	0110 0110:0000 1111:0111 01gg: mod xmmreg r/m
<b>PCMPGT - Packed Compare Greater (signed)</b>	
xmmreg1 with xmmreg2	0110 0110:0000 1111:0110 01gg: 11 xmmreg1 xmmreg2
xmmreg with memory	0110 0110:0000 1111:0110 01gg: mod xmmreg r/m
<b>PEXTRW—Extract Word</b>	
xmmreg to reg32, imm8	0110 0110:0000 1111:1100 0101:11 r32 xmmreg: imm8
<b>PINSRW - Insert Word</b>	
reg32 to xmmreg, imm8	0110 0110:0000 1111:1100 0100:11 xmmreg r32: imm8
m16 to xmmreg, imm8	0110 0110:0000 1111:1100 0100 mod xmmreg r/m: imm8
<b>PMADDWD - Packed Multiply Add</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 0101: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1111 0101: mod xmmreg r/m
<b>PMAXSW—Maximum of Packed Signed Word Integers</b>	
xmmreg to xmmreg	0110 0110:0000 1111:1110 1110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101110 mod xmmreg r/m
<b>PMAXUB—Maximum of Packed Unsigned Byte Integers</b>	
xmmreg to xmmreg	0110 0110:0000 1111:1101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1101 1110 mod xmmreg r/m
<b>PMINSW—Minimum of Packed Signed Word Integers</b>	
xmmreg to xmmreg	0110 0110:0000 1111:1110 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1110 1010 mod xmmreg r/m
<b>PMINUB—Minimum of Packed Unsigned Byte Integers</b>	
xmmreg to xmmreg	0110 0110:0000 1111:1101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1101 1010 mod xmmreg r/m

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-26. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

Instruction and Format	Encoding
<b>PMOVBMSKB - Move Byte Mask To Integer</b> xmmreg to reg32	0110 0110:0000 1111:1101 0111:11 r32 xmmreg
<b>PMULHUW - Packed multiplication, store high word (unsigned)</b> xmmreg2 to xmmreg1 memory to xmmreg	0110 0110:0000 1111:1110 0100: 11 xmmreg1 xmmreg2 0110 0110:0000 1111:1110 0100: mod xmmreg r/m
<b>PMULHW - Packed Multiplication, store high word</b> xmmreg2 to xmmreg1 memory to xmmreg	0110 0110:0000 1111:1110 0101: 11 xmmreg1 xmmreg2 0110 0110:0000 1111:1110 0101: mod xmmreg r/m
<b>PMULLW - Packed Multiplication, store low word</b> xmmreg2 to xmmreg1 memory to xmmreg	0110 0110:0000 1111:1101 0101: 11 xmmreg1 xmmreg2 0110 0110:0000 1111:1101 0101: mod xmmreg r/m
<b>PMULUDQ—Multiply Packed Unsigned Doubleword Integers</b> mmreg to mmreg mem to mmreg xmmreg to xmmreg mem to xmmreg	0000 1111:1111 0100:11 mmreg1 mmreg2 0000 1111:1111 0100: mod mmreg r/m 0110 0110:00001111:1111 0100:11 xmmreg1 xmmreg2 0110 0110:00001111:1111 0100: mod xmmreg r/m
<b>POR - Bitwise Or</b> xmmreg2 to xmmreg1 xmemory to xmmreg	0110 0110:0000 1111:1110 1011: 11 xmmreg1 xmmreg2 0110 0110:0000 1111:1110 1011: mod xmmreg r/m
<b>PSADBW—Compute Sum of Absolute Differences</b> xmmreg to xmmreg mem to xmmreg	0110 0110:0000 1111:1111 0110:11 xmmreg1 xmmreg2 0110 0110:0000 1111:1111 0110: mod xmmreg r/m
<b>PSHUFLW—Shuffle Packed Low Words</b> xmmreg to xmmreg, imm8 mem to xmmreg, imm8	1111 0010:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8 1111 0010:0000 1111:0111 0000:11 mod xmmreg r/m: imm8

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-26. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

Instruction and Format	Encoding
<b>PSHUFHW—Shuffle Packed High Words</b>	
xmmreg to xmmreg, imm8	1111 0011:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0011:0000 1111:0111 0000:11 mod xmmreg r/m: imm8
<b>PSHUFD—Shuffle Packed Doublewords</b>	
xmmreg to xmmreg, imm8	0110 0110:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0111 0000:11 mod xmmreg r/m: imm8
<b>PSLLDQ—Shift Double Quadword Left Logical</b>	
xmmreg, imm8	0110 0110:0000 1111:0111 0011:11 111 xmmreg: imm8
<b>PSLL - Packed Shift Left Logical</b>	
xmmreg1 by xmmreg2	0110 0110:0000 1111:1111 00gg: 11 xmmreg1 xmmreg2
xmmreg by memory	0110 0110:0000 1111:1111 00gg: mod xmmreg r/m
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 110 xmmreg: imm8
<b>PSRA - Packed Shift Right Arithmetic</b>	
xmmreg1 by xmmreg2	0110 0110:0000 1111:1110 00gg: 11 xmmreg1 xmmreg2
xmmreg by memory	0110 0110:0000 1111:1110 00gg: mod xmmreg r/m
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 100 xmmreg: imm8
<b>PSRLDQ—Shift Double Quadword Right Logical</b>	
xmmreg, imm8	0110 0110:00001111:01110011:11 011 xmmreg: imm8
<b>PSRL - Packed Shift Right Logical</b>	
xmmreg1 by xmmreg2	0110 0110:0000 1111:1101 00gg: 11 xmmreg1 xmmreg2
xmmreg by memory	0110 0110:0000 1111:1101 00gg: mod xmmreg r/m
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 010 xmmreg: imm8
<b>PSUBQ—Subtract Packed Quadword Integers</b>	
mmreg to mmreg	0000 1111:111111 011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 1011: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:1111 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1111 1011: mod xmmreg r/m

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-26. Formats and Encodings of SSE2 Integer Instructions (Contd.)**

Instruction and Format	Encoding
<b>PSUB - Subtract With Wrap-around</b>	
xmmreg2 from xmmreg1	0110 0110:0000 1111:1111 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0110 0110:0000 1111:1111 10gg: mod xmmreg r/m
<b>PSUBS - Subtract Signed With Saturation</b>	
xmmreg2 from xmmreg1	0110 0110:0000 1111:1110 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0110 0110:0000 1111:1110 10gg: mod xmmreg r/m
<b>PSUBUS - Subtract Unsigned With Saturation</b>	
xmmreg2 from xmmreg1	0000 1111:1101 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0000 1111:1101 10gg: mod xmmreg r/m
<b>PUNPCKH—Unpack High Data To Next Larger Type</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0110 10gg:11 xmmreg1 Xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 10gg: mod xmmreg r/m
<b>PUNPCKHQDQ—Unpack High Data</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0110 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 1101: mod xmmreg r/m
<b>PUNPCKL—Unpack Low Data To Next Larger Type</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0110 00gg:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 00gg: mod xmmreg r/m
<b>PUNPCKLQDQ—Unpack Low Data</b>	
xmmreg to xmmreg	0110 0110:0000 1111:0110 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 1100: mod xmmreg r/m
<b>PXOR - Bitwise Xor</b>	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1111: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 1111: mod xmmreg r/m

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-27. Format and Encoding of SSE2 Cacheability Instructions**

Instruction and Format	Encoding
<b>MASKMOVDQU—Store Selected Bytes of Double Quadword</b> xmmreg to xmmreg	0110 0110:0000 1111:1111 0111:11 xmmreg1 xmmreg2
<b>CLFLUSH—Flush Cache Line</b> mem	0000 1111:1010 1110:mod r/m
<b>MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint</b> xmmreg to mem	0110 0110:0000 1111:0010 1011: mod xmmreg r/m
<b>MOVNTDQ—Store Double Quadword Using Non-Temporal Hint</b> xmmreg to mem	0110 0110:0000 1111:1110 0111: mod xmmreg r/m
<b>MOVNTI—Store Doubleword Using Non-Temporal Hint</b> reg to mem	0000 1111:1100 0011: mod reg r/m
<b>PAUSE—Spin Loop Hint</b>	1111 0011:1001 0000
<b>LFENCE—Load Fence</b>	0000 1111:1010 1110: 11 101 000
<b>MFENCE—Memory Fence</b>	0000 1111:1010 1110: 11 110 000



## B.9 SSE3 FORMATS AND ENCODINGS TABLE

The tables in this section provide SSE3 formats and encodings. Some SSE3 instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These prefixes are included in the tables.

When in IA-32e mode, use of the REX.R prefix permits instructions that use general purpose and XMM registers to access additional registers. Some instructions require the REX.W prefix to promote the instruction to 64-bit operation. Instructions that require the REX.W prefix are listed (with their opcodes) in Section B.10.

**Table B-28. Formats and Encodings of SSE3 Floating-Point Instructions**

Instruction and Format	Encoding
<b>ADDSD—Add /Sub packed DP FP numbers from XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	01100110:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11010000: mod xmmreg r/m
<b>ADDSPS — Add /Sub packed SP FP numbers from XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	11110010:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:11010000: mod xmmreg r/m
<b>HADDPD — Add horizontally packed DP FP numbers XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	01100110:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111100: mod xmmreg r/m
<b>HADDPS — Add horizontally packed SP FP numbers XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	11110010:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01111100: mod xmmreg r/m
<b>HSUBPD — Sub horizontally packed DP FP numbers XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	01100110:00001111:01111101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111101: mod xmmreg r/m
<b>HSUBPS — Sub horizontally packed SP FP numbers XMM2/Mem to XMM1</b>	
xmmreg2 to xmmreg1	11110010:00001111:01111101:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01111101: mod xmmreg r/m

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-29. Formats and Encodings for SSE3 Event Management Instructions**

Instruction and Format	Encoding
<b>MONITOR</b> — Set up a linear address range to be monitored by hardware eax, ecx, edx	0000 1111 : 0000 0001:11 001 000
<b>MWAIT</b> — Wait until write-back store performed within the range specified by the instruction <b>MONITOR</b> eax, ecx	0000 1111 : 0000 0001:11 001 001

**Table B-30. Formats and Encodings for SSE3 Integer and Move Instructions**

Instruction and Format	Encoding
<b>FISTTP</b> — Store ST in int16 (chop) and pop m16int	11011 111 : mod <sup>A</sup> 001 r/m
<b>FISTTP</b> — Store ST in int32 (chop) and pop m32int	11011 011 : mod <sup>A</sup> 001 r/m
<b>FISTTP</b> — Store ST in int64 (chop) and pop m64int	11011 101 : mod <sup>A</sup> 001 r/m
<b>LDDQU</b> — Load unaligned integer 128-bit xmm, m128	11110010:00001111:11110000: mod <sup>A</sup> xmmreg r/m
<b>MOVDDUP</b> — Move 64 bits representing one DP data from XMM2/Mem to XMM1 and duplicate xmmreg2 to xmmreg1 mem to xmmreg	11110010:00001111:00010010:11 xmmreg1 xmmreg2 11110010:00001111:00010010: mod xmmreg r/m
<b>MOVSHDUP</b> — Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate high xmmreg2 to xmmreg1 mem to xmmreg	11110011:00001111:00010110:11 xmmreg1 xmmreg2 11110011:00001111:00010110: mod xmmreg r/m
<b>MOVSLDUP</b> — Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate low xmmreg2 to xmmreg1 mem to xmmreg	11110011:00001111:00010010:11 xmmreg1 xmmreg2 11110011:00001111:00010010: mod xmmreg r/m

## B.10 SPECIAL ENCODINGS FOR 64-BIT MODE

The following Pentium, P6, MMX, SSE, SSE2, SSE3 instructions are promoted to 64-bit operation in IA-32e mode by using REX.W. However, these entries are special cases that do not follow the general rules (specified in Section B.4).

**Table B-31. Special Case Instructions Promoted Using REX.W**

Instruction and Format	Encoding
<b>CMOVcc – Conditional Move</b> register2 to register1 qwordregister2 to qwordregister1 memory to register memory64 to qwordregister	0100 0R0B 0000 1111: 0100 ttn : 11 reg1 reg2 0100 1R0B 0000 1111: 0100 ttn : 11 qwordreg1 qwordreg2 0100 0RXB 0000 1111 : 0100 ttn : mod reg r/m 0100 1RXB 0000 1111 : 0100 ttn : mod qwordreg r/m
<b>CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer</b> xmmreg to r32 xmmreg to r64 mem64 to r32 mem64 to r64	0100 0R0B 1111 0010:0000 1111:0010 1101:11 r32 xmmreg 0100 1R0B 1111 0010:0000 1111:0010 1101:11 r64 xmmreg 0100 0R0XB 1111 0010:0000 1111:0010 1101: mod r32 r/m 0100 1RXB 1111 0010:0000 1111:0010 1101: mod r64 r/m
<b>CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value</b> r32 to xmmreg1 r64 to xmmreg1 mem to xmmreg mem64 to xmmreg	0100 0R0B 1111 0011:0000 1111:0010 1010:11 xmmreg r32 0100 1R0B 1111 0011:0000 1111:0010 1010:11 xmmreg r64 0100 0RXB 1111 0011:0000 1111:0010 1010: mod xmmreg r/m 0100 1RXB 1111 0011:0000 1111:0010 1010: mod xmmreg r/m
<b>CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value</b> r32 to xmmreg1 r64 to xmmreg1	0100 0R0B 1111 0010:0000 1111:0010 1010:11 xmmreg r32 0100 1R0B 1111 0010:0000 1111:0010 1010:11 xmmreg r64

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-31. Special Case Instructions Promoted Using REX.W (Contd.)**

Instruction and Format	Encoding
mem to xmmreg	0100 0RXB 1111 0010:0000 1111:00101 010: mod xmmreg r/m
mem64 to xmmreg	0100 1RXB 1111 0010:0000 1111:0010 1010: mod xmmreg r/m
<b>CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	0100 0R0B 1111 0011:0000 1111:0010 1101:11 r32 xmmreg
xmmreg to r64	0100 1R0B 1111 0011:0000 1111:0010 1101:11 r64 xmmreg
mem to r32	0100 0RXB 11110011:00001111:00101101: mod r32 r/m
mem32 to r64	0100 1RXB 1111 0011:0000 1111:0010 1101: mod r64 r/m
<b>CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	0100 0R0B 11110010:00001111:00101100:11 r32 xmmreg
xmmreg to r64	0100 1R0B 1111 0010:0000 1111:0010 1100:11 r64 xmmreg
mem64 to r32	0100 0RXB 1111 0010:0000 1111:0010 1100: mod r32 r/m
mem64 to r64	0100 1RXB 1111 0010:0000 1111:0010 1100: mod r64 r/m
<b>CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer</b>	
xmmreg to r32	0100 0R0B 1111 0011:0000 1111:0010 1100:11 r32 xmmreg1
xmmreg to r64	0100 1R0B 1111 0011:0000 1111:0010 1100:11 r64 xmmreg1
mem to r32	0100 0RXB 1111 0011:0000 1111:0010 1100: mod r32 r/m
mem32 to r64	0100 1RXB 1111 0011:0000 1111:0010 1100: mod r64 r/m
<b>MOVD/MOVQ - Move doubleword</b>	
reg to mmxreg	0100 0R0B 0000 1111:0110 1110: 11 mmxreg reg
qwordreg to mmxreg	0100 1R0B 0000 1111:0110 1110: 11 mmxreg qwordreg
reg from mmxreg	0100 0R0B 0000 1111:0111 1110: 11 mmxreg reg

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-31. Special Case Instructions Promoted Using REX.W (Contd.)**

Instruction and Format	Encoding
qwordreg from mmxreg	0100 1R0B 0000 1111:0111 1110: 11 mmxreg qwordreg
mem to mmxreg	0100 0RXB 0000 1111:0110 1110: mod mmxreg r/m
mem64 to mmxreg	0100 1RXB 0000 1111:0110 1110: mod mmxreg r/m
mem from mmxreg	0100 0RXB 0000 1111:0111 1110: mod mmxreg r/m
mem64 from mmxreg	0100 1RXB 0000 1111:0111 1110: mod mmxreg r/m
mmxreg with memory	0100 0RXB 0000 1111:0110 01gg: mod mmxreg r/m
<b>MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask</b>	
xmmreg to r32	0100 0R0B 0000 1111:0101 0000:11 r32 xmmreg
xmmreg to r64	0100 1R0B 00001111:01010000:11 r64 xmmreg
<b>PEXTRW—Extract Word</b>	
mmreg to reg32, imm8	0100 0R0B 0000 1111:1100 0101:11 r32 mmreg: imm8
mmreg to reg64, imm8	0100 1R0B 0000 1111:1100 0101:11 r64 mmreg: imm8
xmmreg to reg32, imm8	0100 0R0B 0110 0110 0000 1111:1100 0101:11 r32 xmmreg: imm8
xmmreg to reg64, imm8	0100 1R0B 0110 0110 0000 1111:1100 0101:11 r64 xmmreg: imm8
<b>PINSRW - Insert Word</b>	
reg32 to mmreg, imm8	0100 0R0B 0000 1111:1100 0100:11 mmreg r32: imm8
reg64 to mmreg, imm8	0100 1R0B 0000 1111:1100 0100:11 mmreg r64: imm8
m16 to mmreg, imm8	0100 0R0B 0000 1111:1100 0100 mod mmreg r/m: imm8
m16 to mmreg, imm8	0100 1RXB 0000 1111:11000100 mod mmreg r/m: imm8
reg32 to xmmreg, imm8	0100 0RXB 0110 0110 0000 1111:1100 0100:11 xmmreg r32: imm8
reg64 to xmmreg, imm8	0100 0RXB 0110 0110 0000 1111:1100 0100:11 xmmreg r64: imm8
m16 to xmmreg, imm8	0100 0RXB 0110 0110 0000 1111:1100 0100 mod xmmreg r/m: imm8

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-31. Special Case Instructions Promoted Using REX.W (Contd.)**

Instruction and Format	Encoding
m16 to xmmreg, imm8	0100 1RXB 0110 0110 0000 1111:1100 0100 mod xmmreg r/m: imm8
<b>PMOVMASK - Move Byte Mask To Integer</b>	
mmreg to reg32	0100 0RXB 0000 1111:1101 0111:11 r32 mmreg
mmreg to reg64	0100 1R0B 0000 1111:1101 0111:11 r64 mmreg
xmmreg to reg32	0100 0RXB 0110 0110 0000 1111:1101 0111:11 r32 mmreg
xmmreg to reg64	0110 0110 0000 1111:1101 0111:11 r64 xmmreg

## B.11 FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS

Table B-32 shows the five different formats used for floating-point instructions. In all cases, instructions are at least two bytes long and begin with the bit pattern 11011.

**Table B-32. General Floating-Point Instruction Formats**

	Instruction										Optional Fields		
	First Byte				Second Byte								
1	11011	OPA		1	mod		1	OPB		r/m	s-i-b	disp	
2	11011	MF			OPA		mod		OPB		r/m	s-i-b	disp
3	11011	d	P	OPA		1	1	OPB		R	ST(i)		
4	11011	0	0	1	1	1	1	OP					
5	11011	0	1	1	1	1	1	OP					
	15-11	10	9	8	7	6	5	4	3	2	1	0	

MF = Memory Format

- 00 — 32-bit real
- 01 — 32-bit integer
- 10 — 64-bit real
- 11 — 16-bit integer

P = Pop

- 0 — Do not pop stack
- 1 — Pop stack after operation

d = Destination

- 0 — Destination is ST(0)
- 1 — Destination is ST(i)

R XOR d = 0 — Destination OP Source

R XOR d = 1 — Source OP Destination

ST(i) = Register stack element *i*

000 = Stack Top

001 = Second stack element

.

.

.

111 = Eighth stack element

## INSTRUCTION FORMATS AND ENCODINGS

The Mod and R/M fields of the ModR/M byte have the same interpretation as the corresponding fields of the integer instructions. The SIB byte and disp (displacement) are optionally present in instructions that have Mod and R/M fields. Their presence depends on the values of Mod and R/M, as for integer instructions.

Table B-33 shows the formats and encodings of the floating-point instructions.

**Table B-33. Floating-Point Instruction Formats and Encodings**

Instruction and Format	Encoding
<b>F2XM1 – Compute <math>2^{ST(0)} - 1</math></b>	11011 001 : 1111 0000
<b>FABS – Absolute Value</b>	11011 001 : 1110 0001
<b>FADD – Add</b>	
$ST(0) \leftarrow ST(0) + 32\text{-bit memory}$	11011 000 : mod 000 r/m
$ST(0) \leftarrow ST(0) + 64\text{-bit memory}$	11011 100 : mod 000 r/m
$ST(d) \leftarrow ST(0) + ST(i)$	11011 d00 : 11 000 ST(i)
<b>FADDP – Add and Pop</b>	
$ST(0) \leftarrow ST(0) + ST(i)$	11011 110 : 11 000 ST(i)
<b>FBLD – Load Binary Coded Decimal</b>	11011 111 : mod 100 r/m
<b>FBSTP – Store Binary Coded Decimal and Pop</b>	11011 111 : mod 110 r/m
<b>FCBS – Change Sign</b>	11011 001 : 1110 0000
<b>FCLEX – Clear Exceptions</b>	11011 011 : 1110 0010
<b>FCOM – Compare Real</b>	
32-bit memory	11011 000 : mod 010 r/m
64-bit memory	11011 100 : mod 010 r/m
$ST(i)$	11011 000 : 11 010 ST(i)
<b>FCOMP – Compare Real and Pop</b>	
32-bit memory	11011 000 : mod 011 r/m
64-bit memory	11011 100 : mod 011 r/m
$ST(i)$	11011 000 : 11 011 ST(i)
<b>FCOMPP – Compare Real and Pop Twice</b>	11011 110 : 11 011 001
<b>FCOMIP – Compare Real, Set EFLAGS, and Pop</b>	11011 111 : 11 110 ST(i)
<b>FCOS – Cosine of <math>ST(0)</math></b>	11011 001 : 1111 1111
<b>FDECSTP – Decrement Stack-Top Pointer</b>	11011 001 : 1111 0110
<b>FDIV – Divide</b>	
$ST(0) \leftarrow ST(0) \div 32\text{-bit memory}$	11011 000 : mod 110 r/m
$ST(0) \leftarrow ST(0) \div 64\text{-bit memory}$	11011 100 : mod 110 r/m
$ST(d) \leftarrow ST(0) \div ST(i)$	11011 d00 : 1111 R ST(i)

# INSTRUCTION FORMATS AND ENCODINGS

**Table B-33. Floating-Point Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
<b>FDIVP – Divide and Pop</b> $ST(0) \leftarrow ST(0) \div ST(i)$	11011 110 : 1111 1 ST(i)
<b>FDIVR – Reverse Divide</b> $ST(0) \leftarrow 32\text{-bit memory} \div ST(0)$ $ST(0) \leftarrow 64\text{-bit memory} \div ST(0)$ $ST(d) \leftarrow ST(i) \div ST(0)$	11011 000 : mod 111 r/m 11011 100 : mod 111 r/m 11011 d00 : 1111 R ST(i)
<b>FDIVRP – Reverse Divide and Pop</b> $ST(0) \leftarrow ST(i) \div ST(0)$	11011 110 : 1111 0 ST(i)
<b>FFREE – Free ST(i) Register</b>	11011 101 : 1100 0 ST(i)
<b>FIADD – Add Integer</b> $ST(0) \leftarrow ST(0) + 16\text{-bit memory}$ $ST(0) \leftarrow ST(0) + 32\text{-bit memory}$	11011 110 : mod 000 r/m 11011 010 : mod 000 r/m
<b>FICOM – Compare Integer</b> 16-bit memory 32-bit memory	11011 110 : mod 010 r/m 11011 010 : mod 010 r/m
<b>FICOMP – Compare Integer and Pop</b> 16-bit memory 32-bit memory	11011 110 : mod 011 r/m 11011 010 : mod 011 r/m
<b>FIDIV</b> $ST(0) \leftarrow ST(0) \div 16\text{-bit memory}$ $ST(0) \leftarrow ST(0) \div 32\text{-bit memory}$	11011 110 : mod 110 r/m 11011 010 : mod 110 r/m
<b>FIDIVR</b> $ST(0) \leftarrow 16\text{-bit memory} \div ST(0)$ $ST(0) \leftarrow 32\text{-bit memory} \div ST(0)$	11011 110 : mod 111 r/m 11011 010 : mod 111 r/m
<b>FILD – Load Integer</b> 16-bit memory 32-bit memory 64-bit memory	11011 111 : mod 000 r/m 11011 011 : mod 000 r/m 11011 111 : mod 101 r/m
<b>FIMUL</b> $ST(0) \leftarrow ST(0) \times 16\text{-bit memory}$ $ST(0) \leftarrow ST(0) \times 32\text{-bit memory}$	11011 110 : mod 001 r/m 11011 010 : mod 001 r/m
<b>FINCSTP – Increment Stack Pointer</b>	11011 001 : 1111 0111
<b>FINIT – Initialize Floating-Point Unit</b>	



Table B-33. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
<b>FIST – Store Integer</b>	
16-bit memory	11011 111 : mod 010 r/m
32-bit memory	11011 011 : mod 010 r/m
<b>FISTP – Store Integer and Pop</b>	
16-bit memory	11011 111 : mod 011 r/m
32-bit memory	11011 011 : mod 011 r/m
64-bit memory	11011 111 : mod 111 r/m
<b>FISUB</b>	
ST(0) ← ST(0) - 16-bit memory	11011 110 : mod 100 r/m
ST(0) ← ST(0) - 32-bit memory	11011 010 : mod 100 r/m
<b>FISUBR</b>	
ST(0) ← 16-bit memory – ST(0)	11011 110 : mod 101 r/m
ST(0) ← 32-bit memory – ST(0)	11011 010 : mod 101 r/m
<b>FLD – Load Real</b>	
32-bit memory	11011 001 : mod 000 r/m
64-bit memory	11011 101 : mod 000 r/m
80-bit memory	11011 011 : mod 101 r/m
ST(i)	11011 001 : 11 000 ST(i)
<b>FLD1 – Load +1.0 into ST(0)</b>	11011 001 : 1110 1000
<b>FLDCW – Load Control Word</b>	11011 001 : mod 101 r/m
<b>FLDENV – Load FPU Environment</b>	11011 001 : mod 100 r/m
<b>FLDL2E – Load <math>\log_2(\epsilon)</math> into ST(0)</b>	11011 001 : 1110 1010
<b>FLDL2T – Load <math>\log_2(10)</math> into ST(0)</b>	11011 001 : 1110 1001
<b>FLDLG2 – Load <math>\log_{10}(2)</math> into ST(0)</b>	11011 001 : 1110 1100
<b>FLDLN2 – Load <math>\log_e(2)</math> into ST(0)</b>	11011 001 : 1110 1101
<b>FLDPI – Load <math>\pi</math> into ST(0)</b>	11011 001 : 1110 1011

## INSTRUCTION FORMATS AND ENCODINGS

**Table B-33. Floating-Point Instruction Formats and Encodings (Contd.)**

Instruction and Format	Encoding
<b>FLDZ – Load +0.0 into ST(0)</b>	11011 001 : 1110 1110
<b>FMUL – Multiply</b>	
ST(0) ← ST(0) × 32-bit memory	11011 000 : mod 001 r/m
ST(0) ← ST(0) × 64-bit memory	11011 100 : mod 001 r/m
ST(d) ← ST(0) × ST(i)	11011 d00 : 1100 1 ST(i)
<b>FMULP – Multiply</b>	
ST(i) ← ST(0) × ST(i)	11011 110 : 1100 1 ST(i)
<b>FNOP – No Operation</b>	11011 001 : 1101 0000
<b>FPATAN – Partial Arc tangent</b>	11011 001 : 1111 0011
<b>FPREM – Partial Remainder</b>	11011 001 : 1111 1000
<b>FPREM1 – Partial Remainder (IEEE)</b>	11011 001 : 1111 0101
<b>FPTAN – Partial Tangent</b>	11011 001 : 1111 0010
<b>FRNDINT – Round to Integer</b>	11011 001 : 1111 1100
<b>FRSTOR – Restore FPU State</b>	11011 101 : mod 100 r/m
<b>FSAVE – Store FPU State</b>	11011 101 : mod 110 r/m
<b>FSCALE – Scale</b>	11011 001 : 1111 1101
<b>FSIN – Sine</b>	11011 001 : 1111 1110
<b>FSINCOS – Sine and Cosine</b>	11011 001 : 1111 1011
<b>FSQRT – Square Root</b>	11011 001 : 1111 1010
<b>FST – Store Real</b>	
32-bit memory	11011 001 : mod 010 r/m
64-bit memory	11011 101 : mod 010 r/m
ST(i)	11011 101 : 11 010 ST(i)
<b>FSTCW – Store Control Word</b>	11011 001 : mod 111 r/m
<b>FSTENV – Store FPU Environment</b>	11011 001 : mod 110 r/m
<b>FSTP – Store Real and Pop</b>	
32-bit memory	11011 001 : mod 011 r/m
64-bit memory	11011 101 : mod 011 r/m
80-bit memory	11011 011 : mod 111 r/m
ST(i)	11011 101 : 11 011 ST(i)
<b>FSTSW – Store Status Word into AX</b>	11011 111 : 1110 0000
<b>FSTSW – Store Status Word into Memory</b>	11011 101 : mod 111 r/m

Table B-33. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
<b>FSUB – Subtract</b>	
ST(0) ← ST(0) – 32-bit memory	11011 000 : mod 100 r/m
ST(0) ← ST(0) – 64-bit memory	11011 100 : mod 100 r/m
ST(d) ← ST(0) – ST(i)	11011 d00 : 1110 R ST(i)
<b>FSUBP – Subtract and Pop</b>	
ST(0) ← ST(0) – ST(i)	11011 110 : 1110 1 ST(i)
<b>FSUBR – Reverse Subtract</b>	
ST(0) ← 32-bit memory – ST(0)	11011 000 : mod 101 r/m
ST(0) ← 64-bit memory – ST(0)	11011 100 : mod 101 r/m
ST(d) ← ST(i) – ST(0)	11011 d00 : 1110 R ST(i)
<b>FSUBRP – Reverse Subtract and Pop</b>	
ST(i) ← ST(i) – ST(0)	11011 110 : 1110 0 ST(i)
<b>FTST – Test</b>	11011 001 : 1110 0100
<b>FUCOM – Unordered Compare Real</b>	11011 101 : 1110 0 ST(i)
<b>FUCOMP – Unordered Compare Real and Pop</b>	11011 101 : 1110 1 ST(i)
<b>FUCOMPP – Unordered Compare Real and Pop Twice</b>	11011 010 : 1110 1001
<b>FUCOMI – Unorderd Compare Real and Set EFLAGS</b>	11011 011 : 11 101 ST(i)
<b>FUCOMIP – Unorderd Compare Real, Set EFLAGS, and Pop</b>	11011 111 : 11 101 ST(i)
<b>FXAM – Examine</b>	11011 001 : 1110 0101
<b>FXCH – Exchange ST(0) and ST(i)</b>	11011 001 : 1100 1 ST(i)
<b>EXTRACT – Extract Exponent and Significand</b>	11011 001 : 1111 0100
<b>FYL2X – ST(1) × log<sub>2</sub>(ST(0))</b>	11011 001 : 1111 0001
<b>FYL2XP1 – ST(1) × log<sub>2</sub>(ST(0) + 1.0)</b>	11011 001 : 1111 1001
<b>FWAIT – Wait until FPU Ready</b>	1001 1011

# INSTRUCTION FORMATS AND ENCODINGS

## B.12 VMX INSTRUCTIONS

Table B-34 describes virtual-machine extensions (VMX).

**Table B-34. Encodings for VMX Instructions**

Instruction and Format	Encoding
<b>VMCALL—Call to VM Monitor</b> Call VMM: causes VM exit.	00001111 00000001 11000001
<b>VMCLEAR—Clear Virtual-Machine Control Structure</b>  mem32:VMCS_data_ptr mem64:VMCS_data_ptr	01100110 00001111 11000111: mod 110 r/m 01100110 00001111 11000111: mod 110 r/m
<b>VMLAUNCH—Launch Virtual Machine</b> Launch VM managed by Current_VMCS	00001111 00000001 11000010
<b>VMRESUME—Resume Virtual Machine</b> Resume VM managed by Current_VMCS	00001111 00000001 11000011
<b>VMPTRLD—Load Pointer to Virtual-Machine Control Structure</b>  mem32 to Current_VMCS_ptr mem64 to Current_VMCS_ptr	00001111 11000111: mod 110 r/m 00001111 11000111: mod 110 r/m
<b>VMPTRST—Store Pointer to Virtual-Machine Control Structure</b>  Current_VMCS_ptr to mem32 Current_VMCS_ptr to mem64	00001111 11000111: mod 111 r/m 00001111 11000111: mod 111 r/m
<b>VMREAD—Read Field from Virtual-Machine Control Structure</b>  r32 (VMCS_fieldn) to r32 r32 (VMCS_fieldn) to mem32 r64 (VMCS_fieldn) to r64 r64 (VMCS_fieldn) to mem64	00001111 01111000: 11 reg2 reg1 00001111 01111000: mod r32 r/m 00001111 01111000: 11 reg2 reg1 00001111 01111000: mod r64 r/m
<b>VMWRITE—Write Field to Virtual-Machine Control Structure</b>  r32 to r32 (VMCS_fieldn) mem32 to r32 (VMCS_fieldn) r64 to r64 (VMCS_fieldn) mem64 to r64 (VMCS_fieldn)	00001111 01111001: 11 reg1 reg2 00001111 01111001: mod r32 r/m 00001111 01111001: 11 reg1 reg2 00001111 01111001: mod r64 r/m
<b>VMXOFF—Leave VMX Operation</b> Leave VMX.	00001111 00000001 11000100
<b>VMXON—Enter VMX Operation</b> Enter VMX.	11110011 00001111 11000111: mod 110 r/m

# C

## **Intel C/C++ Compiler Intrinsics and Functional Equivalents**



# APPENDIX C

## INTEL<sup>®</sup> C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

The two tables in this appendix itemize the Intel C/C++ compiler intrinsics and functional equivalents for the Intel MMX technology, SSE, SSE2, and SSE3 instructions.

There may be additional intrinsics that do not have an instruction equivalent. It is strongly recommended that the reader reference the compiler documentation for the complete list of supported intrinsics. Please refer to <http://www.intel.com/support/performancetools/>.

Table C-1 presents simple intrinsics and Table C-2 presents composite intrinsics. Some intrinsics are “composites” because they require more than one instruction to implement them.

Intel C/C++ Compiler intrinsic names reflect the following naming conventions:

`_mm_<intrin_op>_<suffix>`

where:

<code>&lt;intrin_op&gt;</code>	Indicates the intrinsics basic operation; for example, add for addition and sub for subtraction
<code>&lt;suffix&gt;</code>	Denotes the type of data operated on by the instruction. The first one or two letters of each suffix denotes whether the data is packed (p), extended packed (ep), or scalar (s). The remaining letters denote the type:
s	single-precision floating point
d	double-precision floating point
i128	signed 128-bit integer
i64	signed 64-bit integer
u64	unsigned 64-bit integer
i32	signed 32-bit integer
u32	unsigned 32-bit integer
i16	signed 16-bit integer
u16	unsigned 16-bit integer

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

i8 signed 8-bit integer

u8 unsigned 8-bit integer

The variable `r` is generally used for the intrinsic's return value. A number appended to a variable name indicates the element of a packed object. For example, `r0` is the lowest word of `r`.

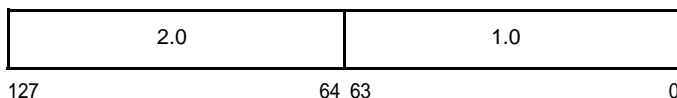
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
double a[2] = {1.0, 2.0};  
__m128d t = _mm_load_pd(a);
```

The result is the same as either of the following:

```
__m128d t = _mm_set_pd(2.0, 1.0);  
__m128d t = _mm_setr_pd(1.0, 2.0);
```

In other words, the XMM register that holds the value `t` will look as follows:



The “scalar” element is 1.0. Due to the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals).

To use an intrinsic in your code, insert a line with the following syntax:

```
data_type intrinsic_name (parameters)
```

Where:

<code>data_type</code>	Is the return data type, which can be either <code>void</code> , <code>int</code> , <code>__m64</code> , <code>__m128</code> , <code>__m128d</code> , or <code>__m128i</code> . Only the <code>_mm_empty</code> intrinsic returns <code>void</code> .
<code>intrinsic_name</code>	Is the name of the intrinsic, which behaves like a function that you can use in your C/C++ code instead of in-lining the actual instruction.
<code>parameters</code>	Represents the parameters required by each intrinsic.



## C.1 SIMPLE INTRINSICS

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic	Description
ADDPD	<code>__m128d _mm_add_pd(__m128d a, __m128d b)</code>	Adds the two DP FP (double-precision, floating-point) values of a and b.
ADDPS	<code>__m128 _mm_add_ps(__m128 a, __m128 b)</code>	Adds the four SP FP (single-precision, floating-point) values of a and b.
ADDSD	<code>__m128d _mm_add_sd(__m128d a, __m128d b)</code>	Adds the lower DP FP values of a and b; the upper three DP FP values are passed through from a.
ADDSS	<code>__m128 _mm_add_ss(__m128 a, __m128 b)</code>	Adds the lower SP FP values of a and b; the upper three SP FP values are passed through from a.
ADDSUBPD	<code>__m128d _mm_addsub_pd(__m128d a, __m128d b)</code>	Add/Subtract packed DP FP numbers from XMM2/Mem to XMM1.
ADDSUBPS	<code>__m128 _mm_addsub_ps(__m128 a, __m128 b)</code>	Add/Subtract packed SP FP numbers from XMM2/Mem to XMM1.
ANDNPD	<code>__m128d _mm_andnot_pd(__m128d a, __m128d b)</code>	Computes the bitwise AND-NOT of the two DP FP values of a and b.
ANDNPS	<code>__m128 _mm_andnot_ps(__m128 a, __m128 b)</code>	Computes the bitwise AND-NOT of the four SP FP values of a and b.
ANDPD	<code>__m128d _mm_and_pd(__m128d a, __m128d b)</code>	Computes the bitwise AND of the two DP FP values of a and b.
ANDPS	<code>__m128 _mm_and_ps(__m128 a, __m128 b)</code>	Computes the bitwise AND of the four SP FP values of a and b.
CLFLUSH	<code>void _mm_clflush(void const *p)</code>	Cache line containing p is flushed and invalidated from all caches in the coherency domain.
CMPPD	<code>__m128d _mm_cmpeq_pd(__m128d a, __m128d b)</code> <code>__m128d _mm_cmplt_pd(__m128d a, __m128d b)</code> <code>__m128d _mm_cmple_pd(__m128d a, __m128d b)</code> <code>__m128d _mm_cmpgt_pd(__m128d a, __m128d b)</code>	Compare for equality. Compare for less-than. Compare for less-than-or-equal. Compare for greater-than.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
	__m128d _mm_cmpge_pd(__m128d a, __m128d b) __m128d _mm_cmpneq_pd(__m128d a, __m128d b) __m128d _mm_cmpnlt_pd(__m128d a, __m128d b) __m128d _mm_cmpngt_pd(__m128d a, __m128d b) __m128d _mm_cmpnge_pd(__m128d a, __m128d b) __m128d _mm_cmpord_pd(__m128d a, __m128d b) __m128d _mm_cmpunord_pd(__m128d a, __m128d b) __m128d _mm_cmpnle_pd(__m128d a, __m128d b)	Compare for greater-than-or-equal. Compare for inequality. Compare for not-less-than. Compare for not-greater-than. Compare for not-greater-than-or-equal. Compare for ordered. Compare for unordered. Compare for not-less-than-or-equal.
CMPPS	__m128 _mm_cmpeq_ps(__m128 a, __m128 b) __m128 _mm_cmplt_ps(__m128 a, __m128 b) __m128 _mm_cmple_ps(__m128 a, __m128 b) __m128 _mm_cmpgt_ps(__m128 a, __m128 b) __m128 _mm_cmpge_ps(__m128 a, __m128 b) __m128 _mm_cmpneq_ps(__m128 a, __m128 b) __m128 _mm_cmpnlt_ps(__m128 a, __m128 b) __m128 _mm_cmpngt_ps(__m128 a, __m128 b) __m128 _mm_cmpnge_ps(__m128 a, __m128 b) __m128 _mm_cmpord_ps(__m128 a, __m128 b) __m128 _mm_cmpunord_ps(__m128 a, __m128 b) __m128 _mm_cmpnle_ps(__m128 a, __m128 b)	Compare for equality. Compare for less-than. Compare for less-than-or-equal. Compare for greater-than. Compare for greater-than-or-equal. Compare for inequality. Compare for not-less-than. Compare for not-greater-than. Compare for not-greater-than-or-equal. Compare for ordered. Compare for unordered. Compare for not-less-than-or-equal.
CMPSD	__m128d _mm_cmpeq_sd(__m128d a, __m128d b) __m128d _mm_cmplt_sd(__m128d a, __m128d b) __m128d _mm_cmple_sd(__m128d a, __m128d b) __m128d _mm_cmpgt_sd(__m128d a, __m128d b) __m128d _mm_cmpge_sd(__m128d a, __m128d b) __m128d _mm_cmpneq_sd(__m128d a, __m128d b) __m128d _mm_cmpnlt_sd(__m128d a, __m128d b)	Compare for equality. Compare for less-than. Compare for less-than-or-equal. Compare for greater-than. Compare for greater-than-or-equal. Compare for inequality. Compare for not-less-than.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
	__m128d __mm_cmpnle_sd(__m128d a, __m128d b)	Compare for not-greater-than.
	__m128d __mm_cmpngt_sd(__m128d a, __m128d b)	Compare for not-greater-than-or-equal.
	__m128d __mm_cmpnge_sd(__m128d a, __m128d b)	Compare for ordered.
	__m128d __mm_cmpord_sd(__m128d a, __m128d b)	Compare for unordered.
	__m128d __mm_cmpunord_sd(__m128d a, __m128d b)	Compare for not-less-than-or-equal.
CMPSS	__m128 __mm_cmpeq_ss(__m128 a, __m128 b)	Compare for equality.
	__m128 __mm_cmplt_ss(__m128 a, __m128 b)	Compare for less-than.
	__m128 __mm_cmple_ss(__m128 a, __m128 b)	Compare for less-than-or-equal.
	__m128 __mm_cmpgt_ss(__m128 a, __m128 b)	Compare for greater-than.
	__m128 __mm_cmpge_ss(__m128 a, __m128 b)	Compare for greater-than-or-equal.
	__m128 __mm_cmpneq_ss(__m128 a, __m128 b)	Compare for inequality.
	__m128 __mm_cmpnlt_ss(__m128 a, __m128 b)	Compare for not-less-than.
	__m128 __mm_cmpnle_ss(__m128 a, __m128 b)	Compare for not-greater-than.
	__m128 __mm_cmpngt_ss(__m128 a, __m128 b)	Compare for not-greater-than-or-equal.
	__m128 __mm_cmpnge_ss(__m128 a, __m128 b)	Compare for ordered.
__m128 __mm_cmpord_ss(__m128 a, __m128 b)	Compare for unordered.	
__m128 __mm_cmpunord_ss(__m128 a, __m128 b)	Compare for not-less-than-or-equal.	
COMISD	int __mm_comieq_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.
	int __mm_comilt_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.
	int __mm_comile_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
	int _mm_comigt_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.
	int _mm_comige_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_comineq_sd(__m128d a, __m128d b)	Compares the lower SDP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.
COMISS	int _mm_comieq_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.
	int _mm_comilt_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.
	int _mm_comile_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_comigt_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.
	int _mm_comige_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.
	int _mm_comineq_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
CVTDQ2PD	<code>__m128d _mm_cvtepi32_pd(__m128i a)</code>	Convert the lower two 32-bit signed integer values in packed form in a to two DP FP values.
CVTDQ2PS	<code>__m128 _mm_cvtepi32_ps(__m128i a)</code>	Convert the four 32-bit signed integer values in packed form in a to four SP FP values.
CVTPD2DQ	<code>__m128i _mm_cvtpd_epi32(__m128d a)</code>	Convert the two DP FP values in a to two 32-bit signed integer values.
CVTPD2PI	<code>__m64 _mm_cvtpd_pi32(__m128d a)</code>	Convert the two DP FP values in a to two 32-bit signed integer values.
CVTPD2PS	<code>__m128 _mm_cvtpd_ps(__m128d a)</code>	Convert the two DP FP values in a to two SP FP values.
CVTPI2PD	<code>__m128d _mm_cvtpi32_pd(__m64 a)</code>	Convert the two 32-bit integer values in a to two DP FP values
CVTPI2PS	<code>__m128 _mm_cvt_pi2ps(__m128 a, __m64 b)</code> <code>__m128 _mm_cvtpi32_ps(__m128 a, __m64 b)</code>	Convert the two 32-bit integer values in packed form in b to two SP FP values; the upper two SP FP values are passed through from a.
CVTPS2DQ	<code>__m128i _mm_cvtps_epi32(__m128 a)</code>	Convert four SP FP values in a to four 32-bit signed integers according to the current rounding mode.
CVTPS2PD	<code>__m128d _mm_cvtps_pd(__m128 a)</code>	Convert the lower two SP FP values in a to DP FP values.
CVTPS2PI	<code>__m64 _mm_cvt_ps2pi(__m128 a)</code> <code>__m64 _mm_cvtps_pi32(__m128 a)</code>	Convert the two lower SP FP values of a to two 32-bit integers according to the current rounding mode, returning the integers in packed form.
CVTSD2SI	<code>int _mm_cvtsd_si32(__m128d a)</code>	Convert the lower DP FP value in a to a 32-bit integer value.
CVTSD2SS	<code>__m128 _mm_cvtsd_ss(__m128 a, __m128d b)</code>	Convert the lower DP FP value in b to a SP FP value; the upper three SP FP values of a are passed through.
CVTSI2SD	<code>__m128d _mm_cvtsi32_sd(__m128d a, int b)</code>	Convert the 32-bit integer value b to a DP FP value; the upper DP FP values are passed through from a.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
CVTSS2SS	<code>__m128 __mm_cvt_si2ss(__m128 a, int b)</code> <code>__m128 __mm_cvtsi32_ss(__m128a, int b)</code>	Convert the 32-bit integer value b to an SP FP value; the upper three SP FP values are passed through from a.
CVTSS2SD	<code>__m128d __mm_cvtss_sd(__m128d a, __m128 b)</code>	Convert the lower SP FP value of b to DP FP value, the upper DP FP value is passed through from a.
CVTSS2SI	<code>int __mm_cvt_ss2si(__m128 a)</code> <code>int __mm_cvtss_si32(__m128 a)</code>	Convert the lower SP FP value of a to a 32-bit integer.
CVTTPD2DQ	<code>__m128i __mm_cvttpd_epi32(__m128d a)</code>	Convert the two DP FP values of a to two 32-bit signed integer values with truncation, the upper two integer values are 0.
CVTTPD2PI	<code>__m64 __mm_cvttpd_pi32(__m128d a)</code>	Convert the two DP FP values of a to 32-bit signed integer values with truncation.
CVTTPS2DQ	<code>__m128i __mm_cvttps_epi32(__m128 a)</code>	Convert four SP FP values of a to four 32-bit integer with truncation.
CVTTPS2PI	<code>__m64 __mm_cvtt_ps2pi(__m128 a)</code> <code>__m64 __mm_cvttps_pi32(__m128 a)</code>	Convert the two lower SP FP values of a to two 32-bit integer with truncation, returning the integers in packed form.
CVTTSD2SI	<code>int __mm_cvttsd_si32(__m128d a)</code>	Convert the lower DP FP value of a to a 32-bit signed integer using truncation.
CVTTSS2SI	<code>int __mm_cvtt_ss2si(__m128 a)</code> <code>int __mm_cvttss_si32(__m128 a)</code>  <code>__m64 __mm_cvtsi32_si64(int i)</code>  <code>int __mm_cvtsi64_si32(__m64 m)</code>	Convert the lower SP FP value of a to a 32-bit integer according to the current rounding mode.  Convert the integer object i to a 64-bit <code>__m64</code> object. The integer value is zero extended to 64 bits.  Convert the lower 32 bits of the <code>__m64</code> object m to an integer.
DIVPD	<code>__m128d __mm_div_pd(__m128d a, __m128d b)</code>	Divides the two DP FP values of a and b.
DIVPS	<code>__m128 __mm_div_ps(__m128 a, __m128 b)</code>	Divides the four SP FP values of a and b.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
DIVSD	<code>__m128d __mm_div_sd(__m128d a, __m128d b)</code>	Divides the lower DP FP values of a and b; the upper three DP FP values are passed through from a.
DIVSS	<code>__m128 __mm_div_ss(__m128 a, __m128 b)</code>	Divides the lower SP FP values of a and b; the upper three SP FP values are passed through from a.
EMMS	<code>void __mm_empty()</code>	Clears the MMX technology state.
HADDPD	<code>__m128d __mm_hadd_pd(__m128d a, __m128d b)</code>	Add horizontally packed DP FP numbers from XMM2/Mem to XMM1
HADDPS	<code>__m128 __mm_hadd_ps(__m128 a, __m128 b)</code>	Add horizontally packed SP FP numbers from XMM2/Mem to XMM1
HSUBPD	<code>__m128d __mm_hsub_pd(__m128d a, __m128d b)</code>	Subtract horizontally packed DP FP numbers in XMM2/Mem from XMM1.
HSUBPS	<code>__m128 __mm_hsub_ps(__m128 a, __m128 b)</code>	Subtract horizontally packed SP FP numbers in XMM2/Mem from XMM1.
LDDQU	<code>__m128i __mm_lddqu_si128(__m128i const *p)</code>	Load 128 bits from Mem to XMM register.
LDMXCSR	<code>__mm_setcsr(unsigned int i)</code>	Sets the control register to the value specified.
LFENCE	<code>void __mm_lfence(void)</code>	Guaranteed that every load that proceeds, in program order, the load fence instruction is globally visible before any load instruction that follows the fence in program order.
MASKMOVDQU	<code>void __mm_maskmoveu_si128(__m128i d, __m128i n, char *p)</code>	Conditionally store byte elements of d to address p. The high bit of each byte in the selector n determines whether the corresponding byte in d will be stored.
MASKMOVQ	<code>void __mm_maskmove_si64(__m64 d, __m64 n, char *p)</code>	Conditionally store byte elements of d to address p. The high bit of each byte in the selector n determines whether the corresponding byte in d will be stored.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
MAXPD	<code>__m128d __mm_max_pd(__m128d a, __m128d b)</code>	Computes the maximums of the two DP FP values of a and b.
MAXPS	<code>__m128 __mm_max_ps(__m128 a, __m128 b)</code>	Computes the maximums of the four SP FP values of a and b.
MAXSD	<code>__m128d __mm_max_sd(__m128d a, __m128d b)</code>	Computes the maximum of the lower DP FP values of a and b; the upper DP FP values are passed through from a.
MAXSS	<code>__m128 __mm_max_ss(__m128 a, __m128 b)</code>	Computes the maximum of the lower SP FP values of a and b; the upper three SP FP values are passed through from a.
MFENCE	<code>void __mm_mfence(void)</code>	Guaranteed that every memory access that proceeds, in program order, the memory fence instruction is globally visible before any memory instruction that follows the fence in program order.
MINPD	<code>__m128d __mm_min_pd(__m128d a, __m128d b)</code>	Computes the minimums of the two DP FP values of a and b.
MINPS	<code>__m128 __mm_min_ps(__m128 a, __m128 b)</code>	Computes the minimums of the four SP FP values of a and b.
MINSD	<code>__m128d __mm_min_sd(__m128d a, __m128d b)</code>	Computes the minimum of the lower DP FP values of a and b; the upper DP FP values are passed through from a.
MINSS	<code>__m128 __mm_min_ss(__m128 a, __m128 b)</code>	Computes the minimum of the lower SP FP values of a and b; the upper three SP FP values are passed through from a.
MONITOR	<code>void __mm_monitor(void const *p, unsigned extensions, unsigned hints)</code>	Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be of a write-back memory caching type.
MOVAPD	<code>__m128d __mm_load_pd(double * p)</code>	Loads two DP FP values. The address p must be 16-byte-aligned.



## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
	<code>void_mm_store_pd(double *p, __m128d a)</code>	Stores two DP FP values to address p. The address p must be 16-byte-aligned.
MOVAPS	<code>__m128_mm_load_ps(float * p)</code>  <code>void_mm_store_ps(float *p, __m128 a)</code>	Loads four SP FP values. The address p must be 16-byte-aligned.  Stores four SP FP values. The address p must be 16-byte-aligned.
MOVD	<code>__m128i_mm_cvtsi32_si128(int a)</code>  <code>int_mm_cvtsi128_si32(__m128i a)</code>  <code>__m64_mm_cvtsi32_si64(int a)</code>  <code>int_mm_cvtsi64_si32(__m64 a)</code>	Moves 32-bit integer a to the lower 32-bit of the 128-bit destination, while zero-extending the upper bits.  Moves lower 32-bit integer of a to a 32-bit signed integer.  Moves 32-bit integer a to the lower 32-bit of the 64-bit destination, while zero-extending the upper bits.  Moves lower 32-bit integer of a to a 32-bit signed integer.
MOVDDUP	<code>__m128d_mm_movedup_pd(__m128d a)</code> <code>__m128d_mm_loadup_pd(double const * dp)</code>	Move 64 bits representing the lower DP data element from XMM2/Mem to XMM1 register and duplicate.
MOVDQA	<code>__m128i_mm_load_si128(__m128i * p)</code>  <code>void_mm_store_si128(__m128i *p, __m128i a)</code>	Loads 128-bit values from p. The address p must be 16-byte-aligned.  Stores 128-bit value in a to address p. The address p must be 16-byte-aligned.
MOVDQU	<code>__m128i_mm_loadu_si128(__m128i * p)</code>  <code>void_mm_storeu_si128(__m128i *p, __m128i a)</code>	Loads 128-bit values from p. The address p need not be 16-byte-aligned.  Stores 128-bit value in a to address p. The address p need not be 16-byte-aligned.
MOVDQ2Q	<code>__m64_mm_movepi64_pi64(__m128i a)</code>	Return the lower 64-bits in a as __m64 type.
MOVHPS	<code>__m128_mm_movehl_ps(__m128 a, __m128 b)</code>	Moves the upper 2 SP FP values of b to the lower 2 SP FP values of the result. The upper 2 SP FP values of a are passed through to the result.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
MOVHPD	<p>__m128d _mm_loadh_pd(__m128d a, double * p)</p> <p>void _mm_storeh_pd(double * p, __m128d a)</p>	<p>Load a DP FP value from the address p to the upper 64 bits of destination; the lower 64 bits are passed through from a.</p> <p>Stores the upper DP FP value of a to the address p.</p>
MOVHPS	<p>__m128 _mm_loadh_pi(__m128 a, __m64 * p)</p> <p>void _mm_storeh_pi(__m64 * p, __m128 a)</p>	<p>Sets the upper two SP FP values with 64 bits of data loaded from the address p; the lower two values are passed through from a.</p> <p>Stores the upper two SP FP values of a to the address p.</p>
MOVLPD	<p>__m128d _mm_loadl_pd(__m128d a, double * p)</p> <p>void _mm_storel_pd(double * p, __m128d a)</p>	<p>Load a DP FP value from the address p to the lower 64 bits of destination; the upper 64 bits are passed through from a.</p> <p>Stores the lower DP FP value of a to the address p.</p>
MOVLPS	<p>__m128 _mm_loadl_pi(__m128 a, __m64 * p)</p> <p>void _mm_storel_pi(__m64 * p, __m128 a)</p>	<p>Sets the lower two SP FP values with 64 bits of data loaded from the address p; the upper two values are passed through from a.</p> <p>Stores the lower two SP FP values of a to the address p.</p>
MOVLHPS	<p>__m128 _mm_movelh_ps(__m128 a, __m128 b)</p>	<p>Moves the lower 2 SP FP values of b to the upper 2 SP FP values of the result. The lower 2 SP FP values of a are passed through to the result.</p>
MOVMSKPD	int _mm_movemask_pd(__m128d a)	Creates a 2-bit mask from the sign bits of the two DP FP values of a.
MOVMSKPS	int _mm_movemask_ps(__m128 a)	Creates a 4-bit mask from the most significant bits of the four SP FP values.
MOVNTDQ	void _mm_stream_si128(__m128i * p, __m128i a)	Stores the data in a to the address p without polluting the caches. If the cache line containing p is already in the cache, the cache will be updated. The address must be 16-byte-aligned.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
MOVNTPD	void_mm_stream_pd(double * p, __m128d a)	Stores the data in a to the address p without polluting the caches. The address must be 16-byte-aligned.
MOVNTPS	void_mm_stream_ps(float * p, __m128 a)	Stores the data in a to the address p without polluting the caches. The address must be 16-byte-aligned.
MOVNTI	void_mm_stream_si32(int * p, int a)	Stores the data in a to the address p without polluting the caches.
MOVNTQ	void_mm_stream_pi(__m64 * p, __m64 a)	Stores the data in a to the address p without polluting the caches.
MOVQ	__m128i _mm_loadl_epi64(__m128i * p)	Loads the lower 64 bits from p into the lower 64 bits of destination and zero-extend the upper 64 bits.
	void_mm_storel_epi64(__m128i * p, __m128i a)	Stores the lower 64 bits of a to the lower 64 bits at p.
	__m128i _mm_move_epi64(__m128i a)	Moves the lower 64 bits of a to the lower 64 bits of destination. The upper 64 bits are cleared.
MOVQ2DQ	__m128i _mm_movpi64_epi64(__m64 a)	Move the 64 bits of a into the lower 64-bits, while zero-extending the upper bits.
MOVSD	__m128d _mm_load_sd(double * p)	Loads a DP FP value from p into the lower DP FP value and clears the upper DP FP value. The address P need not be 16-byte aligned.
	void_mm_store_sd(double * p, __m128d a)	Stores the lower DP FP value of a to address p. The address P need not be 16-byte aligned.
	__m128d _mm_move_sd(__m128d a, __m128d b)	Sets the lower DP FP values of b to destination. The upper DP FP value is passed through from a.
MOVSHDUP	__m128 _mm_movehdup_ps(__m128 a)	Move 128 bits representing packed SP data elements from XMM2/Mem to XMM1 register and duplicate high.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
MOVSLDUP	<code>__m128 _mm_moveldup_ps(__m128 a)</code>	Move 128 bits representing packed SP data elements from XMM2/Mem to XMM1 register and duplicate low.
MOVSS	<code>__m128 _mm_load_ss(float * p)</code>  <code>void _mm_store_ss(float * p, __m128 a)</code> <code>__m128 _mm_move_ss(__m128 a, __m128 b)</code>	Loads an SP FP value into the low word and clears the upper three words.  Stores the lower SP FP value.  Sets the low word to the SP FP value of b. The upper 3 SP FP values are passed through from a.
MOVUPD	<code>__m128d _mm_loadu_pd(double * p)</code>  <code>void _mm_storeu_pd(double *p, __m128d a)</code>	Loads two DP FP values from p. The address p need not be 16-byte-aligned.  Stores two DP FP values in a to p. The address p need not be 16-byte-aligned.
MOVUPS	<code>__m128 _mm_loadu_ps(float * p)</code>  <code>void _mm_storeu_ps(float *p, __m128 a)</code>	Loads four SP FP values. The address need not be 16-byte-aligned.  Stores four SP FP values. The address need not be 16-byte-aligned.
MULPD	<code>__m128d _mm_mul_pd(__m128d a, __m128d b)</code>	Multiplies the two DP FP values of a and b.
MULPS	<code>__m128 _mm_mul_ss(__m128 a, __m128 b)</code>	Multiplies the four SP FP value of a and b.
MULSD	<code>__m128d _mm_mul_sd(__m128d a, __m128d b)</code>	Multiplies the lower DP FP value of a and b; the upper DP FP value are passed through from a.
MULSS	<code>__m128 _mm_mul_ss(__m128 a, __m128 b)</code>	Multiplies the lower SP FP value of a and b; the upper three SP FP values are passed through from a.
MWAIT	<code>void _mm_mwait(unsigned extensions, unsigned hints)</code>	A hint that allows the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.
ORPD	<code>__m128d _mm_or_pd(__m128d a, __m128d b)</code>	Computes the bitwise OR of the two DP FP values of a and b.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
ORPS	<code>__m128 __mm_or_ps(__m128 a, __m128 b)</code>	Computes the bitwise OR of the four SP FP values of a and b.
PACKSSWB	<code>__m128i __mm_packs_epi16(__m128i m1, __m128i m2)</code>	Pack the eight 16-bit values from m1 into the lower eight 8-bit values of the result with signed saturation, and pack the eight 16-bit values from m2 into the upper eight 8-bit values of the result with signed saturation.
PACKSSWB	<code>__m64 __mm_packs_pi16(__m64 m1, __m64 m2)</code>	Pack the four 16-bit values from m1 into the lower four 8-bit values of the result with signed saturation, and pack the four 16-bit values from m2 into the upper four 8-bit values of the result with signed saturation.
PACKSSDW	<code>__m128i __mm_packs_epi32 (__m128i m1, __m128i m2)</code>	Pack the four 32-bit values from m1 into the lower four 16-bit values of the result with signed saturation, and pack the four 32-bit values from m2 into the upper four 16-bit values of the result with signed saturation.
PACKSSDW	<code>__m64 __mm_packs_pi32 (__m64 m1, __m64 m2)</code>	Pack the two 32-bit values from m1 into the lower two 16-bit values of the result with signed saturation, and pack the two 32-bit values from m2 into the upper two 16-bit values of the result with signed saturation.
PACKUSWB	<code>__m128i __mm_packus_epi16(__m128i m1, __m128i m2)</code>	Pack the eight 16-bit values from m1 into the lower eight 8-bit values of the result with unsigned saturation, and pack the eight 16-bit values from m2 into the upper eight 8-bit values of the result with unsigned saturation.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
PACKUSWB	<code>__m64 _mm_packs_pu16(__m64 m1, __m64 m2)</code>	Pack the four 16-bit values from m1 into the lower four 8-bit values of the result with unsigned saturation, and pack the four 16-bit values from m2 into the upper four 8-bit values of the result with unsigned saturation.
PADDB	<code>__m128i _mm_add_epi8(__m128i m1, __m128i m2)</code>	Add the 16 8-bit values in m1 to the 16 8-bit values in m2.
PADDB	<code>__m64 _mm_add_pi8(__m64 m1, __m64 m2)</code>	Add the eight 8-bit values in m1 to the eight 8-bit values in m2.
PADDW	<code>__m128i _mm_addw_epi16(__m128i m1, __m128i m2)</code>	Add the 8 16-bit values in m1 to the 8 16-bit values in m2.
PADDW	<code>__m64 _mm_addw_pi16(__m64 m1, __m64 m2)</code>	Add the four 16-bit values in m1 to the four 16-bit values in m2.
PADDD	<code>__m128i _mm_add_epi32(__m128i m1, __m128i m2)</code>	Add the 4 32-bit values in m1 to the 4 32-bit values in m2.
PADDD	<code>__m64 _mm_add_pi32(__m64 m1, __m64 m2)</code>	Add the two 32-bit values in m1 to the two 32-bit values in m2.
PADDQ	<code>__m128i _mm_add_epi64(__m128i m1, __m128i m2)</code>	Add the 2 64-bit values in m1 to the 2 64-bit values in m2.
PADDQ	<code>__m64 _mm_add_si64(__m64 m1, __m64 m2)</code>	Add the 64-bit value in m1 to the 64-bit value in m2.
PADDSB	<code>__m128i _mm_adds_epi8(__m128i m1, __m128i m2)</code>	Add the 16 signed 8-bit values in m1 to the 16 signed 8-bit values in m2 and saturate.
PADDSB	<code>__m64 _mm_adds_pi8(__m64 m1, __m64 m2)</code>	Add the eight signed 8-bit values in m1 to the eight signed 8-bit values in m2 and saturate.
PADDSW	<code>__m128i _mm_adds_epi16(__m128i m1, __m128i m2)</code>	Add the 8 signed 16-bit values in m1 to the 8 signed 16-bit values in m2 and saturate.
PADDSW	<code>__m64 _mm_adds_pi16(__m64 m1, __m64 m2)</code>	Add the four signed 16-bit values in m1 to the four signed 16-bit values in m2 and saturate.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
PADDUSB	<code>__m128i _mm_adds_epu8(__m128i m1, __m128i m2)</code>	Add the 16 unsigned 8-bit values in m1 to the 16 unsigned 8-bit values in m2 and saturate.
PADDUSB	<code>__m64 _mm_adds_pu8(__m64 m1, __m64 m2)</code>	Add the eight unsigned 8-bit values in m1 to the eight unsigned 8-bit values in m2 and saturate.
PADDUSW	<code>__m128i _mm_adds_epu16(__m128i m1, __m128i m2)</code>	Add the 8 unsigned 16-bit values in m1 to the 8 unsigned 16-bit values in m2 and saturate.
PADDUSW	<code>__m64 _mm_adds_pu16(__m64 m1, __m64 m2)</code>	Add the four unsigned 16-bit values in m1 to the four unsigned 16-bit values in m2 and saturate.
PAND	<code>__m128i _mm_and_si128(__m128i m1, __m128i m2)</code>	Perform a bitwise AND of the 128-bit value in m1 with the 128-bit value in m2.
PAND	<code>__m64 _mm_and_si64(__m64 m1, __m64 m2)</code>	Perform a bitwise AND of the 64-bit value in m1 with the 64-bit value in m2.
PANDN	<code>__m128i _mm_andnot_si128(__m128i m1, __m128i m2)</code>	Perform a logical NOT on the 128-bit value in m1 and use the result in a bitwise AND with the 128-bit value in m2.
PANDN	<code>__m64 _mm_andnot_si64(__m64 m1, __m64 m2)</code>	Perform a logical NOT on the 64-bit value in m1 and use the result in a bitwise AND with the 64-bit value in m2.
PAUSE	<code>void _mm_pause(void)</code>	The execution of the next instruction is delayed by an implementation-specific amount of time. No architectural state is modified.
PAVGB	<code>__m128i _mm_avg_epu8(__m128i a, __m128i b)</code>	Perform the packed average on the 16 8-bit values of the two operands.
PAVGB	<code>__m64 _mm_avg_pu8(__m64 a, __m64 b)</code>	Perform the packed average on the eight 8-bit values of the two operands.
PAVGW	<code>__m128i _mm_avg_epu16(__m128i a, __m128i b)</code>	Perform the packed average on the 8 16-bit values of the two operands.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
PAVGW	<code>__m64 _mm_avg_pu16(__m64 a, __m64 b)</code>	Perform the packed average on the four 16-bit values of the two operands.
PCMPEQB	<code>__m128i _mm_cmpeq_epi8(__m128i m1, __m128i m2)</code>	If the respective 8-bit values in m1 are equal to the respective 8-bit values in m2 set the respective 8-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQB	<code>__m64 _mm_cmpeq_pi8(__m64 m1, __m64 m2)</code>	If the respective 8-bit values in m1 are equal to the respective 8-bit values in m2 set the respective 8-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQW	<code>__m128i _mm_cmpeq_epi16 (__m128i m1, __m128i m2)</code>	If the respective 16-bit values in m1 are equal to the respective 16-bit values in m2 set the respective 16-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQW	<code>__m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)</code>	If the respective 16-bit values in m1 are equal to the respective 16-bit values in m2 set the respective 16-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQD	<code>__m128i _mm_cmpeq_epi32(__m128i m1, __m128i m2)</code>	If the respective 32-bit values in m1 are equal to the respective 32-bit values in m2 set the respective 32-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPEQD	<code>__m64 _mm_cmpeq_pi32(__m64 m1, __m64 m2)</code>	If the respective 32-bit values in m1 are equal to the respective 32-bit values in m2 set the respective 32-bit resulting values to all ones, otherwise set them to all zeroes.



## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
PCMPGTB	<code>__m128i _mm_cmpgt_epi8 (__m128i m1, __m128i m2)</code>	If the respective 8-bit values in m1 are greater than the respective 8-bit values in m2 set the respective 8-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPGTB	<code>__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)</code>	If the respective 8-bit values in m1 are greater than the respective 8-bit values in m2 set the respective 8-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPGTW	<code>__m128i _mm_cmpgt_epi16(__m128i m1, __m128i m2)</code>	If the respective 16-bit values in m1 are greater than the respective 16-bit values in m2 set the respective 16-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPGTW	<code>__m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)</code>	If the respective 16-bit values in m1 are greater than the respective 16-bit values in m2 set the respective 16-bit resulting values to all ones, otherwise set them to all zeroes.
PCMPGTD	<code>__m128i _mm_cmpgt_epi32(__m128i m1, __m128i m2)</code>	If the respective 32-bit values in m1 are greater than the respective 32-bit values in m2 set the respective 32-bit resulting values to all ones, otherwise set them all to zeroes.
PCMPGTD	<code>__m64 _mm_cmpgt_pi32(__m64 m1, __m64 m2)</code>	If the respective 32-bit values in m1 are greater than the respective 32-bit values in m2 set the respective 32-bit resulting values to all ones, otherwise set them all to zeroes.
PEXTRW	<code>int _mm_extract_epi16(__m128i a, int n)</code>	Extracts one of the 8 words of a. The selector n must be an immediate.
PEXTRW	<code>int _mm_extract_pi16(__m64 a, int n)</code>	Extracts one of the four words of a. The selector n must be an immediate.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
PINSRW	<code>__m128i _mm_insert_epi16(__m128i a, int d, int n)</code>	Inserts word d into one of 8 words of a. The selector n must be an immediate.
PINSRW	<code>__m64 _mm_insert_pi16(__m64 a, int d, int n)</code>	Inserts word d into one of four words of a. The selector n must be an immediate.
PMADDWD	<code>__m128i _mm_madd_epi16(__m128i m1 __m128i m2)</code>	Multiply 8 16-bit values in m1 by 8 16-bit values in m2 producing 8 32-bit intermediate results, which are then summed by pairs to produce 4 32-bit results.
PMADDWD	<code>__m64 _mm_madd_pi16(__m64 m1, __m64 m2)</code>	Multiply four 16-bit values in m1 by four 16-bit values in m2 producing four 32-bit intermediate results, which are then summed by pairs to produce two 32-bit results.
PMAXSW	<code>__m128i _mm_max_epi16(__m128i a, __m128i b)</code>	Computes the element-wise maximum of the 16-bit integers in a and b.
PMAXSW	<code>__m64 _mm_max_pi16(__m64 a, __m64 b)</code>	Computes the element-wise maximum of the words in a and b.
PMAXUB	<code>__m128i _mm_max_epu8(__m128i a, __m128i b)</code>	Computes the element-wise maximum of the unsigned bytes in a and b.
PMAXUB	<code>__m64 _mm_max_pu8(__m64 a, __m64 b)</code>	Computes the element-wise maximum of the unsigned bytes in a and b.
PMINSW	<code>__m128i _mm_min_epi16(__m128i a, __m128i b)</code>	Computes the element-wise minimum of the 16-bit integers in a and b.
PMINSW	<code>__m64 _mm_min_pi16(__m64 a, __m64 b)</code>	Computes the element-wise minimum of the words in a and b.
PMINUB	<code>__m128i _mm_min_epu8(__m128i a, __m128i b)</code>	Computes the element-wise minimum of the unsigned bytes in a and b.
PMINUB	<code>__m64 _mm_min_pu8(__m64 a, __m64 b)</code>	Computes the element-wise minimum of the unsigned bytes in a and b.
PMOVMASK	<code>int _mm_movemask_epi8(__m128i a)</code>	Creates an 16-bit mask from the most significant bits of the bytes in a.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
PMOVMSKB	<code>int __mm_movemask_pi8(__m64 a)</code>	Creates an 8-bit mask from the most significant bits of the bytes in a.
PMULHUW	<code>__m128i __mm_mulhi_epu16(__m128i a, __m128i b)</code>	Multiplies the 8 unsigned words in a and b, returning the upper 16 bits of the eight 32-bit intermediate results in packed form.
PMULHUW	<code>__m64 __mm_mulhi_pu16(__m64 a, __m64 b)</code>	Multiplies the 4 unsigned words in a and b, returning the upper 16 bits of the four 32-bit intermediate results in packed form.
PMULHW	<code>__m128i __mm_mulhi_epi16(__m128i m1, __m128i m2)</code>	Multiply 8 signed 16-bit values in m1 by 8 signed 16-bit values in m2 and produce the high 16 bits of the 8 results.
PMULHW	<code>__m64 __mm_mulhi_pi16(__m64 m1, __m64 m2)</code>	Multiply four signed 16-bit values in m1 by four signed 16-bit values in m2 and produce the high 16 bits of the four results.
PMULLW	<code>__m128i __mm_mullo_epi16(__m128i m1, __m128i m2)</code>	Multiply 8 16-bit values in m1 by 8 16-bit values in m2 and produce the low 16 bits of the 8 results.
PMULLW	<code>__m64 __mm_mullo_pi16(__m64 m1, __m64 m2)</code>	Multiply four 16-bit values in m1 by four 16-bit values in m2 and produce the low 16 bits of the four results.
PMULUDQ	<code>__m64 __mm_mul_su32(__m64 m1, __m64 m2)</code>  <code>__m128i __mm_mul_epu32(__m128i m1, __m128i m2)</code>	<p>Multiply lower 32-bit unsigned value in m1 by the lower 32-bit unsigned value in m2 and store the 64 bit results.</p> <p>Multiply lower two 32-bit unsigned value in m1 by the lower two 32-bit unsigned value in m2 and store the two 64 bit results.</p>
POR	<code>__m64 __mm_or_si64(__m64 m1, __m64 m2)</code>	Perform a bitwise OR of the 64-bit value in m1 with the 64-bit value in m2.
POR	<code>__m128i __mm_or_si128(__m128i m1, __m128i m2)</code>	Perform a bitwise OR of the 128-bit value in m1 with the 128-bit value in m2.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
PREFETCHh	<code>void __mm_prefetch(char *a, int sel)</code>	Loads one cache line of data from address p to a location “closer” to the processor. The value sel specifies the type of prefetch operation.
PSADBW	<code>__m128i __mm_sad_epu8(__m128i a, __m128i b)</code>	Compute the absolute differences of the 16 unsigned 8-bit values of a and b; sum the upper and lower 8 differences and store the two 16-bit result into the upper and lower 64 bit.
PSADBW	<code>__m64 __mm_sad_pu8(__m64 a, __m64 b)</code>	Compute the absolute differences of the 8 unsigned 8-bit values of a and b; sum the 8 differences and store the 16-bit result, the upper 3 words are cleared.
PSHUFD	<code>__m128i __mm_shuffle_epi32(__m128i a, int n)</code>	Returns a combination of the four doublewords of a. The selector n must be an immediate.
PSHUFHW	<code>__m128i __mm_shufflehi_epi16(__m128i a, int n)</code>	Shuffle the upper four 16-bit words in a as specified by n. The selector n must be an immediate.
PSHUFLW	<code>__m128i __mm_shufflelo_epi16(__m128i a, int n)</code>	Shuffle the lower four 16-bit words in a as specified by n. The selector n must be an immediate.
PSHUFW	<code>__m64 __mm_shuffle_pi16(__m64 a, int n)</code>	Returns a combination of the four words of a. The selector n must be an immediate.
PSLLW	<code>__m128i __mm_sll_epi16(__m128i m, __m128i count)</code>	Shift each of 8 16-bit values in m left the amount specified by count while shifting in zeroes.
PSLLW	<code>__m128i __mm_slli_epi16(__m128i m, int count)</code>	Shift each of 8 16-bit values in m left the amount specified by count while shifting in zeroes.
PSLLW	<code>__m64 __mm_sll_pi16(__m64 m, __m64 count)</code>	Shift four 16-bit values in m left the amount specified by count while shifting in zeroes. For the best performance, count should be a constant.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
	<code>__m64 __mm_slli_pi16(__m64 m, int count)</code>	Shift four 16-bit values in <code>m</code> left the amount specified by <code>count</code> while shifting in zeroes. For the best performance, <code>count</code> should be a constant.
PSLLD	<code>__m128i __mm_slli_epi32(__m128i m, int count)</code>	Shift each of 4 32-bit values in <code>m</code> left the amount specified by <code>count</code> while shifting in zeroes.
	<code>__m128i __mm_sll_epi32(__m128i m, __m128i count)</code>	Shift each of 4 32-bit values in <code>m</code> left the amount specified by <code>count</code> while shifting in zeroes. For the best performance, <code>count</code> should be a constant.
PSLLD	<code>__m64 __mm_slli_pi32(__m64 m, int count)</code>	Shift two 32-bit values in <code>m</code> left the amount specified by <code>count</code> while shifting in zeroes.
	<code>__m64 __mm_sll_pi32(__m64 m, __m64 count)</code>	Shift two 32-bit values in <code>m</code> left the amount specified by <code>count</code> while shifting in zeroes. For the best performance, <code>count</code> should be a constant.
PSLLQ	<code>__m64 __mm_sll_si64(__m64 m, __m64 count)</code>	Shift the 64-bit value in <code>m</code> left the amount specified by <code>count</code> while shifting in zeroes.
	<code>__m64 __mm_slli_si64(__m64 m, int count)</code>	Shift the 64-bit value in <code>m</code> left the amount specified by <code>count</code> while shifting in zeroes. For the best performance, <code>count</code> should be a constant.
PSLLQ	<code>__m128i __mm_sll_epi64(__m128i m, __m128i count)</code>	Shift each of two 64-bit values in <code>m</code> left by the amount specified by <code>count</code> while shifting in zeroes.
	<code>__m128i __mm_slli_epi64(__m128i m, int count)</code>	Shift each of two 64-bit values in <code>m</code> left by the amount specified by <code>count</code> while shifting in zeroes. For the best performance, <code>count</code> should be a constant.
PSLLDQ	<code>__m128i __mm_slli_si128(__m128i m, int imm)</code>	Shift 128 bit in <code>m</code> left by <code>imm</code> bytes while shifting in zeroes.
PSRAW	<code>__m128i __mm_sra_epi16(__m128i m, __m128i count)</code>	Shift each of 8 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
	<code>__m128i __mm_srai_epi16(__m128i m, int count)</code>	Shift each of 8 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit. For the best performance, <code>count</code> should be a constant.
PSRAW	<code>__m64 __mm_sra_pi16(__m64 m, __m64 count)</code>  <code>__m64 __mm_srai_pi16(__m64 m, int count)</code>	Shift four 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit.  Shift four 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit. For the best performance, <code>count</code> should be a constant.
PSRAD	<code>__m128i __mm_sra_epi32 (__m128i m, __m128i count)</code>  <code>__m128i __mm_srai_epi32 (__m128i m, int count)</code>	Shift each of 4 32-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit.  Shift each of 4 32-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit. For the best performance, <code>count</code> should be a constant.
PSRAD	<code>__m64 __mm_sra_pi32 (__m64 m, __m64 count)</code>  <code>__m64 __mm_srai_pi32 (__m64 m, int count)</code>	Shift two 32-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit.  Shift two 32-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in the sign bit. For the best performance, <code>count</code> should be a constant.
PSRLW	<code>__m128i __mm_srl_epi16 (__m128i m, __m128i count)</code>  <code>__m128i __mm_srli_epi16 (__m128i m, int count)</code>  <code>__m64 __mm_srl_pi16 (__m64 m, __m64 count)</code>	Shift each of 8 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes.  Shift each of 8 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes.  Shift four 16-bit values in <code>m</code> right the amount specified by <code>count</code> while shifting in zeroes.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
	<code>__m64 __mm_srli_pi16(__m64 m, int count)</code>	Shift four 16-bit values in <i>m</i> right the amount specified by <i>count</i> while shifting in zeroes. For the best performance, <i>count</i> should be a constant.
PSRLD	<code>__m128i __mm_srl_epi32 (__m128i m, __m128i count)</code>  <code>__m128i __mm_srli_epi32 (__m128i m, int count)</code>	Shift each of 4 32-bit values in <i>m</i> right the amount specified by <i>count</i> while shifting in zeroes.  Shift each of 4 32-bit values in <i>m</i> right the amount specified by <i>count</i> while shifting in zeroes. For the best performance, <i>count</i> should be a constant.
PSRLD	<code>__m64 __mm_srl_pi32 (__m64 m, __m64 count)</code>  <code>__m64 __mm_srli_pi32 (__m64 m, int count)</code>	Shift two 32-bit values in <i>m</i> right the amount specified by <i>count</i> while shifting in zeroes.  Shift two 32-bit values in <i>m</i> right the amount specified by <i>count</i> while shifting in zeroes. For the best performance, <i>count</i> should be a constant.
PSRLQ	<code>__m128i __mm_srl_epi64 (__m128i m, __m128i count)</code>	Shift the 2 64-bit value in <i>m</i> right the amount specified by <i>count</i> while shifting in zeroes.
	<code>__m128i __mm_srli_epi64 (__m128i m, int count)</code>	Shift the 2 64-bit value in <i>m</i> right the amount specified by <i>count</i> while shifting in zeroes. For the best performance, <i>count</i> should be a constant.
PSRLQ	<code>__m64 __mm_srl_si64 (__m64 m, __m64 count)</code>  <code>__m64 __mm_srli_si64 (__m64 m, int count)</code>	Shift the 64-bit value in <i>m</i> right the amount specified by <i>count</i> while shifting in zeroes.  Shift the 64-bit value in <i>m</i> right the amount specified by <i>count</i> while shifting in zeroes. For the best performance, <i>count</i> should be a constant.
PSRLDQ	<code>__m128i __mm_srli_si128(__m128i m, int imm)</code>	Shift 128 bit in <i>m</i> right by <i>imm</i> bytes while shifting in zeroes.
PSUBB	<code>__m128i __mm_sub_epi8(__m128i m1, __m128i m2)</code>	Subtract the 16 8-bit values in <i>m2</i> from the 16 8-bit values in <i>m1</i> .
PSUBB	<code>__m64 __mm_sub_pi8(__m64 m1, __m64 m2)</code>	Subtract the eight 8-bit values in <i>m2</i> from the eight 8-bit values in <i>m1</i> .

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
PSUBW	<code>__m128i _mm_sub_epi16(__m128i m1, __m128i m2)</code>	Subtract the 8 16-bit values in m2 from the 8 16-bit values in m1.
PSUBW	<code>__m64 _mm_sub_pi16(__m64 m1, __m64 m2)</code>	Subtract the four 16-bit values in m2 from the four 16-bit values in m1.
PSUBD	<code>__m128i _mm_sub_epi32(__m128i m1, __m128i m2)</code>	Subtract the 4 32-bit values in m2 from the 4 32-bit values in m1.
PSUBD	<code>__m64 _mm_sub_pi32(__m64 m1, __m64 m2)</code>	Subtract the two 32-bit values in m2 from the two 32-bit values in m1.
PSUBQ	<code>__m128i _mm_sub_epi64(__m128i m1, __m128i m2)</code>	Subtract the 2 64-bit values in m2 from the 2 64-bit values in m1.
PSUBQ	<code>__m64 _mm_sub_si64(__m64 m1, __m64 m2)</code>	Subtract the 64-bit values in m2 from the 64-bit values in m1.
PSUBSB	<code>__m128i _mm_subs_epi8(__m128i m1, __m128i m2)</code>	Subtract the 16 signed 8-bit values in m2 from the 16 signed 8-bit values in m1 and saturate.
PSUBSB	<code>__m64 _mm_subs_pi8(__m64 m1, __m64 m2)</code>	Subtract the eight signed 8-bit values in m2 from the eight signed 8-bit values in m1 and saturate.
PSUBSW	<code>__m128i _mm_subs_epi16(__m128i m1, __m128i m2)</code>	Subtract the 8 signed 16-bit values in m2 from the 8 signed 16-bit values in m1 and saturate.
PSUBSW	<code>__m64 _mm_subs_pi16(__m64 m1, __m64 m2)</code>	Subtract the four signed 16-bit values in m2 from the four signed 16-bit values in m1 and saturate.
PSUBUSB	<code>__m128i _mm_subs_epu8(__m128i m1, __m128i m2)</code>	Subtract the 16 unsigned 8-bit values in m2 from the 16 unsigned 8-bit values in m1 and saturate.
PSUBUSB	<code>__m64 _mm_subs_pu8(__m64 m1, __m64 m2)</code>	Subtract the eight unsigned 8-bit values in m2 from the eight unsigned 8-bit values in m1 and saturate.
PSUBUSW	<code>__m128i _mm_subs_epu16(__m128i m1, __m128i m2)</code>	Subtract the 8 unsigned 16-bit values in m2 from the 8 unsigned 16-bit values in m1 and saturate.



## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
PSUBUSW	<code>__m64 __mm_subs_pu16(__m64 m1, __m64 m2)</code>	Subtract the four unsigned 16-bit values in m2 from the four unsigned 16-bit values in m1 and saturate.
PUNPCKHBW	<code>__m64 __mm_unpackhi_pi8(__m64 m1, __m64 m2)</code>	Interleave the four 8-bit values from the high half of m1 with the four values from the high half of m2 and take the least significant element from m1.
PUNPCKHBW	<code>__m128i __mm_unpackhi_epi8(__m128i m1, __m128i m2)</code>	Interleave the 8 8-bit values from the high half of m1 with the 8 values from the high half of m2.
PUNPCKHWD	<code>__m64 __mm_unpackhi_pi16(__m64 m1, __m64 m2)</code>	Interleave the two 16-bit values from the high half of m1 with the two values from the high half of m2 and take the least significant element from m1.
PUNPCKHWD	<code>__m128i __mm_unpackhi_epi16(__m128i m1, __m128i m2)</code>	Interleave the 4 16-bit values from the high half of m1 with the 4 values from the high half of m2.
PUNPCKHDQ	<code>__m64 __mm_unpackhi_pi32(__m64 m1, __m64 m2)</code>	Interleave the 32-bit value from the high half of m1 with the 32-bit value from the high half of m2 and take the least significant element from m1.
PUNPCKHDQ	<code>__m128i __mm_unpackhi_epi32(__m128i m1, __m128i m2)</code>	Interleave two 32-bit value from the high half of m1 with the two 32-bit value from the high half of m2.
PUNPCKHQDQ	<code>__m128i __mm_unpackhi_epi64(__m128i m1, __m128i m2)</code>	Interleave the 64-bit value from the high half of m1 with the 64-bit value from the high half of m2.
PUNPCKLBW	<code>__m64 __mm_unpacklo_pi8 (__m64 m1, __m64 m2)</code>	Interleave the four 8-bit values from the low half of m1 with the four values from the low half of m2 and take the least significant element from m1.
PUNPCKLBW	<code>__m128i __mm_unpacklo_epi8 (__m128i m1, __m128i m2)</code>	Interleave the 8 8-bit values from the low half of m1 with the 8 values from the low half of m2.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

<b>Mnemonic</b>	<b>Intrinsic</b>	<b>Description</b>
PUNPCKLWD	<code>__m64 _mm_unpacklo_pi16(__m64 m1, __m64 m2)</code>	Interleave the two 16-bit values from the low half of m1 with the two values from the low half of m2 and take the least significant element from m1.
PUNPCKLWD	<code>__m128i _mm_unpacklo_epi16(__m128i m1, __m128i m2)</code>	Interleave the 4 16-bit values from the low half of m1 with the 4 values from the low half of m2.
PUNPCKLDQ	<code>__m64 _mm_unpacklo_pi32(__m64 m1, __m64 m2)</code>	Interleave the 32-bit value from the low half of m1 with the 32-bit value from the low half of m2 and take the least significant element from m1.
PUNPCKLDQ	<code>__m128i _mm_unpacklo_epi32(__m128i m1, __m128i m2)</code>	Interleave two 32-bit value from the low half of m1 with the two 32-bit value from the low half of m2.
PUNPCKLQDQ	<code>__m128i _mm_unpacklo_epi64(__m128i m1, __m128i m2)</code>	Interleave the 64-bit value from the low half of m1 with the 64-bit value from the low half of m2.
PXOR	<code>__m64 _mm_xor_si64(__m64 m1, __m64 m2)</code>	Perform a bitwise XOR of the 64-bit value in m1 with the 64-bit value in m2.
PXOR	<code>__m128i _mm_xor_si128(__m128i m1, __m128i m2)</code>	Perform a bitwise XOR of the 128-bit value in m1 with the 128-bit value in m2.
RCPPS	<code>__m128 _mm_rcp_ps(__m128 a)</code>	Computes the approximations of the reciprocals of the four SP FP values of a.
RCPSS	<code>__m128 _mm_rcp_ss(__m128 a)</code>	Computes the approximation of the reciprocal of the lower SP FP value of a; the upper three SP FP values are passed through.
RSQRTPS	<code>__m128 _mm_rsqrtps(__m128 a)</code>	Computes the approximations of the reciprocals of the square roots of the four SP FP values of a.
RSQRTSS	<code>__m128 _mm_rsqrtps(__m128 a)</code>	Computes the approximation of the reciprocal of the square root of the lower SP FP value of a; the upper three SP FP values are passed through.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
SFENCE	<code>void __mm_sfence(void)</code>	Guarantees that every preceding store is globally visible before any subsequent store.
SHUFDP	<code>__m128d __mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)</code>	Selects two specific DP FP values from a and b, based on the mask imm8. The mask must be an immediate.
SHUFPS	<code>__m128 __mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)</code>	Selects four specific SP FP values from a and b, based on the mask imm8. The mask must be an immediate.
SQRTPD	<code>__m128d __mm_sqrt_pd(__m128d a)</code>	Computes the square roots of the two DP FP values of a.
SQRTPS	<code>__m128 __mm_sqrt_ps(__m128 a)</code>	Computes the square roots of the four SP FP values of a.
SQRTSD	<code>__m128d __mm_sqrt_sd(__m128d a)</code>	Computes the square root of the lower DP FP value of a; the upper DP FP values are passed through.
SQRTSS	<code>__m128 __mm_sqrt_ss(__m128 a)</code>	Computes the square root of the lower SP FP value of a; the upper three SP FP values are passed through.
STMXCSR	<code>__mm_getcsr(void)</code>	Returns the contents of the control register.
SUBPD	<code>__m128d __mm_sub_pd(__m128d a, __m128d b)</code>	Subtracts the two DP FP values of a and b.
SUBPS	<code>__m128 __mm_sub_ps(__m128 a, __m128 b)</code>	Subtracts the four SP FP values of a and b.
SUBSD	<code>__m128d __mm_sub_sd(__m128d a, __m128d b)</code>	Subtracts the lower DP FP values of a and b. The upper DP FP values are passed through from a.
SUBSS	<code>__m128 __mm_sub_ss(__m128 a, __m128 b)</code>	Subtracts the lower SP FP values of a and b. The upper three SP FP values are passed through from a.
UCOMISD	<code>int __mm_ucomieq_sd(__m128d a, __m128d b)</code>	Compares the lower DP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
	int __mm_ucomilt_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.
	int __mm_ucomile_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.
	int __mm_ucomigt_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.
	int __mm_ucomige_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.
	int __mm_ucomineq_sd(__m128d a, __m128d b)	Compares the lower DP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.
UCOMISS	int __mm_ucomieq_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a equal to b. If a and b are equal, 1 is returned. Otherwise 0 is returned.
	int __mm_ucomilt_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a less than b. If a is less than b, 1 is returned. Otherwise 0 is returned.
	int __mm_ucomile_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a less than or equal to b. If a is less than or equal to b, 1 is returned. Otherwise 0 is returned.
	int __mm_ucomigt_ss(__m128 a, __m128 b)	Compares the lower SP FP value of a and b for a greater than b. If a is greater than b are equal, 1 is returned. Otherwise 0 is returned.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-1. Simple Intrinsics (Contd.)**

Mnemonic	Intrinsic	Description
	<code>int __mm_ucomige_ss(__m128 a, __m128 b)</code>  <code>int __mm_ucomineq_ss(__m128 a, __m128 b)</code>	<p>Compares the lower SP FP value of a and b for a greater than or equal to b. If a is greater than or equal to b, 1 is returned. Otherwise 0 is returned.</p> <p>Compares the lower SP FP value of a and b for a not equal to b. If a and b are not equal, 1 is returned. Otherwise 0 is returned.</p>
UNPCKHPD	<code>__m128d __mm_unpackhi_pd(__m128d a, __m128d b)</code>	Selects and interleaves the upper DP FP values from a and b.
UNPCKHPS	<code>__m128 __mm_unpackhi_ps(__m128 a, __m128 b)</code>	Selects and interleaves the upper two SP FP values from a and b.
UNPCKLPD	<code>__m128d __mm_unpacklo_pd(__m128d a, __m128d b)</code>	Selects and interleaves the lower DP FP values from a and b.
UNPCKLPS	<code>__m128 __mm_unpacklo_ps(__m128 a, __m128 b)</code>	Selects and interleaves the lower two SP FP values from a and b.
XORPD	<code>__m128d __mm_xor_pd(__m128d a, __m128d b)</code>	Computes bitwise EXOR (exclusive-or) of the two DP FP values of a and b.
XORPS	<code>__m128 __mm_xor_ps(__m128 a, __m128 b)</code>	Computes bitwise EXOR (exclusive-or) of the four SP FP values of a and b.

## C.2 COMPOSITE INTRINSICS

Table C-2. Composite Intrinsic

Mnemonic	Intrinsic	Description
(composite)	<code>__m128i _mm_set_epi64(__m64 q1, __m64 q0)</code>	Sets the two 64-bit values to the two inputs.
(composite)	<code>__m128i _mm_set_epi32(int i3, int i2, int i1, int i0)</code>	Sets the 4 32-bit values to the 4 inputs.
(composite)	<code>__m128i _mm_set_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w1,short w0)</code>	Sets the 8 16-bit values to the 8 inputs.
(composite)	<code>__m128i _mm_set_epi8(char w15,char w14, char w13, char w12, char w11, char w10, char w9,char w8,char w7,char w6,char w5, char w4, char w3, char w2,char w1,char w0)</code>	Sets the 16 8-bit values to the 16 inputs.
(composite)	<code>__m128i _mm_set1_epi64(__m64 q)</code>	Sets the 2 64-bit values to the input.
(composite)	<code>__m128i _mm_set1_epi32(int a)</code>	Sets the 4 32-bit values to the input.
(composite)	<code>__m128i _mm_set1_epi16(short a)</code>	Sets the 8 16-bit values to the input.
(composite)	<code>__m128i _mm_set1_epi8(char a)</code>	Sets the 16 8-bit values to the input.
(composite)	<code>__m128i _mm_setr_epi64(__m64 q1, __m64 q0)</code>	Sets the two 64-bit values to the two inputs in reverse order.
(composite)	<code>__m128i _mm_setr_epi32(int i3, int i2, int i1, int i0)</code>	Sets the 4 32-bit values to the 4 inputs in reverse order.
(composite)	<code>__m128i _mm_setr_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w, short w0)</code>	Sets the 8 16-bit values to the 8 inputs in reverse order.
(composite)	<code>__m128i _mm_setr_epi8(char w15,char w14, char w13, char w12, char w11, char w10, char w9,char w8,char w7,char w6,char w5, char w4, char w3, char w2,char w1,char w0)</code>	Sets the 16 8-bit values to the 16 inputs in reverse order.
(composite)	<code>__m128i _mm_setzero_si128()</code>	Sets all bits to 0.
(composite)	<code>__m128 _mm_set_ps(float w) __m128 _mm_set1_ps(float w)</code>	Sets the four SP FP values to w.
(composite)	<code>__m128cmm_set1_pd(double w)</code>	Sets the two DP FP values to w.
(composite)	<code>__m128d _mm_set_sd(double w)</code>	Sets the lower DP FP values to w.
(composite)	<code>__m128d _mm_set_pd(double z, double y)</code>	Sets the two DP FP values to the two inputs.
(composite)	<code>__m128 _mm_set_ps(float z, float y, float x, float w)</code>	Sets the four SP FP values to the four inputs.
(composite)	<code>__m128d _mm_setr_pd(double z, double y)</code>	Sets the two DP FP values to the two inputs in reverse order.

## INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

**Table C-2. Composite Intrinsic (Contd.)**

Mnemonic	Intrinsic	Description
(composite)	<code>__m128 __mm_setr_ps(float z, float y, float x, float w)</code>	Sets the four SP FP values to the four inputs in reverse order.
(composite)	<code>__m128d __mm_setzero_pd(void)</code>	Clears the two DP FP values.
(composite)	<code>__m128 __mm_setzero_ps(void)</code>	Clears the four SP FP values.
MOVSD + shuffle	<code>__m128d __mm_load_pd(double * p)</code> <code>__m128d __mm_load1_pd(double * p)</code>	Loads a single DP FP value, copying it into both DP FP values.
MOVSS + shuffle	<code>__m128 __mm_load_ps1(float * p)</code> <code>__m128 __mm_load1_ps(float * p)</code>	Loads a single SP FP value, copying it into all four words.
MOVAPD + shuffle	<code>__m128d __mm_loadr_pd(double * p)</code>	Loads two DP FP values in reverse order. The address must be 16-byte-aligned.
MOVAPS + shuffle	<code>__m128 __mm_loadr_ps(float * p)</code>	Loads four SP FP values in reverse order. The address must be 16-byte-aligned.
MOVSD + shuffle	<code>void __mm_store1_pd(double * p, __m128d a)</code>	Stores the lower DP FP value across both DP FP values.
MOVSS + shuffle	<code>void __mm_store_ps1(float * p, __m128 a)</code> <code>void __mm_store1_ps(float * p, __m128 a)</code>	Stores the lower SP FP value across four words.
MOVAPD + shuffle	<code>__mm_storer_pd(double * p, __m128d a)</code>	Stores two DP FP values in reverse order. The address must be 16-byte-aligned.
MOVAPS + shuffle	<code>__mm_storer_ps(float * p, __m128 a)</code>	Stores four SP FP values in reverse order. The address must be 16-byte-aligned.

**INTEL C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS**



# **Index**



# INDEX FOR VOLUME 2A & 2B

## Numerics

- 64-bit mode
  - control and debug registers, 2-15
  - default operand size, 2-15
  - direct memory-offset MOVs, 2-14
  - general purpose encodings, B-22
  - immediates, 2-14
  - introduction, 2-9
  - machine instructions, B-1
  - reg (reg) field, B-4
  - REX prefixes, 2-9, B-2
  - RIP-relative addressing, 2-14
  - SIMD encodings, B-47
  - special instruction encodings, B-77
  - summary table notation, 3-6

## A

- AAA instruction, 3-18
- AAD instruction, 3-20
- AAM instruction, 3-22
- AAS instruction, 3-24
- Access rights, segment descriptor, 3-509
- ADC instruction, 3-26, 3-538
- ADD instruction, 3-18, 3-29, 3-251, 3-538
- ADDPD instruction, 3-32
- ADDPS instruction, 3-35
- Addressing methods
  - RIP-relative, 2-14
- Addressing, segments, 1-4
- ADDSD instruction, 3-38
- ADDSS instruction, 3-41
- ADDSUBPD instruction, 3-44
- ADDSUBPS instruction, 3-48
- AND instruction, 3-52, 3-538
- ANDNPD instruction, 3-59
- ANDNPS instruction, 3-61
- ANDPD instruction, 3-55
- ANDPS instruction, 3-57
- Arctangent, x87 FPU operation, 3-348
- ARPL instruction, 3-63

## B

- B (default stack size) flag, segment descriptor, 4-180
- Base (operand addressing), 2-4
- BCD integers
  - packed, 3-251, 3-253, 3-286, 3-288
  - unpacked, 3-18, 3-20, 3-22, 3-24
- Binary numbers, 1-4
- Bit order, 1-2
- BOUND instruction, 3-65
- BOUND range exceeded exception (#BR), 3-65

- Branch hints, 2-2
- Brand information, 3-174
  - processor brand index, 3-178
  - processor brand string, 3-175
- BSF instruction, 3-67
- BSR instruction, 3-69
- BSWAP instruction, 3-71
- BT instruction, 3-73
- BTC instruction, 3-76, 3-538
- BTR instruction, 3-79, 3-538
- BTS instruction, 3-82, 3-538
- Byte order, 1-2

## C

- Cache and TLB information, 3-171
- Caches, invalidating (flushing), 3-476, 4-359
- CALL instruction, 3-85
- CBW instruction, 3-102
- CDQ instruction, 3-249
- CDQE instruction, 3-102
- CF (carry) flag, EFLAGS register, 3-29, 3-73, 3-76, 3-79, 3-82, 3-103, 3-112, 3-255, 3-450, 3-456, 3-682, 4-193, 4-244, 4-259, 4-262, 4-290, 4-304
- CLC instruction, 3-103
- CLD instruction, 3-104
- CLFLUSH instruction, 3-105
  - CPUID flag, 3-170
- CLI instruction, 3-107
- CLTS instruction, 3-110
- CMC instruction, 3-112
- CMOVcc flag, 3-170
- CMOVcc instructions, 3-113
  - CPUID flag, 3-170
- CMP instruction, 3-119
- CMPPD instruction, 3-122
- CMPPS instruction, 3-127
- CMPS instruction, 3-132, 4-213
- CMPSPB instruction, 3-132
- CMPSPD instruction, 3-132, 3-137
- CMPSPQ instruction, 3-132
- CMPSS instruction, 3-141
- CMPSPW instruction, 3-132
- CMPXCHG instruction, 3-145, 3-538
- CMPXCHG16B instruction, 3-148
  - CPUID bit, 3-167
- CMPXCHG8B instruction, 3-148
  - CPUID flag, 3-169
- COMISD instruction, 3-151
- COMISS instruction, 3-154
- Compatibility mode
  - introduction, 2-9
  - see 64-bit mode

## INDEX

- summary table notation, 3-7
  - Compatibility, software, 1-3
  - Condition code flags, EFLAGS register, 3-113
  - Condition code flags, x87 FPU status word
    - flags affected by instructions, 3-14
    - setting, 3-399, 3-401, 3-404
  - Conditional jump, 3-491
  - Conforming code segment, 3-509
  - Constants (floating point), loading, 3-336
  - Control registers, moving values to and from, 3-593
  - Cosine, x87 FPU operation, 3-304, 3-374
  - CPL, 3-107, 4-354
  - CPUID instruction, 3-157, 3-170
    - 36-bit page size extension, 3-170
    - APIC on-chip, 3-169
    - basic CPUID information, 3-158
    - cache and TLB characteristics, 3-158, 3-171
    - CLFLUSH flag, 3-170
    - CLFLUSH instruction cache line size, 3-165
    - CMPXCHG16B flag, 3-167
    - CMPXCHG8B flag, 3-169
    - CPL qualified debug store, 3-166
    - debug extensions, CR4.DE, 3-169
    - debug store supported, 3-170
    - deterministic cache parameters leaf, 3-159
    - extended function information, 3-161
    - feature information, 3-168
    - FPU on-chip, 3-169
    - FSAVE flag, 3-170
    - FXRSTOR flag, 3-170
    - HT technology flag, 3-170
    - IA-32e mode available, 3-161
    - input limits for EAX, 3-162
    - L1 Context ID, 3-166
    - local APIC physical ID, 3-165
    - machine check architecture, 3-169
    - machine check exception, 3-169
    - memory type range registers, 3-169
    - microcode update signature, 3-163
    - MONITOR feature information, 3-174
    - MONITOR/MWAIT flag, 3-166
    - MONITOR/MWAIT leaf, 3-159, 3-160
    - MWAIT feature information, 3-174
    - page attribute table, 3-170
    - page size extension, 3-169
    - performance monitoring features, 3-174
    - physical address bits, 3-162
    - physical address extension, 3-169
    - power management, 3-174
    - processor brand index, 3-165, 3-174
    - processor brand string, 3-161, 3-174
    - processor serial number, 3-158, 3-170
    - processor type field, 3-164
    - PTE global bit, 3-169
    - RDMSR flag, 3-169
    - returned in EBX, 3-165
    - returned in ECX & EDX, 3-165
    - self snoop, 3-170
    - SpeedStep technology, 3-166
    - SS2 extensions flag, 3-170
    - SSE extensions flag, 3-170
    - SSE3 extensions flag, 3-166
    - SYSENTER flag, 3-169
    - SYSEXIT flag, 3-169
    - thermal management, 3-174
    - thermal monitor, 3-166, 3-170
    - time stamp counter, 3-169
    - using CPUID, 3-157
    - vendor ID string, 3-162
    - version information, 3-158, 3-163, 3-174
    - virtual 8086 Mode flag, 3-169
    - virtual address bits, 3-162
    - WRMSR flag, 3-169
  - CQO instruction, 3-249
  - CR0 control register, 4-276
  - CS register, 3-86, 3-462, 3-480, 3-498, 3-588, 4-104
  - CVTDQ2PD instruction, 3-184
  - CVTDQ2PS instruction, 3-186
  - CVTPD2DQ instruction, 3-189
  - CVTPD2PI instruction, 3-192
  - CVTPD2PS instruction, 3-195
  - CVTPI2PD instruction, 3-198
  - CVTPI2PS instruction, 3-201
  - CVTPS2DQ instruction, 3-204
  - CVTPS2PD instruction, 3-207
  - CVTPS2PI instruction, 3-210
  - CVTSD2SI instruction, 3-213
  - CVTSD2SS instruction, 3-216
  - CVTSI2SD instruction, 3-219
  - CVTSI2SS instruction, 3-222
  - CVTSS2SD instruction, 3-225
  - CVTSS2SI instruction, 3-228
  - CVTTPD2DQ instruction, 3-234
  - CVTTPD2PI instruction, 3-231
  - CVTTPS2DQ instruction, 3-237
  - CVTTPS2PI instruction, 3-240
  - CVTTSD2SI instruction, 3-243
  - CVTTSS2SI instruction, 3-246
  - CWD instruction, 3-249
  - CWDE instruction, 3-102
  - C/C++ compiler intrinsics
    - compiler functional equivalents, C-1
    - composite, C-32
    - description of, 3-11
    - lists of, C-1
    - simple, C-3
- ## D
- D (default operation size) flag, segment descriptor, 4-104, 4-109, 4-180
  - DAA instruction, 3-251
  - DAS instruction, 3-253
  - Debug registers, moving value to and from, 3-596
  - DEC instruction, 3-255, 3-538

Denormalized finite number, 3-404  
 DF (direction) flag, EFLAGS register, 3-104, 3-133,  
 3-459, 3-541, 3-657, 4-18, 4-247, 4-291  
 Displacement (operand addressing), 2-4  
 DIV instruction, 3-257  
 Divide error exception (#DE), 3-257  
 DIVPD instruction, 3-261  
 DIVPS instruction, 3-264  
 DIVSD instruction, 3-267  
 DIVSS instruction, 3-270  
 DS register, 3-133, 3-519, 3-540, 3-656, 4-17

## E

EDI register, 4-246, 4-291, 4-297  
 Effective address, 3-525  
 EFLAGS register  
   condition codes, 3-116, 3-295, 3-301  
   flags affected by instructions, 3-14  
   loading, 3-507  
   popping, 4-111  
   popping on return from interrupt, 3-480  
   pushing, 4-185  
   pushing on interrupts, 3-462  
   saving, 4-235  
   status flags, 3-119, 3-494, 4-253, 4-332  
 EIP register, 3-86, 3-462, 3-480, 3-498  
 EMMS instruction, 3-273  
 Encodings  
   See machine instructions, opcodes  
 ENTER instruction, 3-275  
 ES register, 3-519, 4-17, 4-246, 4-297  
 ESI register, 3-133, 3-540, 3-541, 3-656, 4-17,  
 4-291  
 ESP register, 3-86, 4-104  
 Exceptions  
   BOUND range exceeded (#BR), 3-65  
   notation, 1-5  
   overflow exception (#OF), 3-462  
   returning from, 3-480  
 Exponent, extracting from floating-point number,  
 3-420  
 Extract exponent and significand, x87 FPU  
 operation, 3-420

## F

F2XM1 instruction, 3-278, 3-420  
 FABS instruction, 3-280  
 FADD instruction, 3-282  
 FADDP instruction, 3-282  
 Far call, CALL instruction, 3-86  
 Far pointer, loading, 3-519  
 Far return, RET instruction, 4-216  
 FBLD instruction, 3-286  
 FBSTP instruction, 3-288  
 FCHS instruction, 3-291  
 FCLEX instruction, 3-293

FCMOVcc instructions, 3-295  
 FCOM instruction, 3-297  
 FCOMI instruction, 3-301  
 FCOMIP instruction, 3-301  
 FCOMP instruction, 3-297  
 FCOMPP instruction, 3-297  
 FCOS instruction, 3-304  
 FDECSTP instruction, 3-306  
 FDIV instruction, 3-308  
 FDIVP instruction, 3-308  
 FDIVR instruction, 3-312  
 FDIVRP instruction, 3-312  
 Feature information, processor, 3-157  
 FFREE instruction, 3-316  
 FIADD instruction, 3-282  
 FICOM instruction, 3-317  
 FICOMP instruction, 3-317  
 FIDIV instruction, 3-308  
 FIDIVR instruction, 3-312  
 FILD instruction, 3-320  
 FIMUL instruction, 3-343  
 FINCSTP instruction, 3-322  
 FINIT instruction, 3-324  
 FINIT/FNINIT instructions, 3-365  
 FIST instruction, 3-326  
 FISTP instruction, 3-326  
 FISTTP instruction, 3-330  
 FISUB instruction, 3-391  
 FISUBR instruction, 3-395  
 FLD instruction, 3-333  
 FLD1 instruction, 3-336  
 FLDCW instruction, 3-338  
 FLDENV instruction, 3-340  
 FLDL2E instruction, 3-336  
 FLDL2T instruction, 3-336  
 FLDLG2 instruction, 3-336  
 FLDLN2 instruction, 3-336  
 FLDPI instruction, 3-336  
 FLDZ instruction, 3-336  
 Floating point instructions  
   machine encodings, B-80  
 Floating-point exceptions  
   SSE and SSE2 SIMD, 3-16  
   x87 FPU, 3-16  
 Flushing  
   caches, 3-476, 4-359  
   TLB entry, 3-478  
 FMUL instruction, 3-343  
 FMULP instruction, 3-343  
 FNCLEX instruction, 3-293  
 FNINIT instruction, 3-324  
 FNOP instruction, 3-347  
 FNSAVE instruction, 3-365  
 FNSTCW instruction, 3-382  
 FNSTENV instruction, 3-340, 3-385  
 FNSTSW instruction, 3-388  
 FPATAN instruction, 3-348  
 FPREM instruction, 3-351

## INDEX

FPREM1 instruction, 3-354  
FPTAN instruction, 3-357  
FRNDINT instruction, 3-360  
FRSTOR instruction, 3-362  
FS register, 3-519  
FSAVE instruction, 3-365  
FSAVE/FNSAVE instructions, 3-362  
FSCALE instruction, 3-369  
FSIN instruction, 3-372  
FSINCOS instruction, 3-374  
FSQRT instruction, 3-377  
FST instruction, 3-379  
FSTCW instruction, 3-382  
FSTENV instruction, 3-385  
FSTP instruction, 3-379  
FSTSW instruction, 3-388  
FSUB instruction, 3-391  
FSUBP instruction, 3-391  
FSUBR instruction, 3-395  
FSUBRP instruction, 3-395  
FTST instruction, 3-399  
FUCOM instruction, 3-401  
FUCOMI instruction, 3-301  
FUCOMIP instruction, 3-301  
FUCOMP instruction, 3-401  
FUCOMPP instruction, 3-401  
FXAM instruction, 3-404  
FXCH instruction, 3-406  
FXRSTOR instruction, 3-408  
    CPUID flag, 3-170  
FXSAVE instruction, 3-411  
    CPUID flag, 3-170  
EXTRACT instruction, 3-369, 3-420  
FYL2X instruction, 3-422  
FYL2XP1 instruction, 3-424

## G

GDT (global descriptor table), 3-531, 3-534  
GDTR (global descriptor table register), 3-531, 4-256  
General-purpose instructions  
    64-bit encodings, B-22  
    non-64-bit encodings, B-9  
General-purpose registers  
    moving value to and from, 3-588  
    popping all, 4-109  
    pushing all, 4-183  
GS register, 3-519

## H

HADDPD instruction, 3-427, 3-428  
HADDPS instruction, 3-431  
Hexadecimal numbers, 1-4  
HLT instruction, 3-435  
HSUBPD instruction, 3-437  
HSUBPS instruction, 3-441

Hyper-Threading Technology  
    CPUID flag, 3-170

## I

IA-32e mode  
    CPUID flag, 3-161  
    introduction, 2-9  
    see 64-bit mode  
    see compatibility mode  
IA32\_SYSENTER\_CS MSR, 4-325, 4-328  
IA32\_SYSENTER\_EIP MSR, 4-325  
IA32\_SYSENTER\_ESP MSR, 4-325  
IDIV instruction, 3-445  
IDT (interrupt descriptor table), 3-463, 3-531  
IDTR (interrupt descriptor table register), 3-531, 4-271  
IF (interrupt enable) flag, EFLAGS register, 3-107, 4-292  
Immediate operands, 2-4  
IMUL instruction, 3-449  
IN instruction, 3-454  
INC instruction, 3-456, 3-538  
Index (operand addressing), 2-4  
Initialization x87 FPU, 3-324  
INS instruction, 3-458, 4-213  
INSB instruction, 3-458  
INSD instruction, 3-458  
Instruction format  
    base field, 2-4  
    description of reference information, 3-1  
    displacement, 2-4  
    illustration of, 2-1  
    immediate, 2-4  
    index field, 2-4  
    Mod field, 2-4  
    ModR/M byte, 2-4  
    opcode, 2-3  
    operands, 1-4  
    prefixes, 2-2  
    reg/opcode field, 2-4  
    r/m field, 2-4  
    scale field, 2-4  
    SIB byte, 2-4  
    See also: machine instructions, opcodes  
Instruction reference, nomenclature, 3-1  
Instruction set, reference, 3-1  
INSW instruction, 3-458  
INT 3 instruction, 3-462  
Integer, storing, x87 FPU data type, 3-326  
Intel NetBurst microarchitecture, 1-1  
Intel Xeon processor, 1-1  
Inter-privilege level  
    call, CALL instruction, 3-86  
    return, RET instruction, 4-216  
Interrupts  
    interrupt vector 4, 3-462  
    returning from, 3-480

software, 3-462  
 INTn instruction, 3-462  
 INTO instruction, 3-462  
 Intrinsics  
   compiler functional equivalents, C-1  
   composite, C-32  
   description of, 3-11  
   list of, C-1  
   simple, C-3  
 INVD instruction, 3-476  
 INVLPG instruction, 3-478  
 IOPL (I/O privilege level) field, EFLAGS register,  
   3-107, 4-185, 4-292  
 IRET instruction, 3-480  
 IRETD instruction, 3-480

**J**

Jcc instructions, 3-491  
 JMP instruction, 3-497  
 Jump operation, 3-497

**L**

L1 Context ID, 3-166  
 LAHF instruction, 3-507  
 LAR instruction, 3-509  
 LDDQU instruction, 3-513  
 LDMXCSR instruction, 3-516  
 LDS instruction, 3-519  
 LDT (local descriptor table), 3-534  
 LDTR (local descriptor table register), 3-534, 4-274  
 LEA instruction, 3-525  
 LEAVE instruction, 3-528  
 LES instruction, 3-519  
 LFENCE instruction, 3-530  
 LFS instruction, 3-519  
 LGDT instruction, 3-531  
 LGS instruction, 3-519  
 LIDT instruction, 3-531  
 LLDT instruction, 3-534  
 LMSW instruction, 3-536  
 Load effective address operation, 3-525  
 LOCK prefix, 3-27, 3-30, 3-52, 3-76, 3-79, 3-82,  
   3-145, 3-255, 3-456, 3-538, 4-2, 4-6, 4-9,  
   4-244, 4-304, 4-363, 4-365, 4-371  
 Locking operation, 3-538  
 LODS instruction, 3-540, 4-213  
 LODSB instruction, 3-540  
 LODSD instruction, 3-540  
 LODSQ instruction, 3-540  
 LODSW instruction, 3-540  
 Log epsilon, x87 FPU operation, 3-422  
 Log (base 2), x87 FPU operation, 3-424  
 LOOP instructions, 3-543  
 LOOPcc instructions, 3-543  
 LSL instruction, 3-546  
 LSS instruction, 3-519

LTR instruction, 3-550

## M

Machine check architecture  
   CPUID flag, 3-169  
   description, 3-169  
 Machine instructions  
   64-bit mode, B-1  
   condition test (tttn) field, B-7  
   direction bit (d) field, B-8  
   floating-point instruction encodings, B-80  
   general description, B-1  
   general-purpose encodings, B-9–B-46  
   legacy prefixes, B-2  
   MMX encodings, B-48–B-52  
   opcode fields, B-2  
   operand size (w) bit, B-5  
   P6 family encodings, B-52  
   Pentium processor family encodings, B-47  
   reg (reg) field, B-3, B-4  
   REX prefixes, B-2  
   segment register (sreg) field, B-6  
   sign-extend (s) bit, B-5  
   SIMD 64-bit encodings, B-47  
   special 64-bit encodings, B-77  
   special fields, B-2  
   special-purpose register (eee) field, B-7  
   SSE encodings, B-53–B-60  
   SSE2 encodings, B-61–B-74  
   SSE3 encodings, B-75–B-76  
   See also: opcodes  
 Machine status word, CR0 register, 3-536, 4-276  
 MASKMOVDQU instruction, 3-553  
 MASKMOVQ instruction, 3-556  
 MAXPD instruction, 3-559  
 MAXPS instruction, 3-562  
 MAXSD instruction, 3-565  
 MAXSS instruction, 3-568  
 MFENCE instruction, 3-571  
 Microcode update signature, 3-163  
 MINPD instruction, 3-572  
 MINPS instruction, 3-575  
 MINS instruction, 3-578  
 MINSS instruction, 3-581  
 MMX instructions  
   CPUID flag for technology, 3-170  
   encodings, B-48  
 Mod field, instruction format, 2-4  
 Model & family information, 3-163, 3-174  
 ModR/M byte, 2-4  
   16-bit addressing forms, 2-6  
   32-bit addressing forms of, 2-7  
   description of, 2-4  
   format of, 2-1  
 MONITOR instruction, 3-584  
   CPUID flag, 3-166  
   feature data, 3-174

## INDEX

MOV instruction, 3-587  
MOV instruction (control registers), 3-593  
MOV instruction (debug registers), 3-596  
MOVAPD instruction, 3-598  
MOVAPS instruction, 3-601  
MOVD instruction, 3-604  
MOVDDUP instruction, 3-608  
MOVDQ2Q instruction, 3-615  
MOVDQA instruction, 3-611  
MOVDQU instruction, 3-613  
MOVHLPS instruction, 3-617  
MOVHPD instruction, 3-619  
MOVHPS instruction, 3-622  
MOVLHP instruction, 3-625  
MOVLHPS instruction, 3-625  
MOVLPD instruction, 3-627  
MOVLPS instruction, 3-630  
MOVMSKPD instruction, 3-633  
MOVMSKPS instruction, 3-635  
MOVNTDQ instruction, 3-637  
MOVNTI instruction, 3-640  
MOVNTPD instruction, 3-642  
MOVNTPS instruction, 3-645  
MOVNTQ instruction, 3-648  
MOVQ instruction, 3-604, 3-651  
MOVQ2DQ instruction, 3-654  
MOVS instruction, 3-656, 4-213  
MOVSB instruction, 3-656  
MOVSD instruction, 3-656, 3-660  
MOVSHDUP instruction, 3-663  
MOVSLDUP instruction, 3-666  
MOVSQ instruction, 3-656  
MOVSS instruction, 3-669  
MOVSW instruction, 3-656  
MOVSX instruction, 3-672  
MOVSDX instruction, 3-672  
MOVUPD instruction, 3-674  
MOVUPS instruction, 3-677  
MOVZX instruction, 3-680  
MSRs (model specific registers)  
  reading, 4-203  
  writing, 4-361  
MUL instruction, 3-22, 3-682  
MULPD instruction, 3-685  
MULPS instruction, 3-688  
MULSD instruction, 3-691  
MULSS instruction, 3-694  
Multi-byte no operation, 4-4, B-15  
MWAIT instruction, 3-697  
  CPUID flag, 3-166  
  feature data, 3-174

## N

NaN. testing for, 3-399  
Near  
  call, CALL instruction, 3-85  
  return, RET instruction, 4-216

NEG instruction, 3-538, 4-2  
NetBurst microarchitecture (see Intel NetBurst microarchitecture)  
No operation, 4-4, B-15  
Nomenclature, used in instruction reference pages, 3-1  
NOP instruction, 4-4  
NOT instruction, 3-538, 4-6  
Notation  
  bit and byte order, 1-2  
  exceptions, 1-5  
  hexadecimal and binary numbers, 1-4  
  instruction operands, 1-4  
  reserved bits, 1-3  
  segmented addressing, 1-4  
Notational conventions, 1-2  
NT (nested task) flag, EFLAGS register, 3-480

## O

OF (carry) flag, EFLAGS register, 3-450  
OF (overflow) flag, EFLAGS register, 3-29, 3-462, 3-682, 4-244, 4-259, 4-262, 4-304  
Opcode format, 2-3  
Opcodes  
  addressing method codes for, A-2  
  extensions, A-19  
  extensions tables, A-20  
  group numbers, A-19  
  integers  
    one-byte opcodes, A-9  
    two-byte opcodes, A-11  
  key to abbreviations, A-2  
  look-up examples, A-4, A-19, A-22  
  ModR/M byte, A-19  
  one-byte opcodes, A-4, A-9  
  opcode maps, A-1  
  operand type codes for, A-3  
  register codes for, A-4  
  superscripts in tables, A-7  
  two-byte opcodes, A-5, A-6, A-11  
  VMX instructions, B-86  
  x87 ESC instruction opcodes, A-22  
Operands, 1-4  
OR instruction, 3-538, 4-8  
ORPD instruction, 4-11  
ORPS instruction, 4-13  
OUT instruction, 4-15  
OUTS instruction, 4-17, 4-213  
OUTSB instruction, 4-17  
OUTSD instruction, 4-17  
OUTSW instruction, 4-17  
Overflow exception (#OF), 3-462

## P

P6 family processors  
  description of, 1-1



- machine encodings, B-52
- PACKSSDW instruction, 4-22
- PACKSSWB instruction, 4-22
- PACKUSWB instruction, 4-26
- PADDB instruction, 4-30
- PADDD instruction, 4-30
- PADDQ instruction, 4-34
- PADDSB instruction, 4-37
- PADDSW instruction, 4-37
- PADDUSB instruction, 4-41
- PADDUSW instruction, 4-41
- PADDW instruction, 4-30
- PAND instruction, 4-45
- PANDN instruction, 4-48
- PAUSE instruction, 4-51
- PAVGB instruction, 4-52
- PAVGW instruction, 4-52
- PCE flag, CR4 register, 4-206
- PCMPEQB instruction, 4-55
- PCMPEQD instruction, 4-55
- PCMPEQW instruction, 4-55
- PCMPGTB instruction, 4-59
- PCMPGTD instruction, 4-59
- PCMPGTW instruction, 4-59
- PE (protection enable) flag, CR0 register, 3-536
- Pending break enable, 3-170
- Pentium 4 processor, 1-1
- Pentium II processor, 1-1
- Pentium III processor, 1-1
- Pentium M processor, 1-1
- Pentium Pro processor, 1-1
- Pentium processor, 1-1
- Pentium processor family processors
  - machine encodings, B-47
- Performance-monitoring counters
  - CPUID inquiry for, 3-174
  - reading, 4-205
- PEXTRW instruction, 4-64
- Pi, 3-336
- PINSRW instruction, 4-67
- PMADDWD instruction, 4-70
- PMAXSW instruction, 4-74
- PMAXUB instruction, 4-77
- PMINSW instruction, 4-80
- PMINUB instruction, 4-83
- PMOVMSKB instruction, 4-86
- PMULHUW instruction, 4-88
- PMULHW instruction, 4-92
- PMULLW instruction, 4-96
- PMULUDQ instruction, 4-100
- POP instruction, 4-103
- POPA instruction, 4-109
- POPAD instruction, 4-109
- POPF instruction, 4-111
- POPFD instruction, 4-111
- POPFQ instruction, 4-111
- POR instruction, 4-115
- PREFETCHh instruction, 4-118
- Prefixes
  - Address-size override prefix, 2-2
  - Branch hints, 2-2
  - branch hints, 2-2
  - instruction, description of, 2-2
  - legacy prefix encodings, B-2
  - LOCK, 2-2, 3-538
  - Operand-size override prefix, 2-2
  - REP or REPE/REPZ, 2-2
  - REPNE/REPNZ, 2-2
  - REP/REPE/REPZ/REPNE/REPZ, 4-211
  - REX prefix encodings, B-2
  - Segment override prefixes, 2-2
- PSADBW instruction, 4-120
- Pseudo-functions
  - VMfail, 5-2
  - VMfailInvalid, 5-2
  - VMfailValid, 5-2
  - VMsucceed, 5-2
- PSHUFD instruction, 4-124
- PSHUFHW instruction, 4-127
- PSHUFLW instruction, 4-130
- PSHUFW instruction, 4-133
- PSLLD instruction, 4-138
- PSLLDQ instruction, 4-136
- PSLLQ instruction, 4-138
- PSLLW instruction, 4-138
- PSRAD instruction, 4-143
- PSRAW instruction, 4-143
- PSRLD instruction, 4-149
- PSRLDQ instruction, 4-147
- PSRLQ instruction, 4-149
- PSRLW instruction, 4-149
- PSUBB instruction, 4-154
- PSUBD instruction, 4-154
- PSUBQ instruction, 4-158
- PSUBSB instruction, 4-161
- PSUBSW instruction, 4-161
- PSUBUSB instruction, 4-165
- PSUBUSW instruction, 4-165
- PSUBW instruction, 4-154
- PUNPCKHBW instruction, 4-169
- PUNPCKHDQ instruction, 4-169
- PUNPCKHQDQ instruction, 4-169
- PUNPCKHWD instruction, 4-169
- PUNPCKLBW instruction, 4-174
- PUNPCKLDQ instruction, 4-174
- PUNPCKLQDQ instruction, 4-174
- PUNPCKLWD instruction, 4-174
- PUSH instruction, 4-179
- PUSHA instruction, 4-183
- PUSHAD instruction, 4-183
- PUSHF instruction, 4-185
- PUSHFD instruction, 4-185
- PXOR instruction, 4-188

## INDEX

### R

RC (rounding control) field, x87 FPU control word, 3-327, 3-336, 3-379

RCL instruction, 4-191

RCPSS instruction, 4-197

RCPSS instruction, 4-200

RCR instruction, 4-191

RDMSR instruction, 4-203, 4-206, 4-209

  CPUID flag, 3-169

RDPMS instruction, 4-205

RDTSC instruction, 4-209

Reg/opcode field, instruction format, 2-4

Related literature, 1-7

Remainder, x87 FPU operation, 3-354

REP/REPE/REPZ/REPNE/REPNZ prefixes, 3-133, 3-459, 4-18, 4-211

Reserved

  use of reserved bits, 1-3

RET instruction, 4-216

REX prefixes

  addressing modes, 2-11

  and INC/DEC, 2-10

  encodings, 2-10, B-2

  field names, 2-11

  ModR/M byte, 2-10

  overview, 2-9

  REX.B, 2-10

  REX.R, 2-10

  REX.W, 2-10

  special encodings, 2-13

RIP-relative addressing, 2-14

ROL instruction, 4-191

ROR instruction, 4-191

Rounding, round to integer, x87 FPU operation, 3-360

RPL field, 3-63

RSM instruction, 4-227

RSQRTPS instruction, 4-229

RSQRTSS instruction, 4-232

R/m field, instruction format, 2-4

### S

SAL instruction, 4-237

SAR instruction, 4-237

SBB instruction, 3-538, 4-243

Scale (operand addressing), 2-4

Scale, x87 FPU operation, 3-369

Scan string instructions, 4-246

SCAS instruction, 4-213, 4-246

SCASB instruction, 4-246

SCASD instruction, 4-246

SCASW instruction, 4-246

Segment

  descriptor, segment limit, 3-546

  limit, 3-546

  registers, moving values to and from, 3-588

  selector, RPL field, 3-63

Segmented addressing, 1-4

Self Snoop, 3-170

SETcc instructions, 4-251

SF (sign) flag, EFLAGS register, 3-29

SFENCE instruction, 4-255

SGDT instruction, 4-256

SHAF instruction, 4-235

Shift instructions, 4-237

SHL instruction, 4-237

SHLD instruction, 4-259

SHR instruction, 4-237

SHRD instruction, 4-262

SHUFPS instruction, 4-265

SHUFPD instruction, 4-268

SIB byte, 2-4

  32-bit addressing forms of, 2-8

  description of, 2-4

  format of, 2-1

SIDT instruction, 4-256, 4-271

Significand, extracting from floating-point number, 3-420

SIMD floating-point exceptions, unmasking, effects of, 3-516

Sine, x87 FPU operation, 3-372, 3-374

SLDT instruction, 4-274

SMSW instruction, 4-276

SpeedStep technology, 3-166

SQRTPD instruction, 4-278

SQRTPS instruction, 4-281

SQRTSD instruction, 4-284

SQRTSS instruction, 4-287

Square root, Fx87 PU operation, 3-377

SS register, 3-519, 3-588, 4-104

SS3 extensions

  event mgmt instruction encodings, B-76

  floating-point instruction encodings, B-75

SSE extensions

  cacheability instruction encodings, B-60

  CPUID flag, 3-170

  floating-point encodings, B-53

  instruction encodings, B-53

  integer instruction encodings, B-59

  memory ordering encodings, B-60

SSE2 extensions

  cacheability instruction encodings, B-74

  CPUID flag, 3-170

  floating-point encodings, B-61

  integer instruction encodings, B-68

SSE3 extensions

  CPUID extended function information, 3-163

  CPUID flag, 3-166

  integer instruction encodings, B-76

Stack, pushing values on, 4-180

Status flags, EFLAGS register, 3-116, 3-119, 3-295, 3-301, 3-494, 4-253, 4-332

STC instruction, 4-290

STD instruction, 4-291

Stepping information, 3-163, 3-174

STI instruction, 4-292  
 STMXCSR instruction, 4-295  
 STOS instruction, 4-213, 4-297  
 STOSB instruction, 4-297  
 STOSD instruction, 4-297  
 STOSQ instruction, 4-297  
 STOSW instruction, 4-297  
 STR instruction, 4-301  
 String instructions, 3-132, 3-458, 3-540, 3-656,  
 4-17, 4-246, 4-297  
 SUB instruction, 3-24, 3-253, 3-538, 4-303  
 SUBPD instruction, 4-306  
 SUBSS instruction, 4-315  
 Summary table notation, 3-7  
 SWAPGS instruction, 4-318  
 SYSCALL instruction, 4-320  
 SYSENTER instruction, 4-322  
 CPUID flag, 3-169  
 SYSEXIT instruction, 4-326  
 CPUID flag, 3-169  
 SYSRET instruction, 4-330

**T**

Tangent, x87 FPU operation, 3-357  
 Task register  
 loading, 3-550  
 storing, 4-301  
 Task switch  
 CALL instruction, 3-86  
 return from nested task, IRET instruction, 3-480  
 TEST instruction, 4-332  
 Thermal Monitor  
 CPUID flag, 3-170  
 Thermal Monitor 2, 3-166  
 CPUID flag, 3-166  
 Time Stamp Counter, 3-169  
 Time-stamp counter, reading, 4-209  
 TLB entry, invalidating (flushing), 3-478  
 TS (task switched) flag, CR0 register, 3-110  
 TSD flag, CR4 register, 4-209  
 TSS, relationship to task register, 4-301

**U**

UCOMISD instruction, 4-335  
 UCOMISS instruction, 4-338  
 UD2 instruction, 4-341  
 Undefined, format opcodes, 3-399  
 Unordered values, 3-297, 3-399, 3-401  
 UNPCKHPD instruction, 4-342  
 UNPCKHPS instruction, 4-345  
 UNPCKLPD instruction, 4-348  
 UNPCKLPS instruction, 4-351

**V**

VERR instruction, 4-354  
 Version information, processor, 3-157

VERW instruction, 4-354  
 VM (virtual 8086 mode) flag, EFLAGS register,  
 3-480  
 VMCALL instruction, 5-1  
 VMCLEAR instruction, 5-1  
 VMLAUNCH instruction, 5-1  
 VMPTRLD instruction, 5-1  
 VMPTRST instruction, 5-1  
 VMREAD instruction, 5-1  
 VMRESUME instruction, 5-1  
 VMWRITE instruction, 5-1  
 VMXOFF instruction, 5-1  
 VMXON instruction, 5-2

**W**

WAIT/FWAIT instructions, 4-357  
 WBINVD instruction, 4-359  
 Write-back and invalidate caches, 4-359  
 WRMSR instruction, 4-361  
 CPUID flag, 3-169

**X**

x87 FPU  
 checking for pending x87 FPU exceptions,  
 4-357  
 constants, 3-336  
 initialization, 3-324  
 instruction opcodes, A-22  
 x87 FPU control word  
 loading, 3-338, 3-340  
 RC field, 3-327, 3-336, 3-379  
 restoring, 3-362  
 saving, 3-365, 3-385  
 storing, 3-382  
 x87 FPU data pointer, 3-340, 3-362, 3-365, 3-385  
 x87 FPU instruction pointer, 3-340, 3-362, 3-365,  
 3-385  
 x87 FPU last opcode, 3-340, 3-362, 3-365, 3-385  
 x87 FPU status word  
 condition code flags, 3-297, 3-317, 3-399,  
 3-401, 3-404  
 loading, 3-340  
 restoring, 3-362  
 saving, 3-365, 3-385, 3-388  
 TOP field, 3-322  
 x87 FPU flags affected by instructions, 3-14  
 x87 FPU tag word, 3-340, 3-362, 3-365, 3-385  
 XADD instruction, 3-538, 4-363  
 XCHG instruction, 3-538, 4-365  
 XLAB instruction, 4-368  
 XLAT instruction, 4-368  
 XOR instruction, 3-538, 4-370  
 XORPD instruction, 4-373  
 XORPS instruction, 4-375

## INDEX

### Z

ZF (zero) flag, EFLAGS register, 3-145, 3-509,  
3-543, 3-546, 4-213, 4-354

# INTEL SALES OFFICES

## ASIA PACIFIC

### Australia

Intel Corp.  
Level 2  
448 St Kilda Road  
Melbourne VIC  
3004  
Australia  
Fax:613-9862 5599

### China

Intel Corp.  
Paharpur Business  
Centre  
Rm 709, Shaanxi  
Zhongda Int'l Bldg  
No.30 Nandajie Street  
Xian AX710002  
China  
Fax:(86 29) 7203356

Intel Corp.  
Rm 2710, Metropolitan  
Tower  
68 Zourong Rd  
Chongqing CQ  
400015  
China

Intel Corp.  
C1, 15 Flr, Fujian  
Oriental Hotel  
No. 96 East Street  
Fuzhou FJ  
350001  
China

Intel Corp.  
Rm 5803 CITIC Plaza  
233 Tianhe Rd  
Guangzhou GD  
510613  
China

Intel Corp.  
Rm 1003, Orient Plaza  
No. 235 Huayuan Street  
Nangang District  
Harbin HL  
150001  
China

Intel Corp.  
Rm 1751 World Trade  
Center, No 2  
Han Zhong Rd  
Nanjing JS  
210009  
China

Intel Corp.  
Hua Xin International  
Tower  
215 Qing Nian St.  
ShenYang LN  
110015  
China

Intel Corp.  
Suite 1128 CITIC Plaza  
Jinan  
150 Luo Yuan St.  
Jinan SN  
China

Intel Corp.  
Suite 412, Holiday Inn  
Crownse Plaza  
31, Zong Fu Street  
Chengdu SU  
610041  
China  
Fax:86-28-6785965

Intel Corp.  
Room 0724, White Rose  
Hotel  
No 750, MinZhu Road  
WuChang District  
Wuhan UB  
430071  
China

### India

Intel Corp.  
Paharpur Business  
Centre  
New Delhi DH  
110019  
India

Intel Corp.  
Hotel Rang Sharda, 6th  
Floor  
Bandra Reclamation  
Mumbai MH  
400050  
India  
Fax:91-22-6415578

Intel Corp.  
DBS Corporate Club  
31A Cathedral Garden  
Road  
Chennai TD  
600034  
India

Intel Corp.  
DBS Corporate Club  
2nd Floor, 8 A.A.C. Bose  
Road  
Calcutta WB  
700017  
India

### Japan

Intel Corp.  
Kokusai Bldg 5F, 3-1-1,  
Marunouchi  
Chiyoda-Ku, Tokyo  
1000005  
Japan

Intel Corp.  
2-4-1 Terauchi  
Toyonaka-Shi  
Osaka  
5600872  
Japan

### Malaysia

Intel Corp.  
Lot 102 1/F Block A  
Wisma Semantan  
12 Jalan Gelenggang  
Damansara Heights  
Kuala Lumpur SL  
50490  
Malaysia

### Thailand

Intel Corp.  
87 M. Thai Tower, 9th Fl.  
All Seasons Place,  
Wireless Road  
Lumpini, Patumwan  
Bangkok  
10330  
Thailand

### Viet Nam

Intel Corp.  
Hanoi Tung Shing  
Square, Ste #1106  
2 Ngo Quyen St  
Hoan Kiem District  
Hanoi  
Viet Nam

## EUROPE & AFRICA

### Belgium

Intel Corp.  
Woluwelaan 158  
Diegem  
1831  
Belgium

### Czech Rep

Intel Corp.  
Nahorni 14  
Brno  
61600  
Czech Rep

### Denmark

Intel Corp.  
Soelodden 13  
Maaloev  
DK2760  
Denmark

### Germany

Intel Corp.  
Sandstrasse 4  
Aichner  
86551  
Germany

Intel Corp.  
Dr Weyerstrasse 2  
Juelich  
52428  
Germany

Intel Corp.  
Buchenweg 4  
Wildberg  
72218  
Germany

Intel Corp.  
Kemnader Strasse 137  
Bochum  
44797  
Germany

Intel Corp.  
Klaus-Schaefer Strasse  
16-18  
Erfstadt NW  
50374  
Germany

Intel Corp.  
Heldmanskamp 37  
Lemgo NW  
32657  
Germany

### Italy

Intel Corp Italia Spa  
Milanofiori Palazzo E/4  
Assago  
Milan  
20094  
Italy  
Fax:39-02-57501221

### Netherland

Intel Corp.  
Strausslaan 31  
Heesch  
5384CW  
Netherland

### Poland

Intel Poland  
Developments, Inc  
Jerozolimskie Business  
Park  
Jerozolimskie 146c  
Warsaw  
2305  
Poland  
Fax:+48-22-570 81 40

### Portugal

Intel Corp.  
PO Box 20  
Alcabideche  
2765  
Portugal

### Spain

Intel Corp.  
Calle Rioja, 9  
Bajo F Izquierda  
Madrid  
28042  
Spain

### South Africa

Intel SA Corporation  
Bldg 14, South Wing,  
2nd Floor  
Uplands, The Woodlands  
Western Services Road  
Woodmead  
2052  
Sth Africa  
Fax:+27 11 806 4549

Intel Corp.  
19 Summit Place,  
Halfway House  
Cnr 5th and Harry  
Galaun Streets  
Midrad  
1685  
Sth Africa

### United Kingdom

Intel Corp.  
The Manse  
Silver Lane  
Needingworth CAMBS  
PE274SL  
UK

Intel Corp.  
2 Cameron Close  
Long Melford SUFFK  
CO109TS  
UK

### Israel

Intel Corp.  
MTM Industrial Center,  
P.O.Box 498  
Haifa  
31000  
Israel  
Fax:972-4-8655444

## LATIN AMERICA & CANADA

### Argentina

Intel Corp.  
Dock IV - Bldg 3 - Floor 3  
Olga Cossetini 240  
Buenos Aires  
C1107BVA  
Argentina

### Brazil

Intel Corp.  
Rua Carlos Gomez  
111/403  
Porto Alegre  
90480-003  
Brazil

Intel Corp.  
Av. Dr. Chucri Zaidan  
940 - 10th Floor  
San Paulo  
04583-904  
Brazil

### Columbia

Intel Corp.  
Av. Rio Branco,  
1 - Sala 1804  
Rio de Janeiro  
20090-003  
Brazil

### Columbia

Intel Corp.  
Carrera 7 No. 71021  
Torre B. Oficina 603  
Santefe de Bogota  
Columbia

### Mexico

Intel Corp.  
Av. Mexico No. 2798-9B,  
S.H.  
Guadalajara  
44680  
Mexico

Intel Corp.  
Torre Esmeralda II,  
7th Floor  
Blvd. Manuel Avila  
Comacho #36  
Mexico Cith DF  
11000  
Mexico

Intel Corp.  
Piso 19, Suite 4  
Av. Batallon de San  
Patricio No 111  
Monterrey, Nuevo le  
66269  
Mexico

### Canada

Intel Corp.  
168 Bonis Ave, Suite 202  
Scarborough  
MIT3V6  
Canada  
Fax:416-335-7695

Intel Corp.  
3901 Highway #7,  
Suite 403  
Vaughan  
L4L 8L5  
Canada  
Fax:905-856-8868

Intel Corp.  
999 CANADA PLACE,  
Suite 404,#11  
Vancouver BC  
V6C 3E2  
Canada  
Fax:604-844-2813

Intel Corp.  
2650 Queensview Drive,  
Suite 250  
Ottawa ON  
K2B 8H6  
Canada  
Fax:613-820-5936

Intel Corp.  
190 Attwell Drive,  
Suite 500  
Rexdale ON  
M9W 6H8  
Canada  
Fax:416-675-2438

Intel Corp.  
171 St. Clair Ave. E.,  
Suite 6  
Toronto ON  
Canada

Intel Corp.  
1033 Oak Meadow Road  
Oakville ON  
L6M 1J6  
Canada

**USA**  
**California**  
Intel Corp.  
551 Lundy Place  
Milpitas CA  
95035-6833  
USA  
Fax:408-451-8266

Intel Corp.  
1551 N. Tustin Avenue,  
Suite 800  
Santa Ana CA  
92705  
USA  
Fax:714-541-9157

Intel Corp.  
Executive Center del Mar  
12230 El Camino Real  
Suite 140  
San Diego CA  
92130  
USA  
Fax:858-794-5805

Intel Corp.  
1960 E. Grand Avenue,  
Suite 150  
El Segundo CA  
90245  
USA  
Fax:310-640-7133

Intel Corp.  
23120 Alicia Parkway,  
Suite 215  
Mission Viejo CA  
92692  
USA  
Fax:949-586-9499

Intel Corp.  
30851 Agoura Road  
Suite 202  
Agoura Hills CA  
91301  
USA  
Fax:818-874-1166

Intel Corp.  
28202 Cabot Road,  
Suite #363 & #371  
Laguna Niguel CA  
92677  
USA

Intel Corp.  
657 S Cendros Avenue  
Solana Beach CA  
90075  
USA

Intel Corp.  
43769 Abeloe Terrace  
Fremont CA  
94539  
USA

Intel Corp.  
1721 Warburton, #6  
Santa Clara CA  
95050  
USA

**Colorado**  
Intel Corp.  
600 S. Cherry Street,  
Suite 700  
Denver CO  
80222  
USA  
Fax:303-322-8670

**Connecticut**  
Intel Corp.  
Lee Farm Corporate Pk  
83 Wooster Heights  
Road  
Danbury CT  
6810  
USA  
Fax:203-778-2168

**Florida**  
Intel Corp.  
7777 Glades Road  
Suite 310B  
Boca Raton FL  
33434  
USA  
Fax:813-367-5452

**Georgia**  
Intel Corp.  
20 Technology Park,  
Suite 150  
Norcross GA  
30092  
USA  
Fax:770-448-0875

Intel Corp.  
Three Northwinds Center  
2500 Northwinds  
Parkway, 4th Floor  
Alpharetta GA  
30092  
USA  
Fax:770-663-6354

**Idaho**  
Intel Corp.  
910 W. Main Street, Suite  
236  
Boise ID  
83702  
USA  
Fax:208-331-2295

**Illinois**  
Intel Corp.  
425 N. Martingale Road  
Suite 1500  
Schaumburg IL  
60173  
USA  
Fax:847-605-9762

Intel Corp.  
999 Plaza Drive  
Suite 360  
Schaumburg IL  
60173  
USA

Intel Corp.  
551 Arlington Lane  
South Elgin IL  
60177  
USA

**Indiana**  
Intel Corp.  
9465 Counselors Row,  
Suite 200  
Indianapolis IN  
46240  
USA  
Fax:317-805-4939

**Massachusetts**  
Intel Corp.  
125 Nagog Park  
Acton MA  
01720  
USA  
Fax:978-266-3867

Intel Corp.  
59 Composit Way  
suite 202  
Lowell MA  
01851  
USA

Intel Corp.  
800 South Street,  
Suite 100  
Waltham MA  
02154  
USA

**Maryland**  
Intel Corp.  
131 National Business  
Parkway, Suite 200  
Annapolis Junction MD  
20701  
USA  
Fax:301-206-3678

**Michigan**  
Intel Corp.  
32255 Northwestern  
Hwy., Suite 212  
Farmington Hills MI  
48334  
USA  
Fax:248-851-8770

**Minnesota**  
Intel Corp.  
3600 W 80Th St  
Suite 450  
Bloomington MN  
55431  
USA  
Fax:952-831-6497

**North Carolina**  
Intel Corp.  
2000 CentreGreen Way,  
Suite 190  
Cary NC  
27513  
USA  
Fax:919-678-2818

**New Hampshire**  
Intel Corp.  
7 Suffolk Park  
Nashua NH  
03063  
USA

**New Jersey**  
Intel Corp.  
90 Woodbridge Center  
Dr. Suite. 240  
Woodbridge NJ  
07095  
USA  
Fax:732-602-0096

**New York**  
Intel Corp.  
628 Crosskeys Office Pk  
Fairport NY  
14450  
USA  
Fax:716-223-2561

Intel Corp.  
888 Veterans Memorial  
Highway  
Suite 530  
Hauppauge NY  
11788  
USA  
Fax:516-234-5093

**Ohio**  
Intel Corp.  
3401 Park Center Drive  
Suite 220  
Dayton OH  
45414  
USA  
Fax:937-890-8658

Intel Corp.  
56 Milford Drive  
Suite 205  
Hudson OH  
44236  
USA  
Fax:216-528-1026

**Oregon**  
Intel Corp.  
15254 NW Greenbrier  
Parkway, Building B  
Beaverton OR  
97006  
USA  
Fax:503-645-8181

**Pennsylvania**  
Intel Corp.  
925 Harvest Drive  
Suite 200  
Blue Bell PA  
19422  
USA  
Fax:215-641-0785

Intel Corp.  
7500 Brooktree  
Suite 213  
Wexford PA  
15090  
USA  
Fax:714-541-9157

**Texas**  
Intel Corp.  
5000 Quorum Drive,  
Suite 750  
Dallas TX  
75240  
USA  
Fax:972-233-1325

Intel Corp.  
20445 State Highway  
249, Suite 300  
Houston TX  
77070  
USA  
Fax:281-376-2891

Intel Corp.  
8911 Capital of Texas  
Hwy, Suite 4230  
Austin TX  
78759  
USA  
Fax:512-338-9335

Intel Corp.  
7739 La Verdura Drive  
Dallas TX  
75248  
USA

Intel Corp.  
77269 La Cabeza Drive  
Dallas TX  
75249  
USA

Intel Corp.  
3307 Northland Drive  
Austin TX  
78731  
USA

Intel Corp.  
15190 Prestonwood  
Blvd. #925  
Dallas TX  
75248  
USA  
Intel Corp.

**Washington**  
Intel Corp.  
2800 156Th Ave. SE  
Suite 105  
Bellevue WA  
98007  
USA  
Fax:425-746-4495

Intel Corp.  
550 Kirkland Way  
Suite 200  
Kirkland WA  
98033  
USA

**Wisconsin**  
Intel Corp.  
405 Forest Street  
Suites 109/112  
Oconomowoc WI  
53066  
USA