Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 2B: Instruction Set Reference, N-Z

NOTE: The Intel 64 and IA-32 Architectures 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: 253667-021
September 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® 64 architecture 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® 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. For more information, see http://www.intel.com/technology/hyperthread/index.htm; 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. Intel® Virtualization Technology-enabled BIOS and VMM applications are currently in development.

64-bit computing on Intel architecture requires a computer system with a processor, chipset, BIOS, operating system, device drivers and applications enabled for Intel® 64 architecture. Processors will not operate (including 32-bit operation) without an Intel® 64 architecture-enabled BIOS. Performance will vary depending on your hardware and software configurations. Consult with your system vendor for more information.

Enabling Execute Disable Bit functionality requires a PC with a processor with Execute Disable Bit capability and a supporting operating system. Check with your PC manufacturer on whether your system delivers Execute Disable Bit functionality.

Intel, Pentium, Intel Xeon, Intel NetBurst, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo, Intel Core 2 Extreme, Intel Pentium D, Itanium, Intel SpeedStep, 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

CHAPTER 4
INSTRUCTION SET REFERENCE, N-Z

4.1 INSTRUCTIONS (N-Z)

Chapter 4 continues an alphabetical discussion of Intel® 64 and IA-32 instructions (N-Z). See also: Chapter 3, "Instruction Set Reference, A-M," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

NEG—Two's Complement Negation

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

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a 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 a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a 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

Table 4-1. Recommended Multi-Byte Sequence of NOP Instruction (Contd.)

Length	Assembly	Byte Sequence
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 *r/m8*	Valid	Valid	Reverse each bit of *r/m8.*
REX + F6 /2	NOT *r/m8**	Valid	N.E.	Reverse each bit of *r/m8.*
F7 /2	NOT *r/m16*	Valid	Valid	Reverse each bit of *r/m16.*
F7 /2	NOT *r/m32*	Valid	Valid	Reverse each bit of *r/m32.*
REX.W + F7 /2	NOT *r/m64*	Valid	N.E.	Reverse each bit of *r/m64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a 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 a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a 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 *ib*	OR AL, *imm8*	Valid	Valid	AL OR *imm8.*
0D *iw*	OR AX, *imm16*	Valid	Valid	AX OR *imm16.*
0D *id*	OR EAX, *imm32*	Valid	Valid	EAX OR *imm32.*
REX.W + 0D *id*	OR RAX, *imm32*	Valid	N.E.	RAX OR *imm32 (sign-extended).*
80 /1 *ib*	OR *r/m8, imm8*	Valid	Valid	*r/m8* OR *imm8.*
REX + 80 /1 *ib*	OR *r/m8*, imm8*	Valid	N.E.	*r/m8* OR *imm8.*
81 /1 *iw*	OR *r/m16, imm16*	Valid	Valid	*r/m16* OR *imm16.*
81 /1 *id*	OR *r/m32, imm32*	Valid	Valid	*r/m32* OR *imm32.*
REX.W + 81 /1 *id*	OR *r/m64, imm32*	Valid	N.E.	*r/m64* OR *imm32 (sign-extended).*
83 /1 *ib*	OR *r/m16, imm8*	Valid	Valid	*r/m16* OR *imm8 (sign-extended).*
83 /1 *ib*	OR *r/m32, imm8*	Valid	Valid	*r/m32* OR *imm8 (sign-extended).*
REX.W + 83 /1 *ib*	OR *r/m64, imm8*	Valid	N.E.	*r/m64* OR *imm8 (sign-extended).*
08 /*r*	OR *r/m8, r8*	Valid	Valid	*r/m8* OR *r8.*
REX + 08 /*r*	OR *r/m8*, r8**	Valid	N.E.	*r/m8* OR *r8.*
09 /*r*	OR *r/m16, r16*	Valid	Valid	*r/m16* OR *r16.*
09 /*r*	OR *r/m32, r32*	Valid	Valid	*r/m32* OR *r32.*
REX.W + 09 /*r*	OR *r/m64, r64*	Valid	N.E.	*r/m64* OR *r64.*
0A /*r*	OR *r8, r/m8*	Valid	Valid	*r8* OR *r/m8.*
REX + 0A /*r*	OR *r8*, r/m8**	Valid	N.E.	*r8* OR *r/m8.*
0B /*r*	OR *r16, r/m16*	Valid	Valid	*r16* OR *r/m16.*
0B /*r*	OR *r32, r/m32*	Valid	Valid	*r32* OR *r/m32.*
REX.W + 0B /*r*	OR *r64, r/m64*	Valid	N.E.	*r64* OR *r/m64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a 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 a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a 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.

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 *xmm1*, *xmm2/m128*	Valid	Valid	Bitwise OR of *xmm2/m128* and *xmm1*.

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 a 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.

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 *xmm1, xmm2/m128*	Valid	Valid	Bitwise OR of *xmm2/m128* and *xmm1.*

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 a 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.

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 *ib*	OUT *imm8*, AL	Valid	Valid	Output byte in AL to I/O port address *imm8*.
E7 *ib*	OUT *imm8*, AX	Valid	Valid	Output word in AX to I/O port address *imm8.*
E7 *ib*	OUT *imm8*, EAX	Valid	Valid	Output doubleword in EAX to I/O port address *imm8*.
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 *Intel® 64 and IA-32 Architectures 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.

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.

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, *m8*	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTS DX, *m16*	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTS DX, *m32*	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 double-word), 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.

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 *Intel® 64 and IA-32 Architectures 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 support 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.

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 RSI + 1;
                        ELSE RSI ← RSI or – 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 RSI + 2;
                        ELSE RSI ← RSI or – 2;
                    FI;
```

```
                ELSE (* 32-Bit Address Size *)
                    IF DF = 0
                        THEN    ESI ← ESI + 2;
                        ELSE    ESI ← ESI – 2;
                    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 RSI + 4;
                        ELSE RSI ← RSI or – 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.

PABSB/PABSW/PABSD — Packed Absolute Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 38 1C /r	PABSB mm1, mm2/m64	Valid	Valid	Compute the absolute value of bytes in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1C /r	PABSB xmm1, xmm2/m128	Valid	Valid	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
0F 38 1D /r	PABSW mm1, mm2/m64	Valid	Valid	Compute the absolute value of 16-bit integers in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1D /r	PABSW xmm1, xmm2/m128	Valid	Valid	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
0F 38 1E /r	PABSD mm1, mm2/m64	Valid	Valid	Compute the absolute value of 32-bit integers in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1E /r	PABSD xmm1, xmm2/m128	Valid	Valid	Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.

Description

PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on 16-bit words, and PABSD operates on signed 32-bit integers. The source operand can be an MMX 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 or an XMM register. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Operation

```
PABSB with 64 bit operands
    Unsigned DEST[7..0] <- ABS(SRC[7..0])
    Repeat operation for 2nd through 7th bytes
    Unsigned DEST[63..56]  <- ABS(SRC[63..56])
```

PABSB with 128 bit operands:

Unsigned DEST[7..0] <- ABS(SRC[7..0])
Repeat operation for 2nd through 15th bytes
Unsigned DEST[127..120] <- ABS(SRC[127..120])

PABSW with 64 bit operands:

Unsigned DEST[15..0] <- ABS(SRC[15..0])
Repeat operation for 2nd through 3rd 16-bit words
Unsigned DEST[63..48] <- ABS(SRC[63..48])

PABSW with 128 bit operands:

Unsigned DEST[15..0] <- ABS(SRC[15..0])
Repeat operation for 2nd through 7th 16-bit words
Unsigned DEST[127..112] <- ABS(SRC[127..112])

PABSD with 64 bit operands:

Unsigned DEST[31..0] <- ABS(SRC[31..0])
Unsigned DEST[63..32] <- ABS(SRC[63..32])

PABSD with 128 bit operands:

Unsigned DEST[31..0] <- ABS(SRC[31..0])
Repeat operation for 2nd through 3rd 32-bit double words
Unsigned DEST[127..96] <- ABS(SRC[127..96])

Intel C/C++ Compiler Intrinsic Equivalents

PABSB	__m64 _mm_abs_pi8 (__m64 a)
PABSB	__m128i _mm_abs_epi8 (__m128i a)
PABSW	__m64 _mm_abs_pi16 (__m64 a)
PABSW	__m128i _mm_abs_epi16 (__m128i a)
PABSD	__m64 _mm_abs_pi32 (__m64 a)
PABSD	__m128i _mm_abs_epi32 (__m128i a)

Protected Mode Exceptions

#GP(0):	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
	(128-bit operations only) If not aligned on 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.

	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#AC(0)	(64-bit operations only) If alignment checking is enabled and unaligned memory reference is made while the current privilege level is 3.

Real 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 not aligned on 16-byte boundary, regardless of segment.
#UD:	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0
#NM	If TS bit in CR0 is set.
#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)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and 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.
	If CPUID.01H:ECX.SSSE3[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.

PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 63 /r	PACKSSWB *mm1, mm2/m64*	Valid	Valid	Converts 4 packed signed word integers from *mm1* and from *mm2/m64* into 8 packed signed byte integers in *mm1* using signed saturation.
66 0F 63 /r	PACKSSWB *xmm1, xmm2/m128*	Valid	Valid	Converts 8 packed signed word integers from *xmm1* and from *xxm2/m128* into 16 packed signed byte integers in *xxm1* using signed saturation.
0F 6B /r	PACKSSDW *mm1, mm2/m64*	Valid	Valid	Converts 2 packed signed doubleword integers from *mm1* and from *mm2/m64* into 4 packed signed word integers in *mm1* using signed saturation.
66 0F 6B /r	PACKSSDW *xmm1, xmm2/m128*	Valid	Valid	Converts 4 packed signed doubleword integers from *xmm1* and from *xxm2/m128* into 8 packed signed word integers in *xxm1* 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.

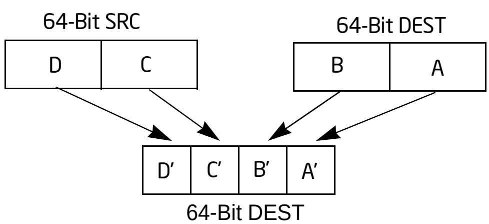


Figure 4-1. Operation of the PACKSSDW Instruction Using 64-bit Operands

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 a 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]);
    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 *mm, mm/m64*	Valid	Valid	Converts 4 signed word integers from *mm* and 4 signed word integers from *mm/m64* into 8 unsigned byte integers in *mm* using unsigned saturation.
66 0F 67 /*r*	PACKUSWB *xmm1, xmm2/m128*	Valid	Valid	Converts 8 signed word integers from *xmm1* and 8 signed word integers from *xmm2/m128* into 16 unsigned byte integers in *xmm1* 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 a 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.

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 *mm, mm/m64*	Valid	Valid	Add packed byte integers from *mm/m64* and *mm*.
66 0F FC /*r*	PADDB *xmm1, xmm2/m128*	Valid	Valid	Add packed byte integers from *xmm2/m128* and *xmm1*.
0F FD /*r*	PADDW *mm, mm/m64*	Valid	Valid	Add packed word integers from *mm/m64* and *mm*.
66 0F FD /*r*	PADDW *xmm1, xmm2/m128*	Valid	Valid	Add packed word integers from *xmm2/m128* and *xmm1*.
0F FE /*r*	PADDD *mm, mm/m64*	Valid	Valid	Add packed doubleword integers from *mm/m64* and *mm*.
66 0F FE /*r*	PADDD *xmm1, xmm2/m128*	Valid	Valid	Add packed doubleword integers from *xmm2/m128* and *xmm1*.

Description

Performs a 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a 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 a 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.

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 *mm1, mm2/m64*	Valid	Valid	Add quadword integer *mm2/m64* to *mm1.*
66 0F D4 /*r*	PADDQ *xmm1, xmm2/m128*	Valid	Valid	Add packed quadword integers *xmm2/m128* to *xmm1.*

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, a 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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
PADDQ instruction with 64-Bit operands:
    DEST[63:0] ← DEST[63:0] + SRC[63:0];

PADDQ instruction with 128-Bit operands:
    DEST[63:0] ← DEST[63:0] + SRC[63:0];
    DEST[127:64] ← DEST[127:64] + SRC[127:64];
```

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.

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 *mm, mm/m64*	Valid	Valid	Add packed signed byte integers from *mm/m64 and mm* and saturate the results.
66 0F EC /*r*	PADDSB *xmm1, xmm2/m128*	Valid	Valid	Add packed signed byte integers from *xmm2/m128* and *xmm1* saturate the results.
0F ED /*r*	PADDSW *mm, mm/m64*	Valid	Valid	Add packed signed word integers from *mm/m64 and mm* and saturate the results.
66 0F ED /*r*	PADDSW *xmm1, xmm2/m128*	Valid	Valid	Add packed signed word integers from *xmm2/m128* and *xmm1* and saturate the results.

Description

Performs a 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a 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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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.
#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 *mm, mm/m64*	Valid	Valid	Add packed unsigned byte integers from *mm/m64 and mm* and saturate the results.
66 0F DC /*r*	PADDUSB xmm1, xmm2/m128	Valid	Valid	Add packed unsigned byte integers from *xmm2/m128* and *xmm1* saturate the results.
0F DD /*r*	PADDUSW *mm, mm/m64*	Valid	Valid	Add packed unsigned word integers from *mm/m64 and mm* and saturate the results.
66 0F DD /*r*	PADDUSW xmm1, xmm2/m128	Valid	Valid	Add packed unsigned word integers from *xmm2/m128* to *xmm1* and saturate the results.

Description

Performs a 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a 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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
PADDUSB 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]

PADDUSB 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]);

PADDUSW 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] );

PADDUSW 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

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.

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.

PALIGNR — Packed Align Right

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 3A 0F	PALIGNR mm1, mm2/m64, imm8	Valid	Valid	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant into mm1.
66 0F 3A 0F	PALIGNR xmm1, xmm2/m128, imm8	Valid	Valid	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant into xmm1

Description

PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX or an XMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e. 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Operation

```
PALIGNR with 64-bit operands:
    temp1[127:0] = CONCATENATE(DEST,SRC)>>(imm8*8)
    DEST[63:0] = temp1[63:0]

PALIGNR with 128-bit operands:
    temp1[255:0] = CONCATENATE(DEST,SRC)>>(imm8*8)
    DEST[127:0] = temp1[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalents

PALIGNR __m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)

PALIGNR __m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
	(128-bit operations only) If not aligned on 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#AC(0)	(64-bit operations only) If alignment checking is enabled and unaligned memory reference is made while the current privilege level is 3.

Real 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 not aligned on 16-byte boundary, regardless of segment.
#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#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)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and 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.
	If CPUID.01H:ECX.SSSE3[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.

PAND—Logical AND

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

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 a 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.
#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 *mm, mm/m64*	Valid	Valid	Bitwise AND NOT of *mm/m64* and *mm*.
66 0F DF /*r*	PANDN *xmm1, xmm2/m128*	Valid	Valid	Bitwise AND NOT of *xmm2/m128* and *xmm1*.

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 a 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.

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 *mm1, mm2/m64*	Valid	Valid	Average packed unsigned byte integers from mm2/m64 and mm1 with rounding.
66 0F E0, /*r*	PAVGB *xmm1, xmm2/m128*	Valid	Valid	Average packed unsigned byte integers from *xmm2/m128* and *xmm1* with rounding.
0F E3 /*r*	PAVGW *mm1, mm2/m64*	Valid	Valid	Average packed unsigned word integers from mm2/m64 and mm1 with rounding.
66 0F E3 /*r*	PAVGW *xmm1, xmm2/m128*	Valid	Valid	Average packed unsigned word integers from *xmm2/m128* and *xmm1* with rounding.

Description

Performs a 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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
PAVGB instruction with 64-bit operands:
    SRC[7:0) ← (SRC[7:0) + DEST[7:0) + 1) >> 1; (* Temp sum before shifting is 9 bits *)
    (* Repeat operation performed for bytes 2 through 6 *)
    SRC[63:56) ← (SRC[63:56) + DEST[63:56) + 1) >> 1;

PAVGW instruction with 64-bit operands:
    SRC[15:0) ← (SRC[15:0) + DEST[15:0) + 1) >> 1; (* Temp sum before shifting is 17 bits *)
    (* Repeat operation performed for words 2 and 3 *)
    SRC[63:48) ← (SRC[63:48) + DEST[63:48) + 1) >> 1;
```

PAVGB instruction with 128-bit operands:

```
SRC[7:0) ← (SRC[7:0) + DEST[7:0) + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 14 *)
SRC[63:56) ← (SRC[63:56) + DEST[63:56) + 1) >> 1;
```

PAVGW instruction with 128-bit operands:

```
SRC[15:0) ← (SRC[15:0) + DEST[15:0) + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 6 *)
SRC[127:48) ← (SRC[127:112) + DEST[127:112) + 1) >> 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.
#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 *mm, mm/m64*	Valid	Valid	Compare packed bytes in *mm/m64* and *mm* for equality.
66 0F 74 /*r*	PCMPEQB *xmm1, xmm2/m128*	Valid	Valid	Compare packed bytes in *xmm2/m128* and xmm1 for equality.
0F 75 /*r*	PCMPEQW *mm, mm/m64*	Valid	Valid	Compare packed words in *mm/m64* and *mm* for equality.
66 0F 75 /*r*	PCMPEQW *xmm1, xmm2/m128*	Valid	Valid	Compare packed words in *xmm2/m128* and xmm1 for equality.
0F 76 /*r*	PCMPEQD *mm, mm/m64*	Valid	Valid	Compare packed doublewords in *mm/m64* and *mm* for equality.
66 0F 76 /*r*	PCMPEQD *xmm1, xmm2/m128*	Valid	Valid	Compare packed doublewords in *xmm2/m128* and xmm1 for equality.

Description

Performs a 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 a 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;

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.

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 *mm, mm/m64*	Valid	Valid	Compare packed signed byte integers in *mm* and *mm/m64* for greater than.
66 0F 64 /*r*	PCMPGTB *xmm1, xmm2/m128*	Valid	Valid	Compare packed signed byte integers in *xmm1* and *xmm2/m128* for greater than.
0F 65 /*r*	PCMPGTW *mm, mm/m64*	Valid	Valid	Compare packed signed word integers in *mm* and *mm/m64* for greater than.
66 0F 65 /*r*	PCMPGTW *xmm1, xmm2/m128*	Valid	Valid	Compare packed signed word integers in *xmm1* and *xmm2/m128* for greater than.
0F 66 /*r*	PCMPGTD *mm, mm/m64*	Valid	Valid	Compare packed signed doubleword integers in *mm* and *mm/m64* for greater than.
66 0F 66 /*r*	PCMPGTD *xmm1, xmm2/m128*	Valid	Valid	Compare packed signed doubleword integers in *xmm1* and *xmm2/m128* for greater than.

Description

Performs a 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 date 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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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.

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.

PEXTRW—Extract Word

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

PHADDW/PHADDD — Packed Horizontal Add

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 38 01 /r	PHADDW mm1, mm2/m64	Valid	Valid	Add 16-bit signed integers horizontally, pack to MM1.
66 0F 38 01 /r	PHADDW xmm1, xmm2/m128	Valid	Valid	Add 16-bit signed integers horizontally, pack to XMM1.
0F 38 02 /r	PHADDD mm1, mm2/m64	Valid	Valid	Add 32-bit signed integers horizontally, pack to MM1.
66 0F 38 02 /r	PHADDD xmm1, xmm2/m128	Valid	Valid	Add 32-bit signed integers horizontally, pack to XMM1.

Description

PHADDW adds two adjacent 16-bit signed integers horizontally from the source and destination operands and packs the 16-bit signed results to the destination operand (first operand). PHADDD adds two adjacent 32-bit signed integers horizontally from the source and destination operands and packs the 32-bit signed results to the destination operand (first operand). Both operands can be MMX or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Operation

```
PHADDW with 64-bit operands:
    mm1[15-0]  = mm1[31-16] + mm1[15-0];
    mm1[31-16] = mm1[63-48] + mm1[47-32];
    mm1[47-32] = mm2/m64[31-16] + mm2/m64[15-0];
    mm1[63-48] = mm2/m64[63-48] + mm2/m64[47-32];

PHADDW with 128-bit operands :
    xmm1[15-0] = xmm1[31-16] + xmm1[15-0];
    xmm1[31-16] = xmm1[63-48] + xmm1[47-32];
    xmm1[47-32] = xmm1[95-80] + xmm1[79-64];
    xmm1[63-48] = xmm1[127-112] + xmm1[111-96];
    xmm1[79-64] = xmm2/m128[31-16] + xmm2/m128[15-0];
    xmm1[95-80] = xmm2/m128[63-48] + xmm2/m128[47-32];
    xmm1[111-96] = xmm2/m128[95-80] + xmm2/m128[79-64];
    xmm1[127-112] = xmm2/m128[127-112] + xmm2/m128[111-96];
```

PHADDD with 64-bit operands :

```
mm1[31-0]  = mm1[63-32] + mm1[31-0];
mm1[63-32] = mm2/m64[63-32] + mm2/m64[31-0];
```

PHADDD with 128-bit operands:

```
xmm1[31-0] = xmm1[63-32] + xmm1[31-0];
xmm1[63-32] = xmm1[127-96] + xmm1[95-64];
xmm1[95-64] = xmm2/m128[63-32] + xmm2/m128[31-0];
xmm1[127-96] = xmm2/m128[127-96] + xmm2/m128[95-64];
```

Intel C/C++ Compiler Intrinsic Equivalents

PHADDW __m64 _mm_hadd_pi16 (__m64 a, __m64 b)

PHADDW __m128i _mm_hadd_epi16 (__m128i a, __m128i b)

PHADDD __m64 _mm_hadd_pi32 (__m64 a, __m64 b)

PHADDD __m128i _mm_hadd_epi32 (__m128i a, __m128i b)

Protected Mode Exceptions

#GP(0):	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
	(128-bit operations only) If not aligned on 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If CR0.EM(bit 2)= 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#AC(0)	(64-bit operations only) If alignment checking is enabled and unaligned memory reference is made while the current privilege level is 3.

Real 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 not aligned on 16-byte boundary, regardless of segment.

#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#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)	If a page fault occurs.
#AC(0)	(64-bit operations only). If alignment checking is enabled and 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.
	If CPUID.01H:ECX.SSSE3[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.

PHADDSW — Packed Horizontal Add and Saturate

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 38 03 /r	PHADDSW mm1, mm2/m64	Valid	Valid	Add 16-bit signed integers horizontally, pack saturated integers to MM1.
66 0F 38 03 /r	PHADDSW xmm1, xmm2/m128	Valid	Valid	Add 16-bit signed integers horizontally, pack saturated integers to XMM1.

Description

PHADDSW adds two adjacent signed 16-bit integers horizontally from the source and destination operands and saturates the signed results; packs the signed, saturated 16-bit results to the destination operand (first operand) Both operands can be MMX or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Operation

```
PHADDSW with 64-bit operands:
    mm1[15-0]  = SaturateToSignedWord((mm1[31-16] + mm1[15-0]);
    mm1[31-16] = SaturateToSignedWord(mm1[63-48] + mm1[47-32]);
    mm1[47-32] = SaturateToSignedWord(mm2/m64[31-16] + mm2/m64[15-0]);
    mm1[63-48] = SaturateToSignedWord(mm2/m64[63-48] + mm2/m64[47-32]);

PHADDSW with 128-bit operands :
    xmm1[15-0]= SaturateToSignedWord(xmm1[31-16] + xmm1[15-0]);
    xmm1[31-16] = SaturateToSignedWord(xmm1[63-48] + xmm1[47-32]);
    xmm1[47-32] = SaturateToSignedWord(xmm1[95-80] + xmm1[79-64]);
    xmm1[63-48] = SaturateToSignedWord(xmm1[127-112] + xmm1[111-96]);
    xmm1[79-64] = SaturateToSignedWord(xmm2/m128[31-16] + xmm2/m128[15-0]);
    xmm1[95-80] = SaturateToSignedWord(xmm2/m128[63-48] + xmm2/m128[47-32]);
    xmm1[111-96] = SaturateToSignedWord(xmm2/m128[95-80] + xmm2/m128[79-64]);
    xmm1[127-112] = SaturateToSignedWord(xmm2/m128[127-112] + xmm2/m128[111-96]);
```

Intel C/C++ Compiler Intrinsic Equivalent

PHADDSW __m64 _mm_hadds_pi16 (__m64 a, __m64 b)

PHADDSW __m128i _mm_hadds_epi16 (__m128i a, __m128i b)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
	(128-bit operations only) If not aligned on 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#AC(0):	(64-bit operations only) If alignment checking is enabled and unaligned memory reference is made while the current privilege level is 3.

Real 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 not aligned on 16-byte boundary, regardless of segment.
#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#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)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and 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.
	If CPUID.01H:ECX.SSSE3[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.

PHSUBW/PHSUBD — Packed Horizontal Subtract

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 38 05 /r	PHSUBW mm1, mm2/m64	Valid	Valid	Subtract 16-bit signed integers horizontally, pack to MM1.
66 0F 38 05 /r	PHSUBW xmm1, xmm2/m128	Valid	Valid	Subtract 16-bit signed integers horizontally, pack to XMM1.
0F 38 06 /r	PHSUBD mm1, mm2/m64	Valid	Valid	Subtract 32-bit signed integers horizontally, pack to MM1.
66 0F 38 06 /r	PHSUBD xmm1, xmm2/m128	Valid	Valid	Subtract 32-bit signed integers horizontally, pack to XMM1.

Description

PHSUBW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands, and packs the signed 16-bit results to the destination operand (first operand). PHSUBD performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair, and packs the signed 32-bit result to the destination operand. Both operands can be MMX or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Operation

```
PHSUBW with 64-bit operands:
    mm1[15-0] = mm1[15-0] - mm1[31-16];
    mm1[31-16] = mm1[47-32] - mm1[63-48];
    mm1[47-32] = mm2/m64[15-0] - mm2/m64[31-16];
    mm1[63-48] = mm2/m64[47-32] - mm2/m64[63-48];

PHSUBW with 128-bit operands:
    xmm1[15-0] = xmm1[15-0] - xmm1[31-16];
    xmm1[31-16] = xmm1[47-32] - xmm1[63-48];
    xmm1[47-32] = xmm1[79-64] - xmm1[95-80];
    xmm1[63-48] = xmm1[111-96] - xmm1[127-112];
```

xmm1[79-64] = xmm2/m128[15-0] - xmm2/m128[31-16];
xmm1[95-80] = xmm2/m128[47-32] - xmm2/m128[63-48];
xmm1[111-96] = xmm2/m128[79-64] - xmm2/m128[95-80];
xmm1[127-112] = xmm2/m128[111-96] - xmm2/m128[127-112];

PHSUBD with 64-bit operands:
mm1[31-0] = mm1[31-0] - mm1[63-32];
mm1[63-32] = mm2/m64[31-0] - mm2/m64[63-32];

PHSUBD with 128-bit operands:
xmm1[31-0] = xmm1[31-0] - xmm1[63-32];
xmm1[63-32] = xmm1[95-64] - xmm1[127-96];
xmm1[95-64] = xmm2/m128[31-0] - xmm2/m128[63-32];
xmm1[127-96] = xmm2/m128[95-64] - xmm2/m128[127-96];

Intel C/C++ Compiler Intrinsic Equivalents

PHSUBW __m64 _mm_hsub_pi16 (__m64 a, __m64 b)

PHSUBW __m128i _mm_hsub_epi16 (__m128i a, __m128i b)

PHSUBD __m64 _mm_hsub_pi32 (__m64 a, __m64 b)

PHSUBD __m128i _mm_hsub_epi32 (__m128i a, __m128i b)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
	(128-bit operations only) If not aligned on 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#MF	If there is a pending x87 FPU exception (64-bit operations only).
#AC(0)	(64-bit operations only) If alignment checking is enabled and unaligned memory reference is made while the current privilege level is 3.

Real 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 not aligned on 16-byte boundary, regardless of segment.
#UD:	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#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)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and 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.
	If CPUID.01H:ECX.SSSE3[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.

PHSUBSW — Packed Horizontal Subtract and Saturate

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 38 07 /r	PHSUBSW mm1, mm2/m64	Valid	Valid	Subtract 16-bit signed integer horizontally, pack saturated integers to MM1.
66 0F 38 07 /r	PHSUBSW xmm1, xmm2/m128	Valid	Valid	Subtract 16-bit signed integer horizontally, pack saturated integers to XMM1

Description

PHSUBSW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed to the destination operand (first operand). Both operands can be MMX or XMM register. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Operation

```
PHSUBSW with 64-bit operands:
    mm1[15-0] = SaturateToSignedWord(mm1[15-0] - mm1[31-16]);
    mm1[31-16] = SaturateToSignedWord(mm1[47-32] - mm1[63-48]);
    mm1[47-32] = SaturateToSignedWord(mm2/m64[15-0] - mm2/m64[31-16]);
    mm1[63-48] = SaturateToSignedWord(mm2/m64[47-32] - mm2/m64[63-48]);

PHSUBSW with 128-bit operands:
    xmm1[15-0] = SaturateToSignedWord(xmm1[15-0] - xmm1[31-16]);
    xmm1[31-16] = SaturateToSignedWord(xmm1[47-32] - xmm1[63-48]);
    xmm1[47-32] = SaturateToSignedWord(xmm1[79-64] - xmm1[95-80]);
    xmm1[63-48] = SaturateToSignedWord(xmm1[111-96] - xmm1[127-112]);
    xmm1[79-64] = SaturateToSignedWord(xmm2/m128[15-0] - xmm2/m128[31-16]);
    xmm1[95-80] =SaturateToSignedWord(xmm2/m128[47-32] - xmm2/m128[63-48]);
    xmm1[111-96] =SaturateToSignedWord(xmm2/m128[79-64] - xmm2/m128[95-80]);
    xmm1[127-112]= SaturateToSignedWord(xmm2/m128[111-96] - xmm2/m128[127-112]);
```

Intel C/C++ Compiler Intrinsic Equivalent

PHSUBSW __m64 _mm_hsubs_pi16 (__m64 a, __m64 b)

PHSUBSW __m128i _mm_hsubs_epi16 (__m128i a, __m128i b)

Protected Mode Exceptions

#GP(0)	if a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
	If not aligned on 16-byte boundary, regardless of segment (128-bit operations only).
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0 (128-bit operations only).
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#MF	If there is a pending x87 FPU exception (64-bit operations only).
#AC(0)	If alignment checking is enabled and unaligned memory reference is made while the current privilege level is 3 (64-bit operations only).

Real Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
	If not aligned on 16-byte boundary, regardless of segment (128-bit operations only).
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0 (128-bit operations only).
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#MF	If there is a pending x87 FPU exception (64-bit operations only).

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode.

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

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:ECX.SSSE3[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.

PINSRW—Insert Word

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

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 a 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;
        CASE (Determine word position) OF
            SEL ← 0:    MASK ← 0000000000000000000000000000FFFFH;
            SEL ← 1:    MASK ← 000000000000000000000000FFFF0000H;
            SEL ← 2:    MASK ← 00000000000000000000FFFF00000000H;
            SEL ← 3:    MASK ← 0000000000000000FFFF000000000000H;
            SEL ← 4:    MASK ← 000000000000FFFF0000000000000000H;
            SEL ← 5:    MASK ← 00000000FFFF00000000000000000000H;
            SEL ← 6:    MASK ← 0000FFFF000000000000000000000000H;
            SEL ← 7:    MASK ← FFFF0000000000000000000000000000H;
    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.

PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 38 04 /r	PMADDUBSW mm1, mm2/m64	Valid	Valid	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to MM1.
66 0F 38 04 /r	PMADDUBSW xmm1, xmm2/m128	Valid	Valid	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to XMM1.

Description

PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Operation

```
PMADDUBSW with 64 bit operands:
    DEST[15-0] = SaturateToSignedWord(SRC[15-8]*DEST[15-8]+SRC[7-0]*DEST[7-0]);
    DEST[31-16] = SaturateToSignedWord(SRC[31-24]*DEST[31-24]+SRC[23-16]*DEST[23-16]);
    DEST[47-32] = SaturateToSignedWord(SRC[47-40]*DEST[47-40]+SRC[39-32]*DEST[39-32]);
    DEST[63-48] = SaturateToSignedWord(SRC[63-56]*DEST[63-56]+SRC[55-48]*DEST[55-48]);

PMADDUBSW with 128 bit operands:
    DEST[15-0] = SaturateToSignedWord(SRC[15-8]* DEST[15-8]+SRC[7-0]*DEST[7-0]);
    // Repeat operation for 2nd through 7th word
    SRC1/DEST[127-112] = SaturateToSignedWord(SRC[127-120]*DEST[127-120]+ SRC[119-
    112]* DEST[119-112]);
```

Intel C/C++ Compiler Intrinsic Equivalents

PMADDUBSW	__m64 _mm_maddubs_pi16 (__m64 a, __m64 b)
PMADDUBSW	__m128i _mm_maddubs_epi16 (__m128i a, __m128i b)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
	(128-bit operations only) If not aligned on 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If CR0.EM = 1.
	If CR4.OSFXSR(bit 9) = 0 (128-bit operations only)
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#AC(0)	(64-bit operations only) If alignment checking is enabled and unaligned memory reference is made while the current privilege level is 3.

Real 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 not aligned on 16-byte boundary, regardless of segment.
#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#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)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and 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.
	If CPUID.01H:ECX.SSSE3[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.

PMADDWD—Multiply and Add Packed Integers

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

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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

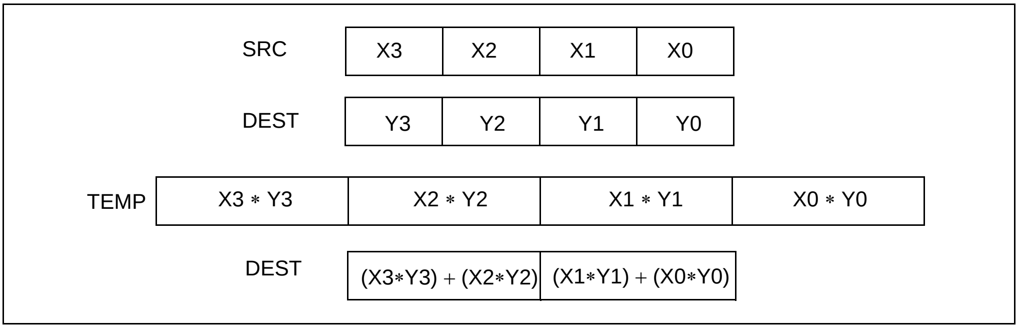


Figure 4-2. PMADDWD Execution Model Using 64-bit Operands

Operation

```
PMADDWD instruction with 64-bit operands:
    DEST[31:0] ← (DEST[15:0] ∗ SRC[15:0]) + (DEST[31:16] ∗ SRC[31:16]);
    DEST[63:32] ← (DEST[47:32] ∗ SRC[47:32]) + (DEST[63:48] ∗ SRC[63:48]);

PMADDWD instruction with 128-bit operands:
    DEST[31:0] ← (DEST[15:0] ∗ SRC[15:0]) + (DEST[31:16] ∗ SRC[31:16]);
    DEST[63:32] ← (DEST[47:32] ∗ SRC[47:32]) + (DEST[63:48] ∗ SRC[63:48]);
    DEST[95:64] ← (DEST[79:64] ∗ SRC[79:64]) + (DEST[95:80] ∗ SRC[95:80]);
    DEST[127:96] ← (DEST[111:96] ∗ SRC[111:96]) + (DEST[127:112] ∗ SRC[127:112]);
```

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.

#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 *mm1, mm2/m64*	Valid	Valid	Compare signed word integers in *mm2/m64* and *mm1* and return maximum values.
66 0F EE /*r*	PMAXSW *xmm1, xmm2/m128*	Valid	Valid	Compare signed word integers in *xmm2/m128* and *xmm1* and return maximum values.

Description

Performs a 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 a 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.

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 *mm1, mm2/m64*	Valid	Valid	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns maximum values.
66 0F DE /*r*	PMAXUB *xmm1, xmm2/m128*	Valid	Valid	Compare unsigned byte integers in *xmm2/m128* and *xmm1* and returns maximum values.

Description

Performs a 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 a 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;
```

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 *mm1, mm2/m64*	Valid	Valid	Compare signed word integers in *mm2/m64* and *mm1* and return minimum values.
66 0F EA /*r*	PMINSW *xmm1, xmm2/m128*	Valid	Valid	Compare signed word integers in *xmm2/m128* and *xmm1* and return minimum values.

Description

Performs a 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 a 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.

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 *mm1, mm2/m64*	Valid	Valid	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns minimum values.
66 0F DA /*r*	PMINUB *xmm1, xmm2/m128*	Valid	Valid	Compare unsigned byte integers in *xmm2/m128* and *xmm1* and returns minimum values.

Description

Performs a 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 a 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;
```

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 *r32, mm*	Valid	Valid	Move a byte mask of *mm* to *r32*.
REX.W + 0F D7 /r	PMOVMSKB *r64, mm*	Valid	N.E.	Move a byte mask of mm to the lower 32-bits of r64 and zero-fill the upper 32-bits.
66 0F D7 /r	PMOVMSKB *r32, xmm*	Valid	Valid	Move a byte mask of *xmm* to *r32*.
REX.W + 66 0F D7 /r	PMOVMSKB *r64, xmm*	Valid	N.E.	Move a byte mask of xmm to the lower 32-bits of r64 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 a 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.

PMULHRSW — Packed Multiply High with Round and Scale

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 38 0B /r	PMULHRSW mm1, mm2/m64	Valid	Valid	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to MM1.
66 0F 38 0B /r	PMULHRSW xmm1, xmm2/m128	Valid	Valid	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to XMM1.

Description

PMULHRSW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Operation

```
PMULHRSW wi64-bit operands:
    temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
    temp1[31:0] = INT32 ((DEST[31:15] * SRC[31:15]) >>14) + 1;
    temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >> 14) + 1;
    temp3[31:0] = INT32 ((DEST[63:48] * SRc[63:48]) >> 14) + 1;
    DEST[15:0] = temp0[16:1];
    DEST[31:15] = temp1[16:1];
    DEST[47:32] = temp2[16:1];
    DEST[63:48] = temp3[16:1];

PMULHRSW with 128-bit operand:
    temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
    temp1[31:0] = INT32 ((DEST[31:15] * SRC[31:15]) >>14) + 1;
    temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
    temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
    temp4[31:0] = INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1;
```

```
temp5[31:0] = INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1;
temp6[31:0] = INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1;
temp7[31:0] = INT32 ((DEST[127:112] * SRC[127:112) >>14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:15] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
DEST[79:64] = temp4[16:1];
DEST[95:80] = temp5[16:1];
DEST[111:96] = temp6[16:1];
DEST[127:112] = temp7[16:1];
```

Intel C/C++ Compiler Intrinsic Equivalents

PMULHRSW	__m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
PMULHRSW	__m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
	(128-bit operations only) If not aligned on 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#AC(0)	(64-bit operations only) If alignment checking is enabled and unaligned memory reference is made while the current privilege level is 3.

Real 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 not aligned on 16-byte boundary, regardless of segment.
#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.

#NM	If TS bit in CR0 is set.
#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)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and 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.
	If CPUID.01H:ECX.SSSE3[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.

PMULHUW—Multiply Packed Unsigned Integers and Store High Result

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

Description

Performs a 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.

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

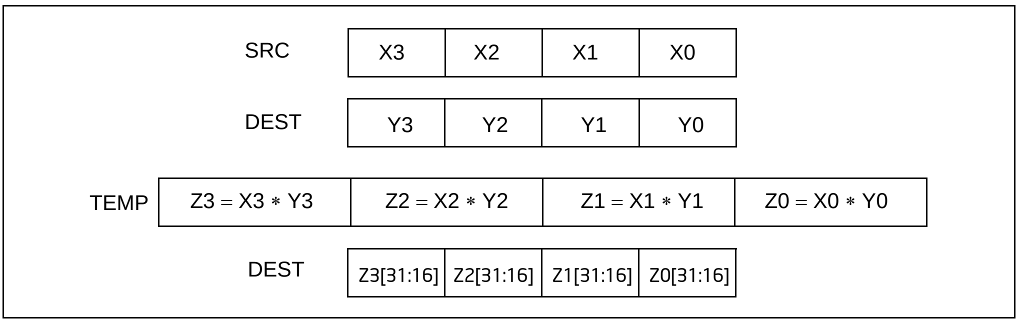


Figure 4-3. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

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.
#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 *mm, mm/m64*	Valid	Valid	Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*.
66 0F E5 /*r*	PMULHW *xmm1, xmm2/m128*	Valid	Valid	Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*.

Description

Performs a 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 a 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.

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 *mm, mm/m64*	Valid	Valid	Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the low 16 bits of the results in *mm1*.
66 0F D5 /*r*	PMULLW *xmm1, xmm2/m128*	Valid	Valid	Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the low 16 bits of the results in *xmm1*.

Description

Performs a 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.

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

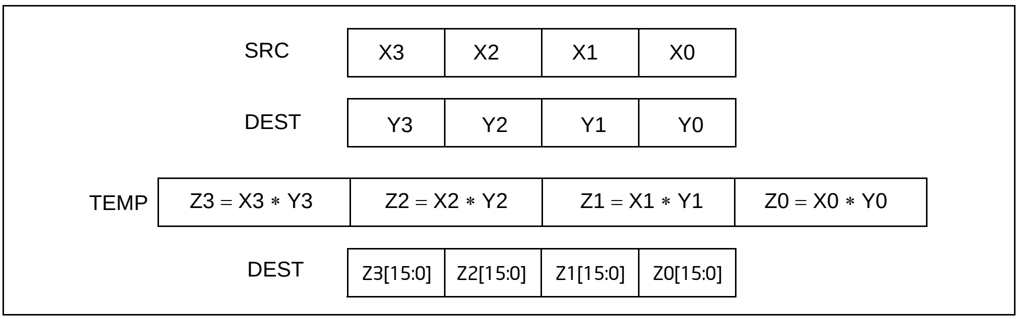


Figure 4-4. PMULLU Instruction Operation Using 64-bit Operands

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 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 *mm1, mm2/m64*	Valid	Valid	Multiply unsigned doubleword integer in *mm1* by unsigned doubleword integer in *mm2/m64*, and store the quadword result in *mm1*.
66 0F F4 /r	PMULUDQ *xmm1, xmm2/m128*	Valid	Valid	Multiply packed unsigned doubleword integers in *xmm1* by packed unsigned doubleword integers in *xmm2/m128*, and store the quadword results in *xmm1*.

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.

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

Operation

```
PMULUDQ instruction with 64-Bit operands:
    DEST[63:0] ← DEST[31:0] ∗ SRC[31:0];

PMULUDQ instruction with 128-Bit operands:
    DEST[63:0] ← DEST[31:0] ∗ SRC[31:0];
    DEST[127:64] ← DEST[95:64] ∗ 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.

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.

POP—Pop a Value from the Stack

Opcode	Instructio n	64-Bit Mode	Compat/ Leg Mode	Description
8F /0	POP r/*m16*	Valid	Valid	Pop top of stack into *m16*; increment stack pointer.
8F /0	POP r/*m32*	N.E.	Valid	Pop top of stack into *m32*; increment stack pointer.
8F /0	POP r/*m64*	Valid	N.E.	Pop top of stack into *m64*; increment stack pointer. Cannot encode 32-bit operand size.
58+ *rw*	POP *r16*	Valid	Valid	Pop top of stack into *r16*; increment stack pointer.
58+ *rd*	POP *r32*	N.E.	Valid	Pop top of stack into *r32*; increment stack pointer.
58+ *rd*	POP *r64*	Valid	N.E.	Pop top of stack into *r64*; 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, 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[1]. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

---

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

In 64-bit mode, using a 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.

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;
```

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);
            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)	If attempt is made to load SS register with NULL segment selector.
	If the destination 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 is used to access memory and it contains a NULL segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits.
	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.
	If the SS register is being loaded and the segment pointed to is a non-writable data segment.

	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	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.
#SS(0)	If the current top of stack is not within the stack segment.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#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 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 PUSHA/PUSHAD 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();
        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/POPFQ—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	POPFQ	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 RF[1], 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 POPFQ (note that the 32-bit operand is not encodable). POPFQ pops 64

1. RF is always zero after execution of POPF. This is because POPF, like all instructions, clears RF as it begins to execute.

bits from the stack, loads the lower 32 bits into RFLAGS, and zero extends the upper bits of RFLAGS.

See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 1*, for more information about the EFLAGS registers.

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 RF, VIP, VIF, and VM can be modified;
                    VIP and VIF are cleared; RF, VM, and all reserved bits are unaffected. *)
                ELSE IF (Operandsize = 64)
                    RFLAGS = Pop(); (* 64-bit pop *)
                    (* All non-reserved flags except RF, VIP, VIF, and VM can be modified; VIP
                    and VIF are cleared; RF, 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, RF, VIP, and
                            VIF can be modified; IF, IOPL, RF, VM, and all reserved
                            bits are unaffected; VIP and VIF are cleared. *)
                        ELSE
                            EFLAGS ← Pop(); (* 32-bit pop *)
                            (* All non-reserved bits except IOPL, RF, VIP, and VIF can be
                             modified; IOPL, RF, 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, RF, VIP, and
                            VIF can be modified; IF, IOPL, RF, VM, and all reserved
                            bits are unaffected; VIP and VIF are cleared. *)
                        ELSE
```

```
                        RFLAGS ← Pop(); (* 64-bit pop *)
                        (* All non-reserved bits except IOPL, RF, VIP, and VIF can be
                        modified; IOPL, RF, 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;
        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 may be affected; see the Operation section for details.

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/POPFD 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.

64-Bit Mode Exceptions

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

POR—Bitwise Logical OR

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

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 a 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.

#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 PREFETCH*h* 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 PREFETCH*h* 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 PREFETCH*h* instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCH*h* instruction is also unordered with respect to CLFLUSH instructions, other PREFETCH*h* 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 *mm1, mm2/m64*	Valid	Valid	Computes the absolute differences of the packed unsigned byte integers from *mm2 /m64* and *mm1*; differences are then summed to produce an unsigned word integer result.
66 0F F6 /*r*	PSADBW *xmm1, xmm2/m128*	Valid	Valid	Computes the absolute differences of the packed unsigned byte integers from *xmm2 /m128* and *xmm1*; 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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

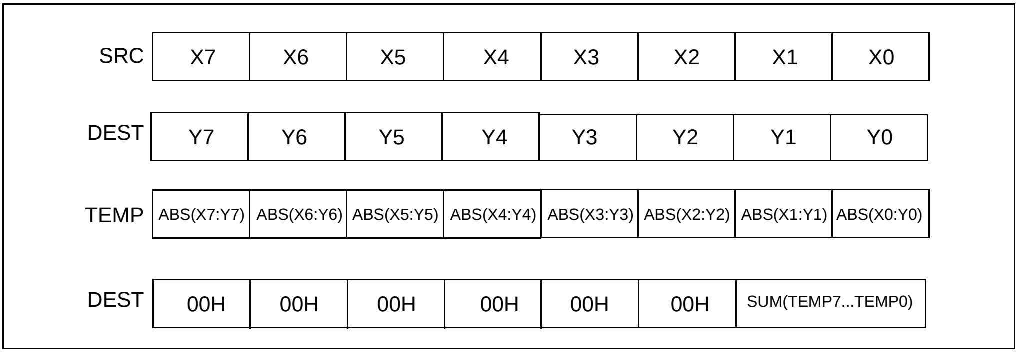


Figure 4-5. PSADBW Instruction Operation Using 64-bit Operands

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.

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.

PSHUFB — Packed Shuffle Bytes

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 38 00 /r	PSHUFB mm1, mm2/m64	Valid	Valid	Shuffle bytes in mm1 according to contents of mm2/m64.
66 0F 38 00 /r	PSHUFB xmm1, xmm2/m128	Valid	Valid	Shuffle bytes in xmm1 according to contents of xmm2/m128.

Description

PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Operation

```
PSHUFB with 64 bit operands:
    for i = 0 to 7 {
        if (SRC[(i * 8)+7] == 1 ) then
            DEST[(i*8)+7...(i*8)+0] <- 0;
        else
            index[2..0] <- SRC[(i*8)+2 .. (i*8)+0];
            DEST[(i*8)+7...(i*8)+0] <- DEST[(index*8+7)..(index*8+0)];
        endif;
    }

PSHUFB with 128 bit operands:
    for i = 0 to 15 {
        if (SRC[(i * 8)+7] == 1 ) then
            DEST[(i*8)+7..(i*8)+0] <- 0;
         else
            index[3..0] <- SRC[(i*8)+3 .. (i*8)+0];
            DEST[(i*8)+7..(i*8)+0] <- DEST[(index*8+7)..(index*8+0)];
```

```
        endif
    }
```

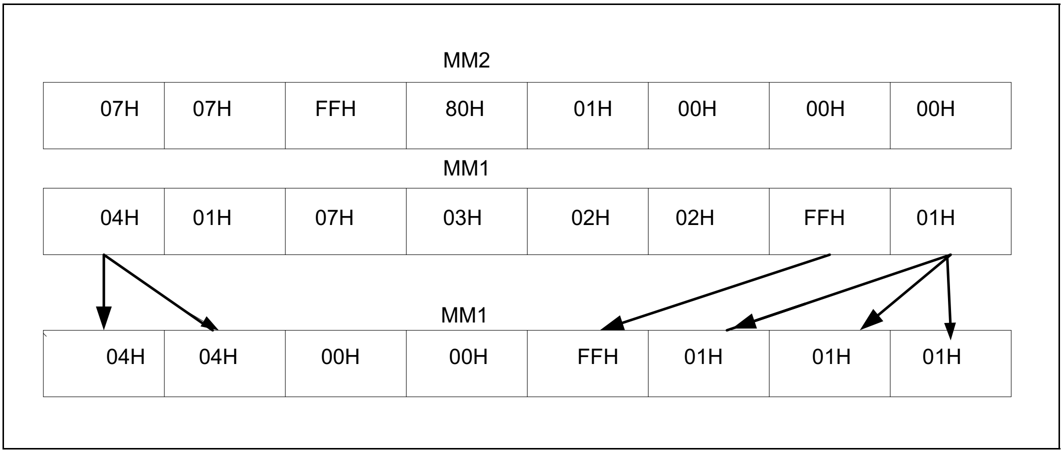


Figure 4-6. PSHUB with 64-Bit Operands

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFB __m64 _mm_shuffle_pi8 (__m64 a, __m64 b)

PSHUFB __m128i _mm_shufflehi_epi16(__m128i a, int n)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
	(128-bit operations only) If not aligned on 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#AC(0)	(64-bit operations only) If alignment checking is enabled and unaligned memory reference is made while the current privilege level is 3.

Real 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 not aligned on 16-byte boundary, regardless of segment.
#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#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)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and 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.
	If CPUID.01H:ECX.SSSE3[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.

PSHUFD—Shuffle Packed Doublewords

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

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 4-7 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 4-7) determines which doubleword from the source operand will be copied to doubleword 0 of the destination operand.

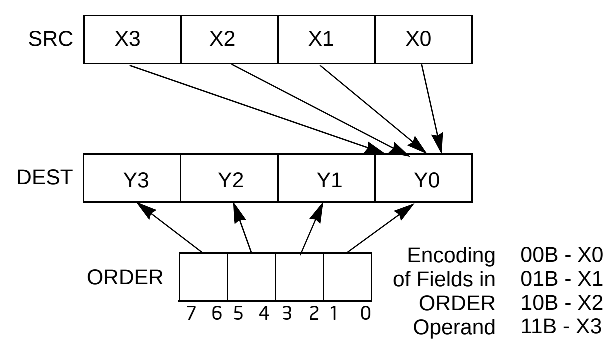


Figure 4-7. PSHUFD Instruction Operation

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 a 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.

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 *xmm1, xmm2/ m128, imm8*	Valid	Valid	Shuffle the high words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.

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 4-7. 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 a 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.

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 *xmm1*, *xmm2/m128*, *imm8*	Valid	Valid	Shuffle the low words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.

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 4-7. 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 a 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.

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 *mm1, mm2/m64, imm8*	Valid	Valid	Shuffle the words in *mm2/m64* based on the encoding in *imm8* and store the result in *mm1*.

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 4-7. 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 a 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.

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.

PSIGNB/PSIGNW/PSIGND — Packed SIGN

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 38 08 /r	PSIGNB mm1, mm2/m64	Valid	Valid	Negate packed byte integers in mm1 if the corresponding sign in mm2/m64 is less than zero.
66 0F 38 08 /r	PSIGNB xmm1, xmm2/m128	Valid	Valid	Negate packed byte integers in xmm1 if the corresponding sign in xmm2/m128 is less than zero.
0F 38 09 /r	PSIGNW mm1, mm2/m64	Valid	Valid	Negate packed 16-bit integers in mm1 if the corresponding sign in mm2/m64 is less than zero.
66 0F 38 09 /r	PSIGNW xmm1, xmm2/m128	Valid	Valid	Negate packed 16-bit integers in xmm1 if the corresponding sign in xmm2/m128 is less than zero.
0F 38 0A /r	PSIGND mm1, mm2/m64	Valid	Valid	Negate packed doubleword integers in mm1 if the corresponding sign in mm2/m64 is less than zero.
66 0F 38 0A /r	PSIGND xmm1, xmm2/m128	Valid	Valid	Negate packed doubleword integers in xmm1 if the corresponding sign in xmm2/m128 is less than zero.

Description

PSIGNB/PSIGNW/PSIGND negates each data element of the destination operand (the first operand) if the sign of the corresponding data element in the source operand (the second operand) is less than zero. If the sign of a data element in the source operand is positive, the corresponding data element in the destination operand is unchanged. If a data element in the source operand is zero, the corresponding data element in the destination operand is set to zero.

PSIGNB operates on signed bytes. PSIGNW operates on 16-bit signed words. PSIGND operates on signed 32-bit integers. Both operands can be MMX register or XMM registers. When the source operand is a 128bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

Operation

```
PSIGNB with 64 bit operands:
    if (SRC[7..0] < 0 )
        DEST[7...0] <- Neg(DEST[7...0])
    else if(SRC[7..0] == 0 )
```

```
        DEST[7...0] <- 0
    else if(SRC[7..0] > 0 )
        DEST[7...0] <- DEST[7...0]
    Repeat operation for 2nd through 7th bytes

    if (SRC[63..56] < 0 )
        DEST[63...56] <- Neg(DEST[63...56])
    else if(SRC[63.. 56] == 0 )
        DEST[63...56] <- 0
    else if(SRC[63.. 56] > 0 )
        DEST[63...56] <- DEST[63...56]

PSIGNB with 128 bit operands:
    if (SRC[7..0] < 0 )
        DEST[7...0] <- Neg(DEST[7...0])
    else if(SRC[7..0] == 0 )
        DEST[7...0] <- 0
    else if(SRC[7..0] > 0 )
        DEST[7...0] <- DEST[7...0]
    Repeat operation for 2nd through 15th bytes
    if (SRC[127..120] < 0 )
        DEST[127...120] <- Neg(DEST[127...120])
    else if(SRC[127.. 120] == 0 )
        DEST[127...120] <- 0
    else if(SRC[127.. 120] > 0 )
        DEST[127...120] <- DEST[127...120]

PSIGNW with 64 bit operands:
    if (SRC[15..0] < 0 )
        DEST[15...0] <- Neg(DEST[15...0])
    else if(SRC[15..0] == 0 )
        DEST[15...0] <- 0
    else if(SRC[15..0] > 0 )
        DEST[15...0] <- DEST[15...0]
Repeat operation for 2nd through 3rd words
    if (SRC[63..48] < 0 )
        DEST[63...48] <- Neg(DEST[63...48])
    else if(SRC[63..48] == 0 )
        DEST[63...48] <- 0
    else if(SRC[63..48] > 0 )
        DEST[63...48] <- DEST[63...48]

PSIGNW with 128 bit operands:
    if (SRC[15..0] < 0 )
```

```
        DEST[15...0] <- Neg(DEST[15...0])
    else if(SRC[15..0] == 0 )
        DEST[15...0] <- 0
    else if(SRC[15..0] > 0 )
        DEST[15...0] <- DEST[15...0]
    Repeat operation for 2nd through 7th words
    if (SRC[127..112] < 0 )
        DEST[127...112] <- Neg(DEST[127...112])
    else if(SRC[127.. 112] == 0 )
        DEST[127...112] <- 0
    else if(SRC[127.. 112] > 0 )
        DEST[127...112] <- DEST[127...112]

PSIGND with 64 bit operands:
    if (SRC[31..0] < 0 )
        DEST[31...0] <- Neg(DEST[31...0])
    else if(SRC[31..0] == 0 )
        DEST[31...0] <- 0
    else if(SRC[31..0] > 0 )
        DEST[31...0] <- DEST[31...0]
    if (SRC[63..32] < 0 )
        DEST[63...32] <- Neg(DEST[63...32])
    else if(SRC[63.. 32] == 0 )
        DEST[63...32] <- 0
    else if(SRC[63.. 32] > 0 )
        DEST[63...32] <- DEST[63...32]

PSIGND with 128 bit operands:
    if (SRC[31..0] < 0 )
        DEST[31...0] <- Neg(DEST[31...0])
    else if(SRC[31..0] == 0 )
        DEST[31...0] <- 0
    else if(SRC[31..0] > 0 )
        DEST[31...0] <- DEST[31...0]
    Repeat operation for 2nd through 3rd double words
    if (SRC[127..96] < 0 )
        DEST[127...96] <- Neg(DEST[127...96])
    else if(SRC[127.. 96] == 0 )
        DEST[127...96] <- 0
    else if(SRC[127.. 96] > 0 )
        DEST[127...96] <- DEST[127...96]
```

Intel C/C++ Compiler Intrinsic Equivalent

PSIGNB   __m64 _mm_sign_pi8 (__m64 a, __m64 b)

PSIGNB	__m128i _mm_sign_epi8 (__m128i a, __m128i b)
PSIGNW	__m64 _mm_sign_pi16 (__m64 a, __m64 b)
PSIGNW	__m128i _mm_sign_epi16 (__m128i a, __m128i b)
PSIGND	__m64 _mm_sign_pi32 (__m64 a, __m64 b)
PSIGND	__m128i _mm_sign_epi32 (__m128i a, __m128i b)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
	(128-bit operations only) If not aligned on 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If CR0.EM = 1.
	(128-bit operations only) If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#AC(0)	(64-bit operations only) If alignment checking is enabled and unaligned memory reference is made while the current privilege level is 3.

Real 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 not aligned on 16-byte boundary, regardless of segment.
#UD	(128-bit operations only) If CR0.EM = 1. If CR4.OSFXSR(bit 9) = 0.
	If CPUID.SSSE3(ECX bit 9) = 0.
#NM	If TS bit in CR0 is set.
#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)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and 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.
	If CPUID.01H:ECX.SSSE3[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.

PSLLDQ—Shift Double Quadword Left Logical

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 73 /7 ib	PSLLDQ *xmm1*, *imm8*	Valid	Valid	Shift *xmm1* left by *imm8* 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 *mm, mm/m64*	Valid	Valid	Shift words in *mm* left *mm/m64* while shifting in 0s.
66 0F F1 /*r*	PSLLW *xmm1, xmm2/m128*	Valid	Valid	Shift words in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 71 /6 ib	PSLLW *xmm1, imm8*	Valid	Valid	Shift words in *mm* left by *imm8* while shifting in 0s.
66 0F 71 /6 ib	PSLLW *xmm1, imm8*	Valid	Valid	Shift words in *xmm1* left by *imm8* while shifting in 0s.
0F F2 /*r*	PSLLD *mm, mm/m64*	Valid	Valid	Shift doublewords in *mm* left by *mm/m64* while shifting in 0s.
66 0F F2 /*r*	PSLLD *xmm1, xmm2/m128*	Valid	Valid	Shift doublewords in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 72 /6 ib	PSLLD *mm, imm8*	Valid	Valid	Shift doublewords in *mm* left by *imm8* while shifting in 0s.
66 0F 72 /6 ib	PSLLD *xmm1, imm8*	Valid	Valid	Shift doublewords in *xmm1* left by *imm8* while shifting in 0s.
0F F3 /*r*	PSLLQ *mm, mm/m64*	Valid	Valid	Shift quadword in *mm* left by *mm/m64* while shifting in 0s.
66 0F F3 /*r*	PSLLQ *xmm1, xmm2/m128*	Valid	Valid	Shift quadwords in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 73 /6 ib	PSLLQ *mm, imm8*	Valid	Valid	Shift quadword in *mm* left by *imm8* while shifting in 0s.
66 0F 73 /6 ib	PSLLQ *xmm1, imm8*	Valid	Valid	Shift quadwords in *xmm1* left by *imm8* 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 4-8 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.

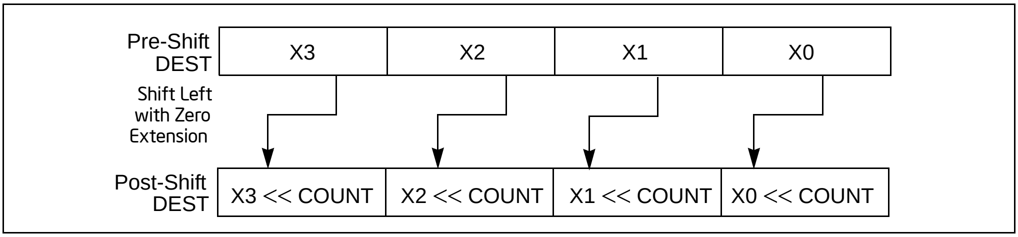


Figure 4-8. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

The PSLLW 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 PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

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

Operation

```
PSLLW 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;

PSLLQ instruction with 64-bit operand:
    IF (COUNT > 63)
```

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

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.

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 *mm, mm/m64*	Valid	Valid	Shift words in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E1 /*r*	PSRAW *xmm1, xmm2/m128*	Valid	Valid	Shift words in *xmm1* right by *xmm2/m128* while shifting in sign bits.
0F 71 /4 ib	PSRAW *mm, imm8*	Valid	Valid	Shift words in *mm* right by *imm8* while shifting in sign bits
66 0F 71 /4 ib	PSRAW *xmm1*, imm8	Valid	Valid	Shift words in *xmm1* right by imm8 while shifting in sign bits
0F E2 /*r*	PSRAD *mm, mm/m64*	Valid	Valid	Shift doublewords in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E2 /*r*	PSRAD *xmm1, xmm2/m128*	Valid	Valid	Shift doubleword in *xmm1* right by *xmm2 /m128* while shifting in sign bits.
0F 72 /4 ib	PSRAD *mm, imm8*	Valid	Valid	Shift doublewords in *mm* right by *imm8* while shifting in sign bits.
66 0F 72 /4 ib	PSRAD *xmm1*, imm8	Valid	Valid	Shift doublewords in *xmm1* right by *imm8* 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 4-9 gives an example of shifting words in a 64-bit operand.)

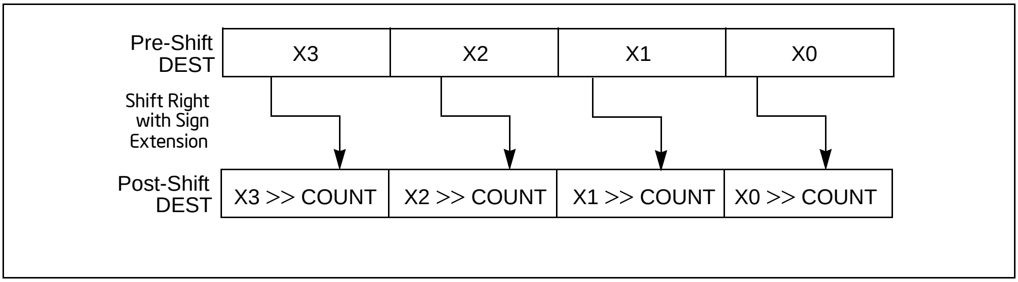


Figure 4-9. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

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.

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 a 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	__m64 _mm_srai_pi16 (__m64 m, int count)
PSRAW	__m64 _mm_sraw_pi16 (__m64 m, __m64 count)
PSRAD	__m64 _mm_srai_pi32 (__m64 m, int count)
PSRAD	__m64 _mm_sra_pi32 (__m64 m, __m64 count)
PSRAW	__m128i _mm_srai_epi16(__m128i m, int count)
PSRAW	__m128i _mm_sra_epi16(__m128i m, __m128i count))
PSRAD	__m128i _mm_srai_epi32 (__m128i m, int count)
PSRAD	__m128i _mm_sra_epi32 (__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.

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 *xmm1, imm8*	Valid	Valid	Shift *xmm1* right by *imm8* 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 a 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.

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 *mm, mm/m64*	Valid	Valid	Shift words in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D1 /*r*	PSRLW *xmm1, xmm2/m128*	Valid	Valid	Shift words in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
0F 71 /2 ib	PSRLW *mm, imm8*	Valid	Valid	Shift words in *mm* right by *imm8* while shifting in 0s.
66 0F 71 /2 ib	PSRLW *xmm1, imm8*	Valid	Valid	Shift words in *xmm1* right by *imm8* while shifting in 0s.
0F D2 /*r*	PSRLD *mm, mm/m64*	Valid	Valid	Shift doublewords in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D2 /*r*	PSRLD *xmm1, xmm2/m128*	Valid	Valid	Shift doublewords in *xmm1* right by amount specified in *xmm2 /m128* while shifting in 0s.
0F 72 /2 ib	PSRLD *mm, imm8*	Valid	Valid	Shift doublewords in *mm* right by *imm8* while shifting in 0s.
66 0F 72 /2 ib	PSRLD *xmm1*, imm8	Valid	Valid	Shift doublewords in *xmm1* right by *imm8* while shifting in 0s.
0F D3 /*r*	PSRLQ *mm, mm/m64*	Valid	Valid	Shift *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D3 /*r*	PSRLQ *xmm1, xmm2/m128*	Valid	Valid	Shift quadwords in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
0F 73 /2 ib	PSRLQ *mm, imm8*	Valid	Valid	Shift *mm* right by *imm8* while shifting in 0s.
66 0F 73 /2 ib	PSRLQ *xmm1, imm8*	Valid	Valid	Shift quadwords in *xmm1* right by *imm8* 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 4-10 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.

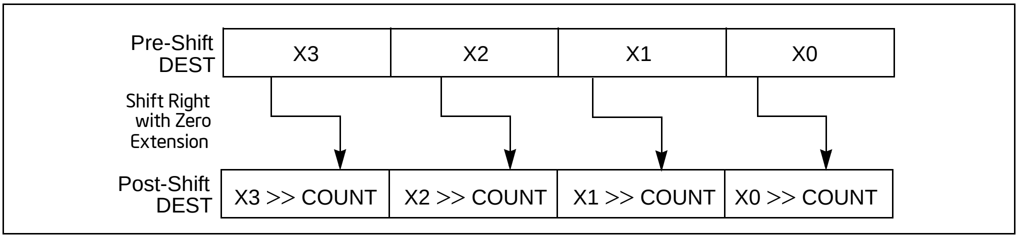


Figure 4-10. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

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 doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode, using a 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)
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 *mm, mm/m64*	Valid	Valid	Subtract packed byte integers in *mm/m64* from packed byte integers in *mm*.
66 0F F8 /*r*	PSUBB *xmm1, xmm2/m128*	Valid	Valid	Subtract packed byte integers in *xmm2/m128* from packed byte integers in *xmm1*.
0F F9 /*r*	PSUBW *mm, mm/m64*	Valid	Valid	Subtract packed word integers in *mm/m64* from packed word integers in *mm*.
66 0F F9 /*r*	PSUBW *xmm1, xmm2/m128*	Valid	Valid	Subtract packed word integers in *xmm2/m128* from packed word integers in *xmm1*.
0F FA /*r*	PSUBD *mm, mm/m64*	Valid	Valid	Subtract packed doubleword integers in *mm/m64* from packed doubleword integers in *mm*.
66 0F FA /*r*	PSUBD *xmm1, xmm2/m128*	Valid	Valid	Subtract packed doubleword integers in *xmm2/mem128* from packed doubleword integers in *xmm1*.

Description

Performs a 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a 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 a 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] ← DEST[7:0] − SRC[7:0];
    (* Repeat subtract operation for 2nd through 7th byte *)
    DEST[63:56] ← DEST[63:56] − SRC[63:56];

PSUBB instruction with 128-bit operands:
    DEST[7:0] ← DEST[7:0] − SRC[7:0];
    (* Repeat subtract operation for 2nd through 14th byte *)
    DEST[127:120] ← DEST[111:120] − SRC[127:120];

PSUBW instruction with 64-bit operands:
    DEST[15:0] ← DEST[15:0] − SRC[15:0];
    (* Repeat subtract operation for 2nd and 3rd word *)
    DEST[63:48] ← DEST[63:48] − SRC[63:48];

PSUBW instruction with 128-bit operands:
    DEST[15:0]  ← DEST[15:0] − SRC[15:0];
    (* Repeat subtract operation for 2nd through 7th word *)
    DEST[127:112] ← DEST[127:112] − SRC[127:112];

PSUBD instruction with 64-bit operands:
    DEST[31:0] ← DEST[31:0] − SRC[31:0];
    DEST[63:32] ← DEST[63:32] − SRC[63:32];

PSUBD instruction with 128-bit operands:
    DEST[31:0]  ← DEST[31:0] − SRC[31:0];
    (* Repeat subtract operation for 2nd and 3rd doubleword *)
    DEST[127:96] ← 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)

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 *mm1, mm2/m64*	Valid	Valid	Subtract quadword integer in *mm1* from *mm2 /m64*.
66 0F FB /*r*	PSUBQ *xmm1, xmm2/m128*	Valid	Valid	Subtract packed quadword integers in *xmm1* from *xmm2 /m128*.

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, a 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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
PSUBQ instruction with 64-Bit operands:
    DEST[63:0] ← DEST[63:0] − SRC[63:0];

PSUBQ instruction with 128-Bit operands:
    DEST[63:0] ← DEST[63:0] − SRC[63:0];
    DEST[127:64] ← DEST[127:64] − SRC[127:64];
```

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.

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 *mm, mm/m64*	Valid	Valid	Subtract signed packed bytes in *mm/m64* from signed packed bytes in *mm* and saturate results.
66 0F E8 /*r*	PSUBSB *xmm1, xmm2/m128*	Valid	Valid	Subtract packed signed byte integers in *xmm2/m128* from packed signed byte integers in *xmm1* and saturate results.
0F E9 /*r*	PSUBSW *mm, mm/m64*	Valid	Valid	Subtract signed packed words in *mm/m64* from signed packed words in *mm* and saturate results.
66 0F E9 /*r*	PSUBSW *xmm1, xmm2/m128*	Valid	Valid	Subtract packed signed word integers in *xmm2/m128* from packed signed word integers in *xmm1* 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a 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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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.

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 *mm, mm/m64*	Valid	Valid	Subtract unsigned packed bytes in *mm/m64* from unsigned packed bytes in *mm* and saturate result.
66 0F D8 /*r*	PSUBUSB *xmm1, xmm2/m128*	Valid	Valid	Subtract packed unsigned byte integers in *xmm2/m128* from packed unsigned byte integers in xmm1 and saturate result.
0F D9 /*r*	PSUBUSW *mm, mm/m64*	Valid	Valid	Subtract unsigned packed words in *mm/m64* from unsigned packed words in *mm* and saturate result.
66 0F D9 /*r*	PSUBUSW *xmm1, xmm2/m128*	Valid	Valid	Subtract packed unsigned word integers in *xmm2/m128* from packed unsigned word integers in xmm1 and saturate result.

Description

Performs a 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a 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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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.

#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 *mm, mm/m64*	Valid	Valid	Unpack and interleave high-order bytes from *mm* and *mm/m64* into *mm*.
66 0F 68 /*r*	PUNPCKHBW *xmm1, xmm2/m128*	Valid	Valid	Unpack and interleave high-order bytes from *xmm1* and *xmm2/m128* into *xmm1*.
0F 69 /*r*	PUNPCKHWD *mm, mm/m64*	Valid	Valid	Unpack and interleave high-order words from *mm* and *mm/m64* into *mm*.
66 0F 69 /*r*	PUNPCKHWD *xmm1, xmm2/m128*	Valid	Valid	Unpack and interleave high-order words from *xmm1* and *xmm2/m128* into *xmm1*.
0F 6A /*r*	PUNPCKHDQ *mm, mm/m64*	Valid	Valid	Unpack and interleave high-order doublewords from *mm* and *mm/m64* into *mm*.
66 0F 6A /*r*	PUNPCKHDQ *xmm1, xmm2/m128*	Valid	Valid	Unpack and interleave high-order doublewords from *xmm1* and *xmm2/m128* into *xmm1*.
66 0F 6D /*r*	PUNPCKHQDQ *xmm1, xmm2/m128*	Valid	Valid	Unpack and interleave high-order quadwords from *xmm1* and *xmm2/m128* into *xmm1*.

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 4-11 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

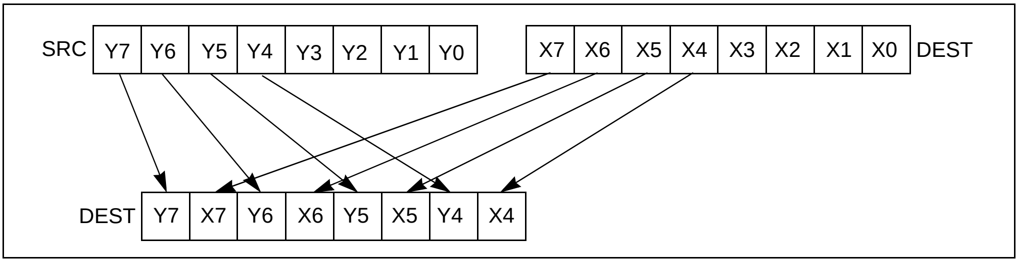


Figure 4-11. PUNPCKHBW Instruction Operation Using 64-bit Operands

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. 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 a 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]  ← DEST[71:64];
    DEST[15:8]  ← SRC[71:64];
    DEST[23:16] ← DEST[79:72];
    DEST[31:24] ← SRC[79:72];
    DEST[39:32] ← DEST[87:80];
    DEST[47:40] ← SRC[87:80];
    DEST[55:48] ← DEST[95:88];
    DEST[63:56] ← SRC[95:88];
    DEST[71:64] ← DEST[103:96];
    DEST[79:72] ← SRC[103:96];
    DEST[87:80] ← DEST[111:104];
    DEST[95:88] ← SRC[111:104];
    DEST[103:96]  ← DEST[119:112];
    DEST[111:104] ← SRC[119:112];
    DEST[119:112] ← DEST[127:120];
    DEST[127:120] ← SRC[127:120];

PUNPCKHWD instruction with 128-bit operands:
    DEST[15:0]  ← DEST[79:64];
    DEST[31:16] ← SRC[79:64];
    DEST[47:32] ← DEST[95:80];
    DEST[63:48] ← SRC[95:80];
    DEST[79:64] ← DEST[111:96];
    DEST[95:80] ← SRC[111:96];
    DEST[111:96]  ← DEST[127:112];
    DEST[127:112] ← SRC[127:112];

PUNPCKHDQ instruction with 128-bit operands:
    DEST[31:0] ← DEST[95:64];
    DEST[63:32]  ← SRC[95:64];
    DEST[95:64]  ← DEST[127:96];
    DEST[127:96] ← SRC[127:96];

PUNPCKHQDQ instruction:
    DEST[63:0] ← DEST[127:64];
    DEST[127:64] ← SRC[127:64];
```

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 *mm, mm/m32*	Valid	Valid	Interleave low-order bytes from *mm* and *mm/m32* into *mm.*
66 0F 60 /*r*	PUNPCKLBW *xmm1, xmm2/m128*	Valid	Valid	Interleave low-order bytes from *xmm1* and *xmm2/m128* into *xmm1*.
0F 61 /*r*	PUNPCKLWD *mm, mm/m32*	Valid	Valid	Interleave low-order words from *mm* and *mm/m32* into *mm.*
66 0F 61 /*r*	PUNPCKLWD *xmm1, xmm2/m128*	Valid	Valid	Interleave low-order words from *xmm1* and *xmm2/m128* into *xmm1*.
0F 62 /*r*	PUNPCKLDQ *mm, mm/m32*	Valid	Valid	Interleave low-order doublewords from *mm* and *mm/m32* into *mm.*
66 0F 62 /*r*	PUNPCKLDQ *xmm1, xmm2/m128*	Valid	Valid	Interleave low-order doublewords from *xmm1* and *xmm2/m128* into xmm1.
66 0F 6C /*r*	PUNPCKLQDQ *xmm1, xmm2/m128*	Valid	Valid	Interleave low-order quadword from *xmm1* and *xmm2/m128* into *xmm1* 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 4-12 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

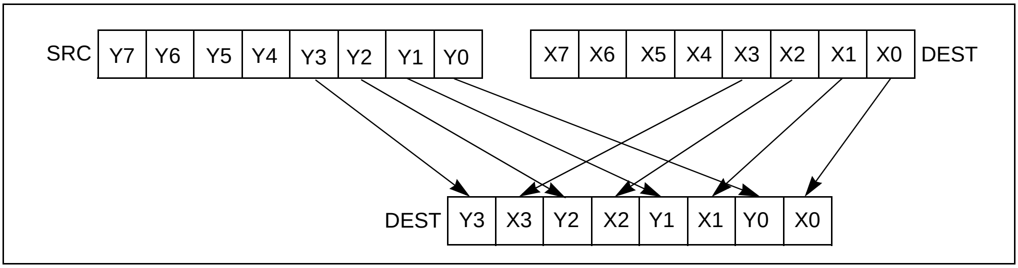


Figure 4-12. PUNPCKLBW Instruction Operation Using 64-bit Operands

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 a 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];
    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.

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 *r/m16*	Valid	Valid	Push *r/m16.*
FF /6	PUSH *r/m32*	N.E.	Valid	Push *r/m32.*
FF /6	PUSH *r/m64*	Valid	N.E.	Push *r/m64.* Default operand size 64-bits.
50+*rw*	PUSH *r16*	Valid	Valid	Push *r16.*
50+*rd*	PUSH *r32*	N.E.	Valid	Push *r32.*
50+*rd*	PUSH *r64*	Valid	N.E.	Push *r64.* Default operand size 64-bits.
6A	PUSH *imm8*	Valid	Valid	Push sign-extended *imm8. Stack pointer is incremented by the size of stack pointer.*
68	PUSH *imm16*	Valid	Valid	Push sign-extended *imm16. Stack pointer is incremented by the size of stack pointer.*
68	PUSH *imm32*	Valid	Valid	Push sign-extended *imm32. Stack pointer is incremented by the size of stack pointer.*
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, an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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 for Intel 64 architecture, real-address and virtual-8086 modes of IA-32 architecture.) 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;
```

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, 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 PUSHA/PUSHAD executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

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);
        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/POPFD 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 *Intel® 64 and IA-32 Architectures 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 pushs 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 when PUSHF/PUSHFD instruction executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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 *mm, mm/m64*	Valid	Valid	Bitwise XOR of *mm/m64* and *mm*.
66 0F EF /*r*	PXOR *xmm1, xmm2/m128*	Valid	Valid	Bitwise XOR of *xmm2/m128* and *xmm1*.

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 a 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.

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 *r/m8*, 1	Valid	Valid	Rotate 9 bits (CF, *r/m8*) left once.
REX + D0 /2	RCL *r/m8**, 1	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) left once.
D2 /2	RCL *r/m8*, CL	Valid	Valid	Rotate 9 bits (CF, *r/m8*) left CL times.
REX + D2 /2	RCL *r/m8**, CL	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) left CL times.
C0 /2 *ib*	RCL *r/m8, imm8*	Valid	Valid	Rotate 9 bits (CF, *r/m8*) left *imm8* times.
REX + C0 /2 *ib*	RCL *r/m8*, imm8*	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) left *imm8* times.
D1 /2	RCL *r/m16*, 1	Valid	Valid	Rotate 17 bits (CF, *r/m16*) left once.
D3 /2	RCL *r/m16*, CL	Valid	Valid	Rotate 17 bits (CF, *r/m16*) left CL times.
C1 /2 *ib*	RCL *r/m16, imm8*	Valid	Valid	Rotate 17 bits (CF, *r/m16*) left *imm8* times.
D1 /2	RCL *r/m32*, 1	Valid	Valid	Rotate 33 bits (CF, *r/m32*) left once.
REX.W + D1 /2	RCL *r/m64*, 1	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) left once. Uses a 6 bit count.
D3 /2	RCL *r/m32*, CL	Valid	Valid	Rotate 33 bits (CF, *r/m32*) left CL times.
REX.W + D3 /2	RCL *r/m64*, CL	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) left CL times. Uses a 6 bit count.
C1 /2 *ib*	RCL *r/m32, imm8*	Valid	Valid	Rotate 33 bits (CF, *r/m32*) left *imm8* times.
REX.W + C1 /2 *ib*	RCL *r/m64, imm8*	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) left *imm8* times. Uses a 6 bit count.
D0 /3	RCR *r/m8*, 1	Valid	Valid	Rotate 9 bits (CF, *r/m8*) right once.
REX + D0 /3	RCR *r/m8**, 1	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) right once.
D2 /3	RCR *r/m8*, CL	Valid	Valid	Rotate 9 bits (CF, *r/m8*) right CL times.
REX + D2 /3	RCR *r/m8**, CL	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) right CL times.

Opcode**	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
C0 /3 *ib*	RCR *r/m8, imm8*	Valid	Valid	Rotate 9 bits (CF, *r/m8*) right *imm8* times.
REX + C0 /3 *ib*	RCR *r/m8*, imm8*	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) right *imm8* times.
D1 /3	RCR *r/m16*, 1	Valid	Valid	Rotate 17 bits (CF, *r/m16*) right once.
D3 /3	RCR *r/m16*, CL	Valid	Valid	Rotate 17 bits (CF, *r/m16*) right CL times.
C1 /3 *ib*	RCR *r/m16, imm8*	Valid	Valid	Rotate 17 bits (CF, *r/m16*) right *imm8* times.
D1 /3	RCR *r/m32*, 1	Valid	Valid	Rotate 33 bits (CF, *r/m32*) right once. Uses a 6 bit count.
REX.W + D1 /3	RCR *r/m64*, 1	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) right once. Uses a 6 bit count.
D3 /3	RCR *r/m32*, CL	Valid	Valid	Rotate 33 bits (CF, *r/m32*) right CL times.
REX.W + D3 /3	RCR *r/m64*, CL	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) right CL times. Uses a 6 bit count.
C1 /3 *ib*	RCR *r/m32, imm8*	Valid	Valid	Rotate 33 bits (CF, *r/m32*) right *imm8* times.
REX.W + C1 /3 *ib*	RCR *r/m64, imm8*	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) right *imm8* times. Uses a 6 bit count.
D0 /0	ROL *r/m8*, 1	Valid	Valid	Rotate 8 bits *r/m8* left once.
REX + D0 /0	ROL *r/m8**, 1	Valid	N.E.	Rotate 8 bits *r/m8* left once
D2 /0	ROL *r/m8*, CL	Valid	Valid	Rotate 8 bits *r/m8* left CL times.
REX + D2 /0	ROL *r/m8**, CL	Valid	N.E.	Rotate 8 bits *r/m8* left CL times.
C0 /0 *ib*	ROL *r/m8, imm8*	Valid	Valid	Rotate 8 bits *r/m8* left *imm8* times.
REX + C0 /0 *ib*	ROL *r/m8*, imm8*	Valid	N.E.	Rotate 8 bits *r/m8* left *imm8* times.
D1 /0	ROL *r/m16*, 1	Valid	Valid	Rotate 16 bits *r/m16* left once.
D3 /0	ROL *r/m16*, CL	Valid	Valid	Rotate 16 bits *r/m16* left CL times.
C1 /0 *ib*	ROL *r/m16, imm8*	Valid	Valid	Rotate 16 bits *r/m16* left *imm8* times.
D1 /0	ROL *r/m32*, 1	Valid	Valid	Rotate 32 bits *r/m32* left once.
REX.W + D1 /0	ROL *r/m64*, 1	Valid	N.E.	Rotate 64 bits *r/m64* left once. Uses a 6 bit count.

Opcode**	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D3 /0	ROL *r/m32*, CL	Valid	Valid	Rotate 32 bits *r/m32* left CL times.
REX.W + D3 /0	ROL *r/m64*, CL	Valid	N.E.	Rotate 64 bits *r/m64* left CL times. Uses a 6 bit count.
C1 /0 *ib*	ROL *r/m32*, *imm8*	Valid	Valid	Rotate 32 bits *r/m32* left *imm8* times.
C1 /0 *ib*	ROL *r/m64*, *imm8*	Valid	N.E.	Rotate 64 bits *r/m64* left *imm8* times. Uses a 6 bit count.
D0 /1	ROR *r/m8*, 1	Valid	Valid	Rotate 8 bits *r/m8* right once.
REX + D0 /1	ROR *r/m8**, 1	Valid	N.E.	Rotate 8 bits *r/m8* right once.
D2 /1	ROR *r/m8*, CL	Valid	Valid	Rotate 8 bits *r/m8* right CL times.
REX + D2 /1	ROR *r/m8**, CL	Valid	N.E.	Rotate 8 bits *r/m8* right CL times.
C0 /1 *ib*	ROR *r/m8*, *imm8*	Valid	Valid	Rotate 8 bits *r/m16* right *imm8* times.
REX + C0 /1 *ib*	ROR *r/m8**, *imm8*	Valid	N.E.	Rotate 8 bits *r/m16* right *imm8* times.
D1 /1	ROR *r/m16*, 1	Valid	Valid	Rotate 16 bits *r/m16* right once.
D3 /1	ROR *r/m16*, CL	Valid	Valid	Rotate 16 bits *r/m16* right CL times.
C1 /1 *ib*	ROR *r/m16*, *imm8*	Valid	Valid	Rotate 16 bits *r/m16* right *imm8* times.
D1 /1	ROR *r/m32*, 1	Valid	Valid	Rotate 32 bits *r/m32* right once.
REX.W + D1 /1	ROR *r/m64*, 1	Valid	N.E.	Rotate 64 bits *r/m64* right once. Uses a 6 bit count.
D3 /1	ROR *r/m32*, CL	Valid	Valid	Rotate 32 bits *r/m32* right CL times.
REX.W + D3 /1	ROR *r/m64*, CL	Valid	N.E.	Rotate 64 bits *r/m64* right CL times. Uses a 6 bit count.
C1 /1 *ib*	ROR *r/m32*, *imm8*	Valid	Valid	Rotate 32 bits *r/m32* right *imm8* times.
REX.W + C1 /1 *ib*	ROR *r/m64*, *imm8*	Valid	N.E.	Rotate 64 bits *r/m64* right *imm8* times. Uses a 6 bit count.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a 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 a 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.

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 ← OperandSize;
CASE (determine count) OF
    SIZE ← 8:     tempCOUNT ← (COUNT AND 1FH) MOD 9;
    SIZE ← 16:    tempCOUNT ← (COUNT AND 1FH) MOD 17;
    SIZE ← 32:    tempCOUNT ← COUNT AND 1FH;
```

```
    SIZE ← 64:    tempCOUNT ← COUNT AND 3FH;
ESAC;

(* RCL instruction operation *)
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← MSB(DEST);
        DEST ← (DEST ∗ 2) + CF;
        CF ← tempCF;
        tempCOUNT ← tempCOUNT – 1;
    OD;
ELIHW;
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;

(* RCR instruction operation *)
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← LSB(SRC);
        DEST ← (DEST / 2) + (CF * 2^SIZE);
        CF ← tempCF;
        tempCOUNT ← tempCOUNT – 1;
    OD;

(* ROL and ROR instructions *)
SIZE ← 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 ∗ 2^SIZE);
            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.

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 *xmm1, xmm2/m128*	Valid	Valid	Computes the approximate reciprocals of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.

Description

Performs a 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a 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 ∞ 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 $|1.11111111110100000000000B*2^{125}|$ 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 a 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)
```

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 *xmm1*, xmm2/m32	Valid	Valid	Computes the approximate reciprocal of the scalar single-precision floating-point value in *xmm2/m32* and stores the result in *xmm1*.

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 *Intel® 64 and IA-32 Architectures 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 ∞ 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 $|1.11111111110100000000000B*2^{125}|$ 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 a 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.

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 *Intel® 64 and IA-32 Architectures 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 *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

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	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium 4, Intel Xeon processors	(Family 0FH; Model 03H, 04H, 06H) and (L3 is absent)	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium M processors	Family 06H, Model 09H, 0DH	0, 1	0, 1
64-bit Intel Xeon processors with L3	(Family 0FH; Model 03H, 04H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17
Intel® Core™ Solo and Intel Core Duo processors	Family 06H, Model 0EH	0, 1	0, 1
Intel® Core™2 Duo processor, Intel Xeon processor 5100 Series - general-purpose PMC	Family 06H, Model 0FH	0, 1	0, 1
Intel Core 2 Duo processor, Intel Xeon processor 5100 Series - fixed-function PMC	Family 06H, Model 0FH	8000_0000H, 8000_0001H	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.

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 *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, lists the events that can be counted for various processors in the Intel 64 and IA-32 architecture families.

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 tow 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

```
(* Intel Core 2 Duo processor family and Intel Xeon processor 5100 series*)

IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[31] = 1)
```

```
        EAX ← IA32_FIXED_CTR(ECX)[30:0];
        EDX ← IA32_FIXED_CTR(ECX)[39:32];
    ELSE IF (ECX[30:0] in valid range)
        EAX ← PMC(ECX[30:0])[31:0];
        EDX ← PMC(ECX[30:0])[39:32];
    ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;

(* 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;
```

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 *Intel® 64 and IA-32 Architectures 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

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/REPNZ—Repeat String Operation Prefix

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 6C	REP INS *m8*, DX	Valid	Valid	Input (E)CX bytes from port DX into ES:[(E)DI].
F3 6C	REP INS *m8*, DX	Valid	N.E.	Input RCX bytes from port DX into [RDI].
F3 6D	REP INS *m16*, DX	Valid	Valid	Input (E)CX words from port DX into ES:[(E)DI.]
F3 6D	REP INS *m32*, DX	Valid	Valid	Input (E)CX doublewords from port DX into ES:[(E)DI].
F3 6D	REP INS *r/m32*, DX	Valid	N.E.	Input RCX default size from port DX into [RDI].
F3 A4	REP MOVS *m8, m8*	Valid	Valid	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A4	REP MOVS *m8, m8*	Valid	N.E.	Move RCX bytes from [RSI] to [RDI].
F3 A5	REP MOVS *m16, m16*	Valid	Valid	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI].
F3 A5	REP MOVS *m32, m32*	Valid	Valid	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A5	REP MOVS *m64, m64*	Valid	N.E.	Move RCX quadwords from [RSI] to [RDI].
F3 6E	REP OUTS DX, *r/m8*	Valid	Valid	Output (E)CX bytes from DS:[(E)SI] to port DX.
F3 REX.W 6E	REP OUTS DX, *r/m8**	Valid	N.E.	Output RCX bytes from [RSI] to port DX.
F3 6F	REP OUTS DX, *r/m16*	Valid	Valid	Output (E)CX words from DS:[(E)SI] to port DX.
F3 6F	REP OUTS DX, *r/m32*	Valid	Valid	Output (E)CX doublewords from DS:[(E)SI] to port DX.
F3 REX.W 6F	REP OUTS DX, *r/m32*	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.
F3 REX.W 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.
F3 REX.W AD	REP LODS RAX	Valid	N.E.	Load RCX quadwords from [RSI] to RAX.

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 AA	REP STOS *m8*	Valid	Valid	Fill (E)CX bytes at ES:[(E)DI] with AL.
F3 REX.W AA	REP STOS *m8*	Valid	N.E.	Fill RCX bytes at [RDI] with AL.
F3 AB	REP STOS *m16*	Valid	Valid	Fill (E)CX words at ES:[(E)DI] with AX.
F3 AB	REP STOS *m32*	Valid	Valid	Fill (E)CX doublewords at ES:[(E)DI] with EAX.
F3 REX.W AB	REP STOS *m64*	Valid	N.E.	Fill RCX quadwords at [RDI] with RAX.
F3 A6	REPE CMPS *m8, m8*	Valid	Valid	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A6	REPE CMPS *m8, m8*	Valid	N.E.	Find non-matching bytes in [RDI] and [RSI].
F3 A7	REPE CMPS *m16, m16*	Valid	Valid	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI].
F3 A7	REPE CMPS *m32, m32*	Valid	Valid	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A7	REPE CMPS *m64, m64*	Valid	N.E.	Find non-matching quadwords in [RDI] and [RSI].
F3 AE	REPE SCAS *m8*	Valid	Valid	Find non-AL byte starting at ES:[(E)DI].
F3 REX.W AE	REPE SCAS *m8*	Valid	N.E.	Find non-AL byte starting at [RDI].
F3 AF	REPE SCAS *m16*	Valid	Valid	Find non-AX word starting at ES:[(E)DI].
F3 AF	REPE SCAS *m32*	Valid	Valid	Find non-EAX doubleword starting at ES:[(E)DI].
F3 REX.W AF	REPE SCAS *m64*	Valid	N.E.	Find non-RAX quadword starting at [RDI].
F2 A6	REPNE CMPS *m8, m8*	Valid	Valid	Find matching bytes in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A6	REPNE CMPS *m8, m8*	Valid	N.E.	Find matching bytes in [RDI] and [RSI].
F2 A7	REPNE CMPS *m16, m16*	Valid	Valid	Find matching words in ES:[(E)DI] and DS:[(E)SI].
F2 A7	REPNE CMPS *m32, m32*	Valid	Valid	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A7	REPNE CMPS *m64, m64*	Valid	N.E.	Find matching doublewords in [RDI] and [RSI].

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F2 AE	REPNE SCAS *m8*	Valid	Valid	Find AL, starting at ES:[(E)DI].
F2 REX.W AE	REPNE SCAS *m8*	Valid	N.E.	Find AL, starting at [RDI].
F2 AF	REPNE SCAS *m16*	Valid	Valid	Find AX, starting at ES:[(E)DI].
F2 AF	REPNE SCAS *m32*	Valid	Valid	Find EAX, starting at ES:[(E)DI].
F2 REX.W AF	REPNE SCAS *m64*	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 a 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).

When the REPE/REPZ and REPNE/REPNZ 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 a REX prefix in the form of REX.W does promote operation to 64-bit operands for other REP/REPNE/REPZ/REPNZ 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 REPNZ 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 *iw*	RET *imm16*	Valid	Valid	Near return to calling procedure and pop *imm16* bytes from stack.
CA *iw*	RET *imm16*	Valid	Valid	Far return to calling procedure and pop *imm16* 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 *Intel® 64 and IA-32 Architectures 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
                        RSP ← RSP + SRC; (* Release parameters from stack *)
                    ELSE (* StackAddressSize = 16 *)
                        SP ← SP + SRC; (* Release parameters from stack *)
                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;
    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
            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;
        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;
                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 then 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.
	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 *Intel® 64 and IA-32 Architectures 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.

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 *xmm1*, *xmm2/m128*	Valid	Valid	Computes the approximate reciprocals of the square roots of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.

Description

Performs a 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a 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 ∞ 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 a 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_rsqrt_ps(__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.
	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 *xmm1, xmm2/m32*	Valid	Valid	Computes the approximate reciprocal of the square root of the low single-precision floating-point value in *xmm2/m32* and stores the results in *xmm1*.

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 doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures 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 ∞ 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 a 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_rsqrt_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.

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.

NOTE:

* 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.

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 *r/m8*, 1	Valid	Valid	Multiply *r/m8* by 2, once.
REX + D0 /4	SAL *r/m8***, 1	Valid	N.E.	Multiply *r/m8* by 2, once.
D2 /4	SAL *r/m8*, CL	Valid	Valid	Multiply *r/m8* by 2, CL times.
REX + D2 /4	SAL *r/m8***, CL	Valid	N.E.	Multiply *r/m8* by 2, CL times.
C0 /4 *ib*	SAL *r/m8, imm8*	Valid	Valid	Multiply *r/m8* by 2, *imm8* times.
REX + C0 /4 *ib*	SAL *r/m8**, imm8*	Valid	N.E.	Multiply *r/m8* by 2, *imm8* times.
D1 /4	SAL *r/m16*, 1	Valid	Valid	Multiply *r/m16* by 2, once.
D3 /4	SAL *r/m16*, CL	Valid	Valid	Multiply *r/m16* by 2, CL times.
C1 /4 *ib*	SAL *r/m16, imm8*	Valid	Valid	Multiply *r/m16* by 2, *imm8* times.
D1 /4	SAL *r/m32*, 1	Valid	Valid	Multiply *r/m32* by 2, once.
REX.W + D1 /4	SAL *r/m64*, 1	Valid	N.E.	Multiply *r/m64* by 2, once.
D3 /4	SAL *r/m32*, CL	Valid	Valid	Multiply *r/m32* by 2, CL times.
REX.W + D3 /4	SAL *r/m64*, CL	Valid	N.E.	Multiply *r/m64* by 2, CL times.
C1 /4 *ib*	SAL *r/m32, imm8*	Valid	Valid	Multiply *r/m32* by 2, *imm8* times.
REX.W + C1 /4 *ib*	SAL *r/m64, imm8*	Valid	N.E.	Multiply *r/m64* by 2, *imm8* times.
D0 /7	SAR *r/m8*, 1	Valid	Valid	Signed divide* *r/m8* by 2, once.
REX + D0 /7	SAR *r/m8***, 1	Valid	N.E.	Signed divide* *r/m8* by 2, once.
D2 /7	SAR *r/m8*, CL	Valid	Valid	Signed divide* *r/m8* by 2, CL times.
REX + D2 /7	SAR *r/m8***, CL	Valid	N.E.	Signed divide* *r/m8* by 2, CL times.
C0 /7 *ib*	SAR *r/m8, imm8*	Valid	Valid	Signed divide* *r/m8* by 2, *imm8* time.
REX + C0 /7 *ib*	SAR *r/m8**, imm8*	Valid	N.E.	Signed divide* *r/m8* by 2, *imm8* times.
D1 /7	SAR *r/m16*,1	Valid	Valid	Signed divide* *r/m16* by 2, once.

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D3 /7	SAR *r/m16*, CL	Valid	Valid	Signed divide* *r/m16* by 2, CL times.
C1 /7 *ib*	SAR *r/m16, imm8*	Valid	Valid	Signed divide* *r/m16* by 2, *imm8* times.
D1 /7	SAR *r/m32*, 1	Valid	Valid	Signed divide* *r/m32* by 2, once.
REX.W + D1 /7	SAR *r/m64*, 1	Valid	N.E.	Signed divide* *r/m64* by 2, once.
D3 /7	SAR *r/m32*, CL	Valid	Valid	Signed divide* *r/m32* by 2, CL times.
REX.W + D3 /7	SAR *r/m64*, CL	Valid	N.E.	Signed divide* *r/m64* by 2, CL times.
C1 /7 *ib*	SAR *r/m32, imm8*	Valid	Valid	Signed divide* *r/m32* by 2, *imm8* times.
REX.W + C1 /7 *ib*	SAR *r/m64, imm8*	Valid	N.E.	Signed divide* *r/m64* by 2, *imm8* times
D0 /4	SHL *r/m8*, 1	Valid	Valid	Multiply *r/m8* by 2, once.
REX + D0 /4	SHL *r/m8***, 1	Valid	N.E.	Multiply *r/m8* by 2, once.
D2 /4	SHL *r/m8*, CL	Valid	Valid	Multiply *r/m8* by 2, CL times.
REX + D2 /4	SHL *r/m8***, CL	Valid	N.E.	Multiply *r/m8* by 2, CL times.
C0 /4 *ib*	SHL *r/m8, imm8*	Valid	Valid	Multiply *r/m8* by 2, *imm8* times.
REX + C0 /4 *ib*	SHL *r/m8**, imm8*	Valid	N.E.	Multiply *r/m8* by 2, *imm8* times.
D1 /4	SHL *r/m16*,1	Valid	Valid	Multiply *r/m16* by 2, once.
D3 /4	SHL *r/m16*, CL	Valid	Valid	Multiply *r/m16* by 2, CL times.
C1 /4 *ib*	SHL *r/m16, imm8*	Valid	Valid	Multiply *r/m16* by 2, *imm8* times.
D1 /4	SHL *r/m32*,1	Valid	Valid	Multiply *r/m32* by 2, once.
REX.W + D1 /4	SHL *r/m64*,1	Valid	N.E.	Multiply *r/m64* by 2, once.
D3 /4	SHL *r/m32*, CL	Valid	Valid	Multiply *r/m32* by 2, CL times.
REX.W + D3 /4	SHL *r/m64*, CL	Valid	N.E.	Multiply *r/m64* by 2, CL times.
C1 /4 *ib*	SHL *r/m32, imm8*	Valid	Valid	Multiply *r/m32* by 2, *imm8* times.
REX.W + C1 /4 *ib*	SHL *r/m64, imm8*	Valid	N.E.	Multiply *r/m64* by 2, *imm8* times.

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D0 /5	SHR *r/m8*,1	Valid	Valid	Unsigned divide *r/m8* by 2, once.
REX + D0 /5	SHR *r/m8***, 1	Valid	N.E.	Unsigned divide *r/m8* by 2, once.
D2 /5	SHR *r/m8*, CL	Valid	Valid	Unsigned divide *r/m8* by 2, CL times.
REX + D2 /5	SHR *r/m8***, CL	Valid	N.E.	Unsigned divide *r/m8* by 2, CL times.
C0 /5 *ib*	SHR *r/m8, imm8*	Valid	Valid	Unsigned divide *r/m8* by 2, *imm8* times.
REX + C0 /5 *ib*	SHR *r/m8**, imm8*	Valid	N.E.	Unsigned divide *r/m8* by 2, *imm8* times.
D1 /5	SHR *r/m16*, 1	Valid	Valid	Unsigned divide *r/m16* by 2, once.
D3 /5	SHR *r/m16*, CL	Valid	Valid	Unsigned divide *r/m16* by 2, CL times
C1 /5 *ib*	SHR *r/m16, imm8*	Valid	Valid	Unsigned divide *r/m16* by 2, *imm8* times.
D1 /5	SHR *r/m32*, 1	Valid	Valid	Unsigned divide *r/m32* by 2, once.
REX.W + D1 /5	SHR *r/m64*, 1	Valid	N.E.	Unsigned divide *r/m64* by 2, once.
D3 /5	SHR *r/m32*, CL	Valid	Valid	Unsigned divide *r/m32* by 2, CL times.
REX.W + D3 /5	SHR *r/m64*, CL	Valid	N.E.	Unsigned divide *r/m64* by 2, CL times.
C1 /5 *ib*	SHR *r/m32, imm8*	Valid	Valid	Unsigned divide *r/m32* by 2, *imm8* times.
REX.W + C1 /5 *ib*	SHR *r/m64, imm8*	Valid	N.E.	Unsigned divide *r/m64* by 2, *imm8* 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 a 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 *Intel® 64 and IA-32 Architectures 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 *Intel® 64 and IA-32 Architectures 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

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 a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a 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.

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 *ib*	SBB AL, *imm8*	Valid	Valid	Subtract with borrow *imm8* from AL.
1D *iw*	SBB AX, *imm16*	Valid	Valid	Subtract with borrow *imm16* from AX.
1D *id*	SBB EAX, *imm32*	Valid	Valid	Subtract with borrow *imm32* from EAX.
REX.W + 1D *id*	SBB RAX, *imm32*	Valid	N.E.	Subtract with borrow sign-extended *imm.32 to 64-bits* from RAX.
80 /3 *ib*	SBB *r/m8, imm8*	Valid	Valid	Subtract with borrow *imm8* from *r/m8.*
REX + 80 /3 *ib*	SBB *r/m8*, imm8*	Valid	N.E.	Subtract with borrow *imm8* from *r/m8.*
81 /3 *iw*	SBB *r/m16, imm16*	Valid	Valid	Subtract with borrow *imm16* from *r/m16.*
81 /3 *id*	SBB *r/m32, imm32*	Valid	Valid	Subtract with borrow *imm32* from *r/m32.*
REX.W + 81 /3 *id*	SBB *r/m64, imm32*	Valid	N.E.	Subtract with borrow sign-extended *imm32 to 64-bits* from *r/m64.*
83 /3 *ib*	SBB *r/m16, imm8*	Valid	Valid	Subtract with borrow sign-extended *imm8* from *r/m16.*
83 /3 *ib*	SBB *r/m32, imm8*	Valid	Valid	Subtract with borrow sign-extended *imm8* from *r/m32.*
REX.W + 83 /3 *ib*	SBB *r/m64, imm8*	Valid	N.E.	Subtract with borrow sign-extended *imm8* from *r/m64.*
18 /*r*	SBB *r/m8, r8*	Valid	Valid	Subtract with borrow *r8* from *r/m8.*
REX + 18 /*r*	SBB *r/m8*, r8*	Valid	N.E.	Subtract with borrow *r8* from *r/m8.*
19 /*r*	SBB *r/m16, r16*	Valid	Valid	Subtract with borrow *r16* from *r/m16.*
19 /*r*	SBB *r/m32, r32*	Valid	Valid	Subtract with borrow *r32* from *r/m32.*

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
REX.W + 19 /*r*	SBB *r/m64, r64*	Valid	N.E.	Subtract with borrow *r64* from *r/m64.*
1A /*r*	SBB *r8, r/m8*	Valid	Valid	Subtract with borrow *r/m8* from *r8.*
REX + 1A /*r*	SBB *r8*, r/m8**	Valid	N.E.	Subtract with borrow *r/m8* from *r8.*
1B /*r*	SBB *r16, r/m16*	Valid	Valid	Subtract with borrow *r/m16* from *r16.*
1B /*r*	SBB *r32, r/m32*	Valid	Valid	Subtract with borrow *r/m32* from *r32.*
REX.W + 1B /*r*	SBB *r64, r/m64*	Valid	N.E.	Subtract with borrow *r/m64* from *r64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a 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 a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a 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 – (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 *m8*	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI, then set status flags[1].
AF	SCAS *m16*	Valid	Valid	Compare AX with word at ES:(E)DI or RDI, then set status flags.[a]
AF	SCAS *m32*	Valid	Valid	Compare EAX with doubleword at ES(E)DI or RDI then set status flags.[a]
REX.W + AF	SCAS *m64*	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.[a]
AF	SCASW	Valid	Valid	Compare AX with word at ES:(E)DI or RDI then set status flags.[a]
AF	SCASD	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.[a]
REX.W + AF	SCASQ	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.

NOTES:

1. 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 a 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 cmparison)
    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;
```

64-bit Mode:

```
IF (Byte cmparison)
    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;
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.

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.

SET*cc*—Set Byte on Condition

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 97	SETA *r/m8*	Valid	Valid	Set byte if above (CF=0 and ZF=0).
REX + 0F 97	SETA *r/m8**	Valid	N.E.	Set byte if above (CF=0 and ZF=0).
0F 93	SETAE *r/m8*	Valid	Valid	Set byte if above or equal (CF=0).
REX + 0F 93	SETAE *r/m8**	Valid	N.E.	Set byte if above or equal (CF=0).
0F 92	SETB *r/m8*	Valid	Valid	Set byte if below (CF=1).
REX + 0F 92	SETB *r/m8**	Valid	N.E.	Set byte if below (CF=1).
0F 96	SETBE *r/m8*	Valid	Valid	Set byte if below or equal (CF=1 or ZF=1).
REX + 0F 96	SETBE *r/m8**	Valid	N.E.	Set byte if below or equal (CF=1 or ZF=1).
0F 92	SETC *r/m8*	Valid	Valid	Set byte if carry (CF=1).
REX + 0F 92	SETC *r/m8**	Valid	N.E.	Set byte if carry (CF=1).
0F 94	SETE *r/m8*	Valid	Valid	Set byte if equal (ZF=1).
REX + 0F 94	SETE *r/m8**	Valid	N.E.	Set byte if equal (ZF=1).
0F 9F	SETG *r/m8*	Valid	Valid	Set byte if greater (ZF=0 and SF=OF).
REX + 0F 9F	SETG *r/m8**	Valid	N.E.	Set byte if greater (ZF=0 and SF=OF).
0F 9D	SETGE *r/m8*	Valid	Valid	Set byte if greater or equal (SF=OF).
REX + 0F 9D	SETGE *r/m8**	Valid	N.E.	Set byte if greater or equal (SF=OF).
0F 9C	SETL *r/m8*	Valid	Valid	Set byte if less (SF≠ OF).
REX + 0F 9C	SETL *r/m8**	Valid	N.E.	Set byte if less (SF≠ OF).
0F 9E	SETLE *r/m8*	Valid	Valid	Set byte if less or equal (ZF=1 or SF≠ OF).
REX + 0F 9E	SETLE *r/m8**	Valid	N.E.	Set byte if less or equal (ZF=1 or SF≠ OF).
0F 96	SETNA *r/m8*	Valid	Valid	Set byte if not above (CF=1 or ZF=1).
REX + 0F 96	SETNA *r/m8**	Valid	N.E.	Set byte if not above (CF=1 or ZF=1).
0F 92	SETNAE *r/m8*	Valid	Valid	Set byte if not above or equal (CF=1).

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
REX + 0F 92	SETNAE *r/m8**	Valid	N.E.	Set byte if not above or equal (CF=1).
0F 93	SETNB *r/m8*	Valid	Valid	Set byte if not below (CF=0).
REX + 0F 93	SETNB *r/m8**	Valid	N.E.	Set byte if not below (CF=0).
0F 97	SETNBE *r/m8*	Valid	Valid	Set byte if not below or equal (CF=0 and ZF=0).
REX + 0F 97	SETNBE *r/m8**	Valid	N.E.	Set byte if not below or equal (CF=0 and ZF=0).
0F 93	SETNC *r/m8*	Valid	Valid	Set byte if not carry (CF=0).
REX + 0F 93	SETNC *r/m8**	Valid	N.E.	Set byte if not carry (CF=0).
0F 95	SETNE *r/m8*	Valid	Valid	Set byte if not equal (ZF=0).
REX + 0F 95	SETNE *r/m8**	Valid	N.E.	Set byte if not equal (ZF=0).
0F 9E	SETNG *r/m8*	Valid	Valid	Set byte if not greater (ZF=1 or SF≠ OF)
REX + 0F 9E	SETNG *r/m8**	Valid	N.E.	Set byte if not greater (ZF=1 or SF≠ OF).
0F 9C	SETNGE *r/m8*	Valid	Valid	Set byte if not greater or equal (SF≠ OF).
REX + 0F 9C	SETNGE *r/m8**	Valid	N.E.	Set byte if not greater or equal (SF≠ OF).
0F 9D	SETNL *r/m8*	Valid	Valid	Set byte if not less (SF=OF).
REX + 0F 9D	SETNL *r/m8**	Valid	N.E.	Set byte if not less (SF=OF).
0F 9F	SETNLE *r/m8*	Valid	Valid	Set byte if not less or equal (ZF=0 and SF=OF).
REX + 0F 9F	SETNLE *r/m8**	Valid	N.E.	Set byte if not less or equal (ZF=0 and SF=OF).
0F 91	SETNO *r/m8*	Valid	Valid	Set byte if not overflow (OF=0).
REX + 0F 91	SETNO *r/m8**	Valid	N.E.	Set byte if not overflow (OF=0).
0F 9B	SETNP *r/m8*	Valid	Valid	Set byte if not parity (PF=0).
REX + 0F 9B	SETNP *r/m8**	Valid	N.E.	Set byte if not parity (PF=0).
0F 99	SETNS *r/m8*	Valid	Valid	Set byte if not sign (SF=0).
REX + 0F 99	SETNS *r/m8**	Valid	N.E.	Set byte if not sign (SF=0).
0F 95	SETNZ *r/m8*	Valid	Valid	Set byte if not zero (ZF=0).
REX + 0F 95	SETNZ *r/m8**	Valid	N.E.	Set byte if not zero (ZF=0).
0F 90	SETO *r/m8*	Valid	Valid	Set byte if overflow (OF=1)
REX + 0F 90	SETO *r/m8**	Valid	N.E.	Set byte if overflow (OF=1).
0F 9A	SETP *r/m8*	Valid	Valid	Set byte if parity (PF=1).
REX + 0F 9A	SETP *r/m8**	Valid	N.E.	Set byte if parity (PF=1).
0F 9A	SETPE *r/m8*	Valid	Valid	Set byte if parity even (PF=1).

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
REX + 0F 9A	SETPE *r/m8**	Valid	N.E.	Set byte if parity even (PF=1).
0F 9B	SETPO *r/m8*	Valid	Valid	Set byte if parity odd (PF=0).
REX + 0F 9B	SETPO *r/m8**	Valid	N.E.	Set byte if parity odd (PF=0).
0F 98	SETS *r/m8*	Valid	Valid	Set byte if sign (SF=1).
REX + 0F 98	SETS *r/m8**	Valid	N.E.	Set byte if sign (SF=1).
0F 94	SETZ *r/m8*	Valid	Valid	Set byte if zero (ZF=1).
REX + 0F 94	SETZ *r/m8**	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 a 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*cc* 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 OF. These alternate mnemonics are provided to make code more intelligible. Appendix B, “EFLAGS Condition Codes,” in the *Intel® 64 and IA-32 Architectures 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*cc* 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.

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 *m*	Valid	Valid	Store GDTR to *m*.

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, *Intel® 64 and IA-32 Architectures 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.

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 *r/m16, r16, imm8*	Valid	Valid	Shift *r/m16* to left *imm8* places while shifting bits from *r16* in from the right.
0F A5	SHLD *r/m16, r16*, CL	Valid	Valid	Shift *r/m16* to left CL places while shifting bits from *r16* in from the right.
0F A4	SHLD *r/m32, r32, imm8*	Valid	Valid	Shift *r/m32* to left *imm8* places while shifting bits from *r32* in from the right.
REX.W + 0F A4	SHLD *r/m64, r64, imm8*	Valid	N.E.	Shift *r/m64* to left *imm8* places while shifting bits from *r64* in from the right.
0F A5	SHLD *r/m32, r32*, CL	Valid	Valid	Shift *r/m32* to left CL places while shifting bits from *r32* in from the right.
REX.W + 0F A5	SHLD *r/m64, r64*, CL	Valid	N.E.	Shift *r/m64* to left CL places while shifting bits from *r64* 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 a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a 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, 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 *r/m16, r16, imm8*	Valid	Valid	Shift *r/m16* to right *imm8* places while shifting bits from *r16* in from the left.
0F AD	SHRD *r/m16, r16*, CL	Valid	Valid	Shift *r/m16* to right CL places while shifting bits from *r16* in from the left.
0F AC	SHRD *r/m32, r32, mm8*	Valid	Valid	Shift *r/m32* to right *imm8* places while shifting bits from *r32* in from the left.
REX.W + 0F AC	SHRD *r/m64, r64, imm8*	Valid	N.E.	Shift *r/m64* to right *imm8* places while shifting bits from *r64* in from the left.
0F AD	SHRD *r/m32, r32*, CL	Valid	Valid	Shift *r/m32* to right CL places while shifting bits from *r32* in from the left.
REX.W + 0F AD	SHRD *r/m64, r64*, CL	Valid	N.E.	Shift *r/m64* to right CL places while shifting bits from *r64* 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 a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a 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.

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 *xmm1*, *xmm2/m128*, *imm8*	Valid	Valid	Shuffle packed double-precision floating-point values selected by *imm8* from *xmm1* and *xmm2/m128* to *xmm1*.

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 4-13). The select operand (third operand) determines which values are moved to the destination operand.

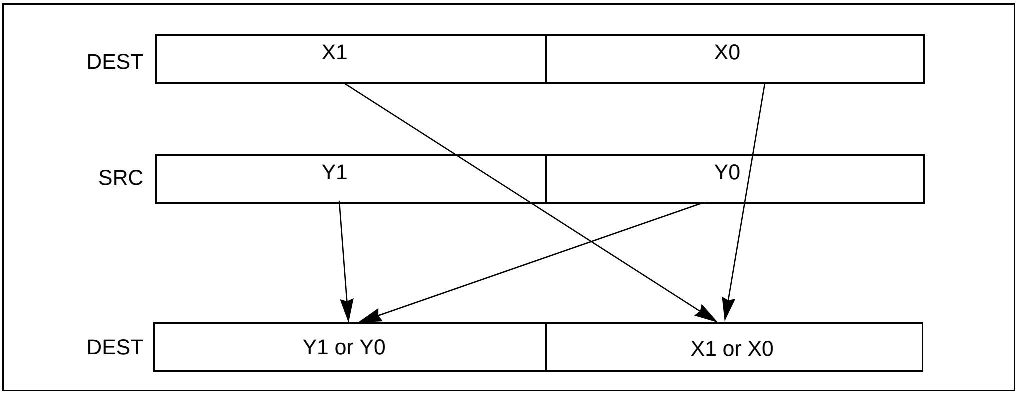


Figure 4-13. SHUFPD Shuffle Operation

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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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 *xmm1, xmm2/m128, imm8*	Valid	Valid	Shuffle packed single-precision floating-point values selected by *imm8* from *xmm1* and *xmm1/m128* to *xmm1*.

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 4-14). The select operand (third operand) determines which values are moved to the destination operand.

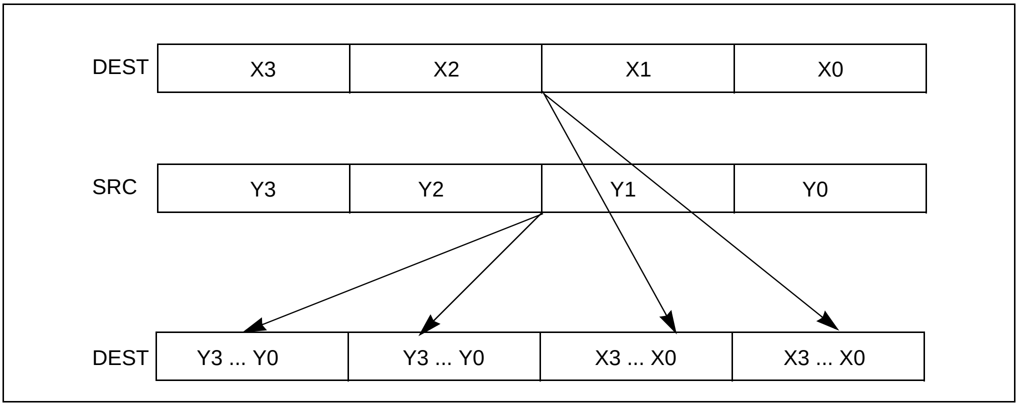


Figure 4-14. SHUFPS Shuffle Operation

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 a 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.
#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 *m*	Valid	Valid	Store IDTR to *m*.

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, *Intel® 64 and IA-32 Architectures 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;
```

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 *r/m16*	Valid	Valid	Stores segment selector from LDTR in *r/m16*.
REX.W + 0F 00 /0	SLDT *r64/m16*	Valid	Valid	Stores segment selector from LDTR in *r64/m16*.

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 a 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 *r/m16*	Valid	Valid	Store machine status word to *r/m16*.
0F 01 /4	SMSW *r32/m16*	Valid	Valid	Store machine status word in low-order 16 bits of *r32/m16*; high-order 16 bits of *r32* are undefined.
REX.W + 0F 01 /4	SMSW *r64/m16*	Valid	Valid	Store machine status word in low-order 16 bits of *r64/m16*; high-order 16 bits of *r32* 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 *Intel® 64 and IA-32 Architectures 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 *xmm1, xmm2/m128*	Valid	Valid	Computes square roots of the packed double-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.

Description

Performs a 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, using a 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.
#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 *xmm1, xmm2/m128*	Valid	Valid	Computes square roots of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.

Description

Performs a 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 *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

In 64-bit mode, using a 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.

#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 *xmm1, xmm2/m64*	Valid	Valid	Computes square root of the low double-precision floating-point value in *xmm2/m64* and stores the results in *xmm1*.

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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar double-precision floating-point operation.

In 64-bit mode, using a 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	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.

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 *xmm1, xmm2/m32*	Valid	Valid	Computes square root of the low single-precision floating-point value in *xmm2/m32* and stores the results in *xmm1.*

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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

In 64-bit mode, using a 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.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.
	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 ← 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[1]. 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 (and SMIs) 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.

---

1. 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**

NOTES:

X = This setting has no impact.

Operation

```
IF PE = 0  (* Executing in real-address mode *)
    THEN
        IF ← 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
                        IF ← 1;  (* Set Interrupt Flag *)
                ELSE
                    IF (IOPL < CPL) and (CPL = 3) and (VIP = 0)
                        THEN
                            VIF ← 1;  (* Set Virtual Interrupt Flag *)
                        ELSE
                            #GP(0);
                    FI;
                FI;
            ELSE  (* Executing in Virtual-8086 mode *)
                IF IOPL = 3
                    THEN
                        IF ← 1;  (* Set Interrupt Flag *)
```

```
                ELSE
                    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;
        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 *m32*	Valid	Valid	Store contents of MXCSR register to *m32*.

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 ← 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.

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 *m8*	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 *m16*	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 *m32*	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 *m64*	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 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 a 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.
	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.
#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 *r/m16*	Valid	Valid	Stores segment selector from TR in *r/m16*.

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.

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 *ib*	SUB AL, *imm8*	Valid	Valid	Subtract *imm8* from AL.
2D *iw*	SUB AX, *imm16*	Valid	Valid	Subtract *imm16* from AX.
2D *id*	SUB EAX, *imm32*	Valid	Valid	Subtract *imm32* from EAX.
REX.W + 2D *id*	SUB RAX, *imm32*	Valid	N.E.	Subtract *imm32* sign-extended to 64-bits from RAX.
80 /5 *ib*	SUB *r/m8, imm8*	Valid	Valid	Subtract *imm8* from *r/m8.*
REX + 80 /5 *ib*	SUB *r/m8*, imm8*	Valid	N.E.	Subtract *imm8* from *r/m8.*
81 /5 *iw*	SUB *r/m16, imm16*	Valid	Valid	Subtract *imm16* from *r/m16.*
81 /5 *id*	SUB *r/m32, imm32*	Valid	Valid	Subtract *imm32* from *r/m32.*
REX.W + 81 /5 *id*	SUB *r/m64, imm32*	Valid	N.E.	Subtract *imm32* sign-extended to 64-bits from *r/m64.*
83 /5 *ib*	SUB *r/m16, imm8*	Valid	Valid	Subtract sign-extended *imm8* from *r/m16.*
83 /5 *ib*	SUB *r/m32, imm8*	Valid	Valid	Subtract sign-extended *imm8* from *r/m32.*
REX.W + 83 /5 *ib*	SUB *r/m64, imm8*	Valid	N.E.	Subtract sign-extended *imm8* from *r/m64.*
28 /*r*	SUB *r/m8, r8*	Valid	Valid	Subtract *r8* from *r/m8.*
REX + 28 /*r*	SUB *r/m8*, r8**	Valid	N.E.	Subtract *r8* from *r/m8.*
29 /*r*	SUB *r/m16, r16*	Valid	Valid	Subtract *r16* from *r/m16.*
29 /*r*	SUB *r/m32, r32*	Valid	Valid	Subtract *r32* from *r/m32.*
REX.W + 29 /*r*	SUB *r/m64, r32*	Valid	N.E.	Subtract *r64* from *r/m64.*
2A /*r*	SUB *r8, r/m8*	Valid	Valid	Subtract *r/m8* from *r8.*
REX + 2A /*r*	SUB *r8*, r/m8**	Valid	N.E.	Subtract *r/m8* from *r8.*
2B /*r*	SUB *r16, r/m16*	Valid	Valid	Subtract *r/m16* from *r16.*
2B /*r*	SUB *r32, r/m32*	Valid	Valid	Subtract *r/m32* from *r32.*
REX.W + 2B /*r*	SUB *r64, r/m64*	Valid	N.E.	Subtract *r/m64* from *r64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a 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.

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 a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a 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 *xmm1, xmm2/m128*	Valid	Valid	Subtract packed double-precision floating-point values in *xmm2/m128* from *xmm1*.

Description

Performs a 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, using a 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.

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 *xmm1 xmm2/m128*	Valid	Valid	Subtract packed single-precision floating-point values in *xmm2/mem* from *xmm1*.

Description

Performs a 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, using a 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.

#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 *xmm1, xmm2/m64*	Valid	Valid	Subtracts the low double-precision floating-point values in *xmm2/mem64* from *xmm1*.

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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar double-precision floating-point operation.

In 64-bit mode, using a 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	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.

SUBSS—Subtract Scalar Single-Precision Floating-Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 5C /*r*	SUBSS *xmm1, xmm2/m32*	Valid	Valid	Subtract the lower single-precision floating-point values in *xmm2/m32* from *xmm1*.

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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

In 64-bit mode, using a 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.

#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 0FFFFFFFFH).
- 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.L ≠ 1 ) or (IA32_EFER.LMA ≠ 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) ← 0xFFFFF;
SS(GRANULAR) ← 1;

Flags Affected

All.

Protected Mode Exceptions

#UD	If Mode ≠ 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 Intel 64 and 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;
```

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 ← FFFFFH;                      (* 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 ← FFFFFH;                      (* 4-GByte limit *)
SS.ARbyte.G ← 1;                        (* 4-KByte granularity *)
SS.ARbyte.S ←;
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 protections 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 Intel 64 and 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 ← FFFFFH;                      (* 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 *);
SS.BASE ← 0;                              (* Flat segment *)
SS.LIMIT ← FFFFFH;                        (* 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.LMA ≠ 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 *ib*	TEST AL, i*mm8*	Valid	Valid	AND *imm8* with AL; set SF, ZF, PF according to result.
A9 *iw*	TEST AX, i*mm16*	Valid	Valid	AND *imm16* with AX; set SF, ZF, PF according to result.
A9 *id*	TEST EAX, i*mm32*	Valid	Valid	AND *imm32* with EAX; set SF, ZF, PF according to result.
REX.W + A9 *id*	TEST RAX, i*mm32*	Valid	N.E.	AND *imm32* sign-extended to 64-bits with RAX; set SF, ZF, PF according to result.
F6 /0 *ib*	TEST *r/m8, imm8*	Valid	Valid	AND *imm8* with *r/m8*; set SF, ZF, PF according to result.
REX + F6 /0 *ib*	TEST *r/m8*, imm8*	Valid	N.E.	AND *imm8* with *r/m8*; set SF, ZF, PF according to result.
F7 /0 *iw*	TEST *r/m16, imm16*	Valid	Valid	AND *imm16* with *r/m16*; set SF, ZF, PF according to result.
F7 /0 *id*	TEST *r/m32, imm32*	Valid	Valid	AND *imm32* with *r/m32*; set SF, ZF, PF according to result.
REX.W + F7 /0 *id*	TEST *r/m64, imm32*	Valid	N.E.	AND *imm32* sign-extended to 64-bits with *r/m64*; set SF, ZF, PF according to result.
84 /*r*	TEST *r/m8, r8*	Valid	Valid	AND *r8* with *r/m8*; set SF, ZF, PF according to result.
REX + 84 /*r*	TEST *r/m8*, r8**	Valid	N.E.	AND *r8* with *r/m8*; set SF, ZF, PF according to result.
85 /*r*	TEST *r/m16, r16*	Valid	Valid	AND *r16* with *r/m16*; set SF, ZF, PF according to result.
85 /*r*	TEST *r/m32, r32*	Valid	Valid	AND *r32* with *r/m32*; set SF, ZF, PF according to result.
REX.W + 85 /*r*	TEST *r/m64, r64*	Valid	N.E.	AND *r64* with *r/m64*; 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 a 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 a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a 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.

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 *xmm1, xmm2/m64*	Valid	Valid	Compares (unordered) the low double-precision floating-point values in *xmm1* and *xmm2/m64* and set the EFLAGS accordingly.

Description

Performs and 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 a 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_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 *xmm1, xmm2/m32*	Valid	Valid	Compare lower single-precision floating-point value in *xmm1* register with lower single-precision floating-point value in *xmm2/mem* and set the status flags accordingly.

Description

Performs and unordered compare of the single-precision floating-point values in the low doublewords 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 a 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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
RESULT ← UnorderedCompare(SRC1[31:0] <> SRC2[31:0]) {
(* 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.
	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 *xmm1*, *xmm2/m128*	Valid	Valid	Unpacks and Interleaves double-precision floating-point values from high quadwords of *xmm1* and *xmm2/m128*.

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 4-15. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

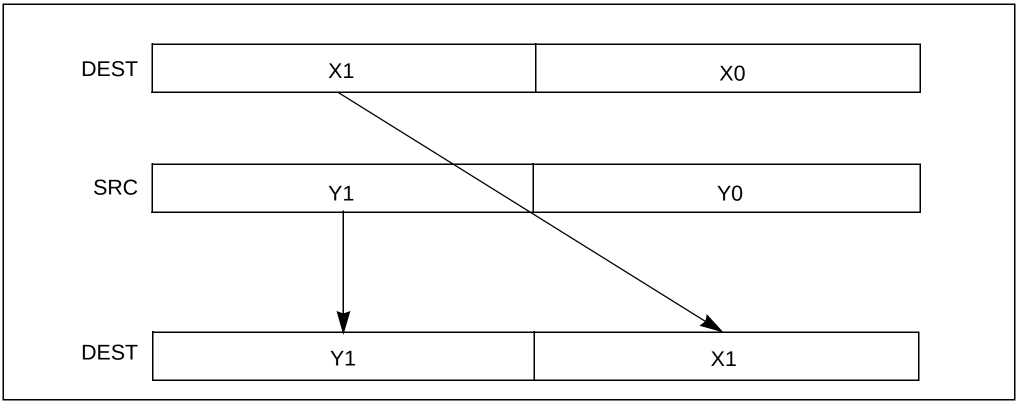


Figure 4-15. UNPCKHPD Instruction High Unpack and Interleave Operation

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 a 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.

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 *xmm1, xmm2/m128*	Valid	Valid	Unpacks and Interleaves single-precision floating-point values from high quadwords of *xmm1* and *xmm2/mem* into *xmm1*.

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 4-16. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

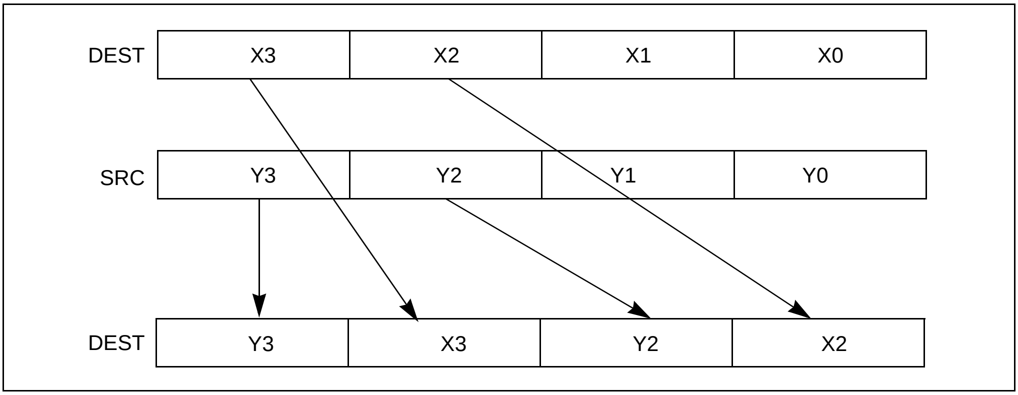


Figure 4-16. UNPCKHPS Instruction High Unpack and Interleave Operation

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 a 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];
```

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 /r	UNPCKLPD *xmm1, xmm2/m128*	Valid	Valid	Unpacks and Interleaves double-precision floating-point values from low quadwords of *xmm1* and *xmm2/m128*.

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 4-17. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

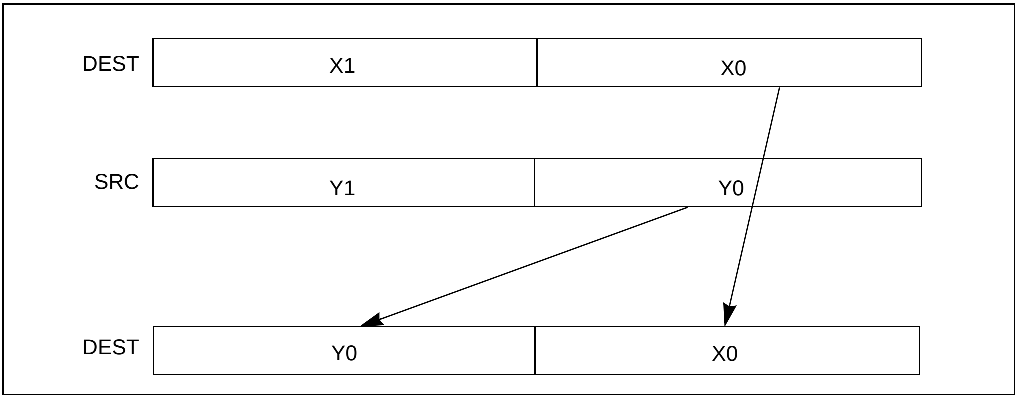


Figure 4-17. UNPCKLPD Instruction Low Unpack and Interleave Operation

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 a 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.

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 *xmm1, xmm2/m128*	Valid	Valid	Unpacks and Interleaves single-precision floating-point values from low quadwords of *xmm1* and *xmm2/mem* into *xmm1*.

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 4-18. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

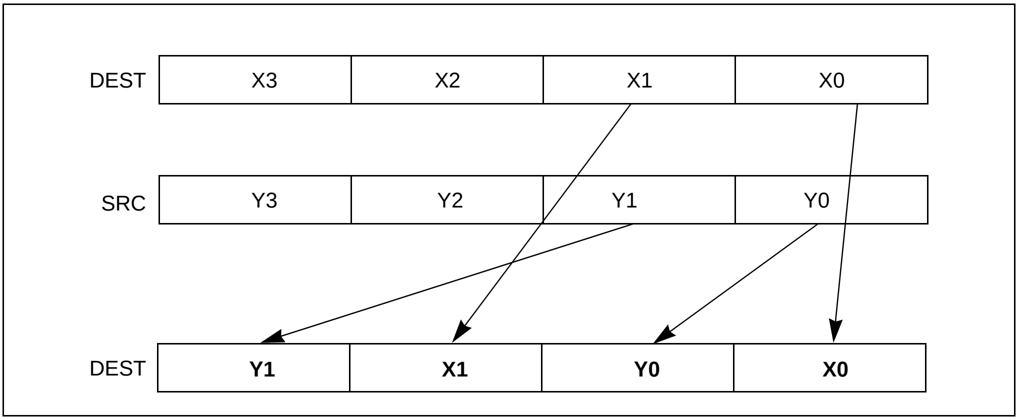


Figure 4-18. UNPCKLPS Instruction Low Unpack and Interleave Operation

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 a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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 *r/m16*	Valid	Valid	Set ZF=1 if segment specified with *r/m16* can be read.
0F 00 /5	VERW *r/m16*	Valid	Valid	Set ZF=1 if segment specified with *r/m16* 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.

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 *Intel® 64 and IA-32 Architectures 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.

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 *Intel® 64 and IA-32 Architectures 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 Intel 64 and 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.

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 *Intel® 64 and IA-32 Architectures 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 *Intel® 64 and IA-32 Architectures 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 *Intel® 64 and IA-32 Architectures 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.

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 a 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 a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a 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.

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, *r16*	Valid	Valid	Exchange *r16* with AX.
90+*rw*	XCHG *r16*, AX	Valid	Valid	Exchange AX with *r16.*
90+*rd*	XCHG EAX, *r32*	Valid	Valid	Exchange *r32* with EAX.
REX.W + 90+*rd*	XCHG RAX, *r64*	Valid	N.E.	Exchange *r64* with RAX.
90+*rd*	XCHG *r32*, EAX	Valid	Valid	Exchange EAX with *r32.*
REX.W + 90+*rd*	XCHG *r64*, RAX	Valid	N.E.	Exchange RAX with *r64.*
86 /*r*	XCHG *r/m8, r8*	Valid	Valid	Exchange *r8* (byte register) with byte from *r/m8.*
REX + 86 /*r*	XCHG *r/m8*, r8**	Valid	N.E.	Exchange *r8* (byte register) with byte from *r/m8.*
86 /*r*	XCHG *r8, r/m8*	Valid	Valid	Exchange byte from *r/m8* with *r8* (byte register).
REX + 86 /*r*	XCHG *r8*, r/m8**	Valid	N.E.	Exchange byte from *r/m8* with *r8* (byte register).
87 /*r*	XCHG *r/m16, r16*	Valid	Valid	Exchange *r16* with word from *r/m16.*
87 /*r*	XCHG *r16, r/m16*	Valid	Valid	Exchange word from *r/m16* with *r16.*
87 /*r*	XCHG *r/m32, r32*	Valid	Valid	Exchange *r32* with doubleword from *r/m32.*
REX.W + 87 /*r*	XCHG *r/m64, r64*	Valid	N.E.	Exchange *r64* with quadword from *r/m64.*
87 /*r*	XCHG *r32, r/m32*	Valid	Valid	Exchange doubleword from *r/m32* with *r32.*
REX.W + 87 /*r*	XCHG *r64, r/m64*	Valid	N.E.	Exchange quadword from *r/m64* with *r64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information on bus locking.)

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 a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a 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 *m8*	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 *ib*	XOR AL, i*mm8*	Valid	Valid	AL XOR *imm8.*
35 *iw*	XOR AX, i*mm16*	Valid	Valid	AX XOR *imm16.*
35 *id*	XOR EAX, i*mm32*	Valid	Valid	EAX XOR *imm32.*
REX.W + 35 *id*	XOR RAX, i*mm32*	Valid	N.E.	RAX XOR *imm32 (sign-extended).*
80 /6 *ib*	XOR *r/m8, imm8*	Valid	Valid	*r/m8* XOR *imm8.*
REX + 80 /6 *ib*	XOR *r/m8*, imm8*	Valid	N.E.	*r/m8* XOR *imm8.*
81 /6 *iw*	XOR *r/m16, imm16*	Valid	Valid	*r/m16* XOR *imm16.*
81 /6 *id*	XOR *r/m32, imm32*	Valid	Valid	*r/m32* XOR *imm32.*
REX.W + 81 /6 *id*	XOR *r/m64, imm32*	Valid	N.E.	*r/m64* XOR *imm32 (sign-extended).*
83 /6 *ib*	XOR *r/m16, imm8*	Valid	Valid	*r/m16* XOR *imm8 (sign-extended).*
83 /6 *ib*	XOR *r/m32, imm8*	Valid	Valid	*r/m32* XOR *imm8 (sign-extended).*
REX.W + 83 /6 *ib*	XOR *r/m64, imm8*	Valid	N.E.	*r/m64* XOR *imm8 (sign-extended).*
30 /*r*	XOR *r/m8, r8*	Valid	Valid	*r/m8* XOR *r8.*
REX + 30 /*r*	XOR *r/m8*, r8**	Valid	N.E.	*r/m8* XOR *r8.*
31 /*r*	XOR *r/m16, r16*	Valid	Valid	*r/m16* XOR *r16.*
31 /*r*	XOR *r/m32, r32*	Valid	Valid	*r/m32* XOR *r32.*
REX.W + 31 /*r*	XOR *r/m64, r64*	Valid	N.E.	*r/m64* XOR *r64.*
32 /*r*	XOR *r8, r/m8*	Valid	Valid	*r8* XOR *r/m8.*
REX + 32 /*r*	XOR *r8*, r/m8**	Valid	N.E.	*r8* XOR *r/m8.*
33 /*r*	XOR *r16, r/m16*	Valid	Valid	*r16* XOR *r/m16.*
33 /*r*	XOR *r32, r/m32*	Valid	Valid	*r32* XOR *r/m32.*
REX.W + 33 /*r*	XOR *r64, r/m64*	Valid	N.E.	*r64* XOR *r/m64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a 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 a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a 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.

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 *xmm1, xmm2/m128*	Valid	Valid	Bitwise exclusive-OR of *xmm2/m128* and *xmm1*.

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 a 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.

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 *xmm1, xmm2/m128*	Valid	Valid	Bitwise exclusive-OR of *xmm2/m128* and *xmm1*.

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 a 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.

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.

CHAPTER 5
VMX INSTRUCTION REFERENCE

5.1 OVERVIEW

This chapter describes the virtual-machine extensions (VMX) for the Intel 64 and IA-32 architectures. 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:

- V**MPTRLD —** 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.

- **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 *Intel® 64 and IA-32 Architectures 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 Intel® 64 and IA-32 Architectures 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 are not valid (see Section 24.16.6.1 of the Intel® 64 and IA-32 Archi-
tectures Software Developer's Manual, Volume 3B)
    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
```

```
            leave SMM;
            VMfailValid(VMCALL with incorrect MSEG revision identifier);
        ELSE
            read SMM-monitor features field in MSEG (see Section 24.16.6.2,
            in the Intel® 64 and IA-32 Architectures 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 Intel® 64 and IA-32 Architectures 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/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 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.

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 *Intel® 64 and IA-32 Architectures 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 64 architecture and
        addr sets any bits beyond the physical-address width) OR
        (processor does not support Intel 64 architecture, 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;
        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/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 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 *Intel® 64 and IA-32 Architectures 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
                    Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B);
                    ELSE
                        Attempt to load MSRs from VM-entry MSR-load area;
                        IF failure
                            THEN VM entry fails (see Section 22.7, in the Intel® 64 and IA-32
                            Architectures 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*.

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.[1]

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 64 architecture and
        addr sets any bits beyond the processor's physical-address width) OR
        processor does not support Intel 64 architecture 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;
```

1. 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*).

```
        FI;
FI;
```

Flags Affected

See the operation section and Section 5.2.

Use of Prefixes

LOCK	Causes #UD
REPNE/REPNZ	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 /7	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/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 a 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 a memory destination operand.
#SS(0)	If a memory destination operand effective address is outside the SS segment limit.
	If the SS register contains an unusable segment.
#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.

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/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 a 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 a memory source operand.
#SS(0)	If a memory source operand effective address is outside the SS segment limit.
	If the SS register contains an unusable segment.
#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.

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.[1]

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/REPNZ and REP/REPE/REPZ)
Segment overrides	Ignored
Operand size	Causes #UD

1. See the information on MONITOR/MWAIT in Chapter 7, “Multiple-Processor Management,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

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
F3 0F C7 /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.[1]

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 operation2) 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 64 architecture and
                addr sets any bits beyond the VMX physical-address width) or
                (processor does not support Intel 64 architecture 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;
```

1. See the information on MONITOR/MWAIT in Chapter 7, “Multiple-Processor Management,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

2. See Section 19.8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

```
                        ELSE
                            current-VMCS pointer ← FFFFFFFF_FFFFFFFFH;
                            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.

APPENDIX A
OPCODE MAP

Use the opcode tables in this chapter to interpret Intel 64 and 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/SSSE3, 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 lower-case 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.

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 *Intel® 64 and IA-32 Architectures 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 (OFH). 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 *Intel® 64 and IA-32 Architectures 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).

Example A-2. Look-up Example for 2-Byte Opcodes

Look-up opcode 0FA4050000000003H 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 (00000000H). 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:00000000H, 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 *Intel® 64 and IA-32 Architectures 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).

A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS

See Table A-2 through Table A-5 below. The tables are multiple page presenations. 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^{i64}	POP ES^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	ADC						PUSH SS^{i64}	POP SS^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	AND						SEG=ES (Prefix)	DAA^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	XOR						SEG=SS (Prefix)	AAA^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
4	INC^{i64} general register / REX^{o64} 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^{d64} general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA^{i64}/ PUSHAD^{i64}	POPA^{i64}/ POPAD^{i64}	BOUND^{i64} Gv, Ma	ARPL^{i64} Ew, Gw MOVSXD^{o64} Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc^{f64}, 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^{1A}				TEST		XCHG	
	Eb, Ib	Ev, Iz	Eb, Ib^{i64}	Ev, Ib	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, Ib	CL/R9L, Ib	DL/R10L, Ib	BL/R11L, Ib	AH/R12L, Ib	CH/R13L, Ib	DH/R14L, Ib	BH/R15L, Ib
C	Shift Grp 2^{1A}		RETN^{f64} Iw	RETN^{f64}	LES^{i64} Gz, Mp	LDS^{i64} Gz, Mp	Grp 11^{1A} - MOV	
	Eb, Ib	Ev, Ib					Eb, Ib	Ev, Iz
D	Shift Grp 2^{1A}				AAM^{i64} Ib	AAD^{i64} Ib		XLAT/ XLATB
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	LOOPNE^{f64}/ LOOPNZ^{f64} Jb	LOOPE^{f64}/ LOOPZ^{f64} Jb	LOOP^{f64} Jb	JrCXZ^{f64}/ Jb	IN		OUT	
					AL, Ib	eAX, Ib	Ib, AL	Ib, eAX
F	LOCK (Prefix)		REPNE (Prefix)	REP/ REPE (Prefix)	HLT	CMC	Unary Grp 3^{1A}	
							Eb	Ev

Table A-2. One-byte Opcode Map: (08H — FFH) *

	8	9	A	B	C	D	E	F
0	OR						PUSH CS^{i64}	2-byte escape (Table A-3)
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	SBB						PUSH DS^{i64}	POP DS^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	SUB						SEG=CS (Prefix)	DAS^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	CMP						SEG=DS (Prefix)	AAS^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
4	DEC^{i64} general register / REX^{o64} 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^{d64} into general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSH^{d64} Iz	IMUL Gv, Ev, Iz	PUSH^{d64} Ib	IMUL Gv, Ev, Ib	INS/ INSB Yb, DX	INS/ INSW/ INSD Yz, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/ OUTSD DX, Xz
7	Jcc^{f64}, Jb- Short displacement jump on condition							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
8	MOV				MOV Ev, Sw	LEA Gv, M	MOV Sw, Ew	Grp 1A^{1A} POP^{d64} Ev
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev				
9	CBW/ CWDE/ CDQE	CWD/ CDQ/ CQO	CALLF^{i64} Ap	FWAIT/ WAIT	PUSHF/D/Q ^{d64}/ Fv	POPF/D/Q ^{d64}/ Fv	SAHF	LAHF
A	TEST		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
	AL, Ib	rAX, Iz						
B	MOV immediate word or double into word, double, or quad register							
	rAX/r8, Iv	rCX/r9, Iv	rDX/r10, Iv	rBX/r11, Iv	rSP/r12, Iv	rBP/r13, Iv	rSI/r14, Iv	rDI/r15 , Iv
C	ENTER Iw, Ib	LEAVE^{d64}	RETF Iw	RETF	INT 3	INT Ib	INTO^{i64}	IRET/D/Q
D	ESC (Escape to coprocessor instruction set)							
E	CALL^{f64}	JMP			IN		OUT	
	Jz	near^{f64} Jz	far^{i64} AP	short^{f64} Jb	AL, DX	eAX, DX	DX, AL	DX, eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4^{1A}	INC/DEC Grp 5^{1A}

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[1A]	Grp 7[1A]	LAR Gv, Ew	LSL Gv, Ew		SYSCALL[o64]	CLTS	SYSRET[o64]
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 movlhps 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	RDTSC	RDMSR	RDPMC	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 sqrtss (F3) Vss, Wss sqrtpd (66) Vpd, Wpd sqrtsd (F2) Vsd, Wsd	rsqrtps Vps, Wps rsqrtss (F3) Vss, Wss	rcpps Vps, Wps rcpss (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 pshuflw (F2) Vdq Wdq,Ib	(Grp 12[1A])	(Grp 13[1A])	(Grp 14[1A])	pcmpeqb Pq, Qq pcmpeqb (66) Vdq, Wdq	pcmpeqw Pq, Qq pcmpeqw (66) Vdq, Wdq	pcmpeqd Pq, Qq pcmpeqd (66) Vdq, Wdq	emms

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[1B]		NOP Ev		
1	Prefetch[1C] (Grp 16[1A])							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	cvttps2pi Qq, Wps cvttss2si (F3) Gd, Wss cvttpd2pi (66) Qdq, Wpd cvttsd2si (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	cvtdq2ps Vps, Wdq cvtps2dq (66) Vdq, Wps cvttps2dq (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[f64], 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/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A	PUSH[d64] FS	POP[d64] FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B	CMPXCHG		LSS Gv, Mp	BTR Ev, Gv	LFS Gv, Mp	LGS Gv, Mp	MOVZX	
	Eb, Gb	Ev, Gv					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 cmpsd (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[1A]
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 pmovmksb (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	pmulhuw Pq, Qq pmulhuw (66) Vdq, Wdq	pmulhw Pq, Qq pmulhw (66) Vdq, Wdq	cvtpd2dq (F2) Vdq, Wpd cvttpd2dq (66) Vdq, Wpd cvtdq2pd (F3) Vpd, Wdq	movntq Mq, Pq movntdq (66) Mdq, Vdq
F	lddqu (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

Table A-3. Two-byte Opcode Map: 88H — FFH (First Byte is 0FH) *

	8	9	A	B	C	D	E	F
8	Jcc[f64], 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[d64] GS	POP[d64] GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15[1A])[1C]	IMUL Gv, Ev
B	*JMPE* (reserved for emulator on IPF)	Grp 10[1A] Invalid Opcode[1B]	Grp 8[1A] 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	phaddd Pq, Qq phaddd (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								

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								

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, Ib palignr(66) Vdq, Wdq, Ib
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-2. 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.

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^{f64} Ev	CALLF Ep	JMPN^{f64} Ev	JMPF Ep	PUSH^{d64} 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)						SWAPGS ^{o64}(000)
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, Ib psrlw (66) Udq, Ib		psraw Nq, Ib psraw (66) Udq, Ib		psllw Nq, Ib psllw (66) Udq, Ib	
0F 72	13	mem								
		11B			psrld Nq, Ib psrld (66) Udq, Ib		psrad Nq, Ib psrad (66) Udq, Ib		pslld Nq, Ib pslld (66) Udq, Ib	
0F 73	14	mem								
		11B			psrlq Nq, Ib psrlq (66) Udq, Ib	psrldq (66) Udq, Ib			psllq Nq, Ib psllq (66) Udq, Ib	pslldq (66) Udq, Ib
0F AE	15	mem	fxsave	fxrstor	ldmxcsr	stmxcsr				clflush
		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.

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:

- 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-3. 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.

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	FCMOVB							
	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	FCMOVBE							
	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	FCMOVE							
	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	FCMOVU							
	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.

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.

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		FCOMPP						
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.

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	FCOMIP							
	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	FUCOMIP							
	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.

APPENDIX B
INSTRUCTION FORMATS AND ENCODINGS

This appendix provides machine instruction formats and encodings of IA-32 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. Those instruction formats also apply to Intel 64 architecture. 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)

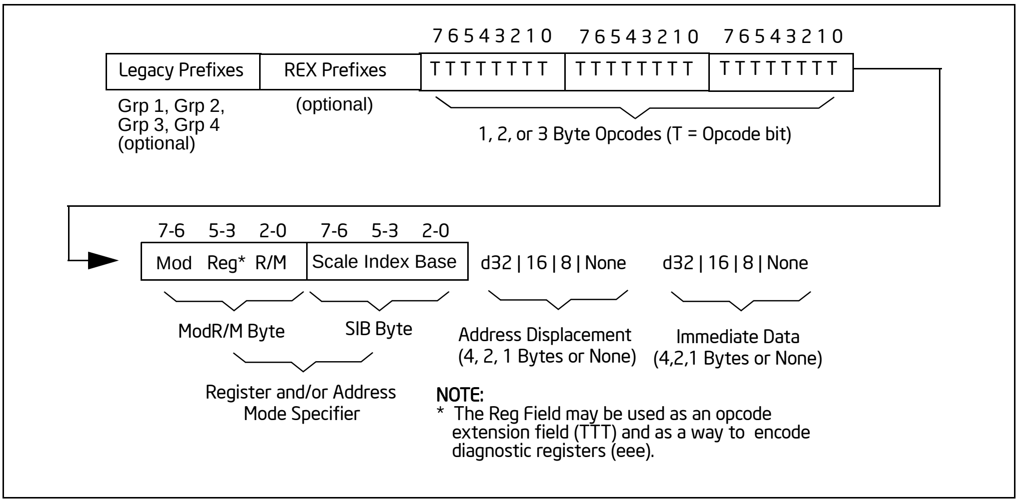


Figure B-1. General Machine Instruction Format

The following sections discuss this format.

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 *Intel® 64 and IA-32 Architectures 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 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 *Intel® 64 and IA-32 Architectures 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 *Intel® 64 and IA-32 Architectures 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
tttn	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

Table B-3. Encoding of reg Field When w Field is Present in Instruction

Register Specified by reg Field During 16-Bit Data Operations		
	Function of w Field	
reg	When w = 0	When w = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Register Specified by reg Field During 32-Bit Data Operations		
	Function of w Field	
reg	When w = 0	When w = 1
000	AL	EAX
001	CL	ECX
010	DL	EDX
011	BL	EBX
100	AH	ESP
101	CH	EBP
110	DH	ESI
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		
	Function of w Field	
reg	When w = 0	When w = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH[1]	SP
101	CH[1]	BP
110	DH[1]	SI
111	BH[1]	DI

Register Specified by reg Field During 32-Bit Data Operations		
	Function of w Field	
reg	When w = 0	When w = 1
000	AL	EAX
001	CL	ECX
010	DL	EDX
011	BL	EBX
100	AH*	ESP
101	CH*	EBP
110	DH*	ESI
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.

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
00	ES
01	CS
10	SS
11	DS

3-Bit sreg3 Field	Segment Register Selected
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS
110	Reserved[1]
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[1]	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 (tttn) Field

For conditional instructions (such as conditional jumps and set on condition), the condition test field (tttn) 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 tttn field is located in bits 3, 2, 1, and 0 of the opcode byte.
- For 2-byte primary opcodes, the tttn field is located in bits 3, 2, 1, and 0 of the second opcode byte.

Table B-10 shows the encoding of the tttn field.

Table B-10. Encoding of Conditional Test (tttn) 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
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
AAA – ASCII Adjust after Addition	0011 0111
AAD – ASCII Adjust AX before Division	1101 0101 : 0000 1010
AAM – ASCII Adjust AX after Multiply	1101 0100 : 0000 1010
AAS – ASCII Adjust AL after Subtraction	0011 1111
ADC – ADD with Carry	
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
ADD – Add	
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

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
immediate to AL, AX, or EAX	0000 010w : immediate data
immediate to memory	1000 00sw : mod 000 r/m : immediate data
AND – Logical AND	
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
ARPL – Adjust RPL Field of Selector	
from register	0110 0011 : 11 reg1 reg2
from memory	0110 0011 : mod reg r/m
BOUND – Check Array Against Bounds	0110 0010 : mod^{A} reg r/m
BSF – Bit Scan Forward	
register1, register2	0000 1111 : 1011 1100 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1100 : mod reg r/m
BSR – Bit Scan Reverse	
register1, register2	0000 1111 : 1011 1101 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1101 : mod reg r/m
BSWAP – Byte Swap	0000 1111 : 1100 1 reg
BT – Bit Test	
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
BTC – Bit Test and Complement	
register, immediate	0000 1111 : 1011 1010 : 11 111 reg: imm8 data

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
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
BTR – Bit Test and Reset	
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
BTS – Bit Test and Set	
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
CALL – Call Procedure (in same segment)	
direct	1110 1000 : full displacement
register indirect	1111 1111 : 11 010 reg
memory indirect	1111 1111 : mod 010 r/m
CALL – Call Procedure (in other segment)	
direct	1001 1010 : unsigned full offset, selector
indirect	1111 1111 : mod 011 r/m
CBW – Convert Byte to Word	1001 1000
CDQ – Convert Doubleword to Qword	1001 1001
CLC – Clear Carry Flag	1111 1000
CLD – Clear Direction Flag	1111 1100
CLI – Clear Interrupt Flag	1111 1010
CLTS – Clear Task-Switched Flag in CR0	0000 1111 : 0000 0110

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
CMC – Complement Carry Flag	1111 0101
CMP – Compare Two Operands	
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
CMPS/CMPSB/CMPSW/CMPSD – Compare String Operands	1010 011w
CMPXCHG – Compare and Exchange	
register1, register2	0000 1111 : 1011 000w : 11 reg2 reg1
memory, register	0000 1111 : 1011 000w : mod reg r/m
CPUID – CPU Identification	0000 1111 : 1010 0010
CWD – Convert Word to Doubleword	1001 1001
CWDE – Convert Word to Doubleword	1001 1000
DAA – Decimal Adjust AL after Addition	0010 0111
DAS – Decimal Adjust AL after Subtraction	0010 1111
DEC – Decrement by 1	
register	1111 111w : 11 001 reg
register (alternate encoding)	0100 1 reg
memory	1111 111w : mod 001 r/m
DIV – Unsigned Divide	
AL, AX, or EAX by register	1111 011w : 11 110 reg
AL, AX, or EAX by memory	1111 011w : mod 110 r/m
ENTER – Make Stack Frame for High Level Procedure	1100 1000 : 16-bit displacement : 8-bit level (L)
HLT – Halt	1111 0100
IDIV – Signed Divide	

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
AL, AX, or EAX by register	1111 011w : 11 111 reg
AL, AX, or EAX by memory	1111 011w : mod 111 r/m
IMUL – Signed Multiply	
AL, AX, or EAX with register	1111 011w : 11 101 reg
AL, AX, or EAX with memory	1111 011w : mod 101 reg
register1 with register2	0000 1111 : 1010 1111 : 11 : reg1 reg2
register with memory	0000 1111 : 1010 1111 : mod reg r/m
register1 with immediate to register2	0110 10s1 : 11 reg1 reg2 : immediate data
memory with immediate to register	0110 10s1 : mod reg r/m : immediate data
IN – Input From Port	
fixed port	1110 010w : port number
variable port	1110 110w
INC – Increment by 1	
reg	1111 111w : 11 000 reg
reg (alternate encoding)	0100 0 reg
memory	1111 111w : mod 000 r/m
INS – Input from DX Port	0110 110w
INT n – Interrupt Type n	1100 1101 : type
INT – Single-Step Interrupt 3	1100 1100
INTO – Interrupt 4 on Overflow	1100 1110
INVD – Invalidate Cache	0000 1111 : 0000 1000
INVLPG – Invalidate TLB Entry	0000 1111 : 0000 0001 : mod 111 r/m
IRET/IRETD – Interrupt Return	1100 1111
***Jcc* – Jump if Condition is Met**	
8-bit displacement	0111 tttn : 8-bit displacement
full displacement	0000 1111 : 1000 tttn : full displacement
JCXZ/JECXZ – Jump on CX/ECX Zero Address-size prefix differentiates JCXZ and JECXZ	1110 0011 : 8-bit displacement
JMP – Unconditional Jump (to same segment)	

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
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
JMP – Unconditional Jump (to other segment)	
direct intersegment	1110 1010 : unsigned full offset, selector
indirect intersegment	1111 1111 : mod 101 r/m
LAHF – Load Flags into AHRegister	1001 1111
LAR – Load Access Rights Byte	
from register	0000 1111 : 0000 0010 : 11 reg1 reg2
from memory	0000 1111 : 0000 0010 : mod reg r/m
LDS – Load Pointer to DS	1100 0101 : mod^A reg r/m
LEA – Load Effective Address	1000 1101 : mod^A reg r/m
LEAVE – High Level Procedure Exit	1100 1001
LES – Load Pointer to ES	1100 0100 : mod^A reg r/m
LFS – Load Pointer to FS	0000 1111 : 1011 0100 : mod^A reg r/m
LGDT – Load Global Descriptor Table Register	0000 1111 : 0000 0001 : mod^A 010 r/m
LGS – Load Pointer to GS	0000 1111 : 1011 0101 : mod^A reg r/m
LIDT – Load Interrupt Descriptor Table Register	0000 1111 : 0000 0001 : mod^A 011 r/m
LLDT – Load Local Descriptor Table Register	
LDTR from register	0000 1111 : 0000 0000 : 11 010 reg
LDTR from memory	0000 1111 : 0000 0000 : mod 010 r/m
LMSW – Load Machine Status Word	
from register	0000 1111 : 0000 0001 : 11 110 reg
from memory	0000 1111 : 0000 0001 : mod 110 r/m
LOCK – Assert LOCK# Signal Prefix	1111 0000
LODS/LODSB/LODSW/LODSD – Load String Operand	1010 110w
LOOP – Loop Count	1110 0010 : 8-bit displacement

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
LOOPZ/LOOPE – Loop Count while Zero/Equal	1110 0001 : 8-bit displacement
LOOPNZ/LOOPNE – Loop Count while not Zero/Equal	1110 0000 : 8-bit displacement
LSL – Load Segment Limit	
from register	0000 1111 : 0000 0011 : 11 reg1 reg2
from memory	0000 1111 : 0000 0011 : mod reg r/m
LSS – Load Pointer to SS	0000 1111 : 1011 0010 : mod[A] reg r/m
LTR – Load Task Register	
from register	0000 1111 : 0000 0000 : 11 011 reg
from memory	0000 1111 : 0000 0000 : mod 011 r/m
MOV – Move Data	
register1 to register2	1000 100w : 11 reg1 reg2
register2 to register1	1000 101w : 11 reg1 reg2
memory to reg	1000 101w : mod reg r/m
reg to memory	1000 100w : mod reg r/m
immediate to register	1100 011w : 11 000 reg : immediate data
immediate to register (alternate encoding)	1011 w reg : immediate data
immediate to memory	1100 011w : mod 000 r/m : immediate data
memory to AL, AX, or EAX	1010 000w : full displacement
AL, AX, or EAX to memory	1010 001w : full displacement
MOV – Move to/from Control Registers	
CR0 from register	0000 1111 : 0010 0010 : 11 000 reg
CR2 from register	0000 1111 : 0010 0010 : 11 010reg
CR3 from register	0000 1111 : 0010 0010 : 11 011 reg
CR4 from register	0000 1111 : 0010 0010 : 11 100 reg
register from CR0-CR4	0000 1111 : 0010 0000 : 11 eee reg
MOV – Move to/from Debug Registers	
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

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
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
MOV – Move to/from Segment Registers	
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
MOVS/MOVSB/MOVSW/MOVSD – Move Data from String to String	1010 010w
MOVSX – Move with Sign-Extend	
register2 to register1	0000 1111 : 1011 111w : 11 reg1 reg2
memory to reg	0000 1111 : 1011 111w : mod reg r/m
MOVZX – Move with Zero-Extend	
register2 to register1	0000 1111 : 1011 011w : 11 reg1 reg2
memory to register	0000 1111 : 1011 011w : mod reg r/m
MUL – Unsigned Multiply	
AL, AX, or EAX with register	1111 011w : 11 100 reg
AL, AX, or EAX with memory	1111 011w : mod 100 reg
NEG – Two's Complement Negation	
register	1111 011w : 11 011 reg
memory	1111 011w : mod 011 r/m
NOP – No Operation	1001 0000
NOP – Multi-byte No Operation[1]	
register	0000 1111 0001 1111 : 11 000 reg
memory	0000 1111 0001 1111 : mod 000 r/m
NOT – One's Complement Negation	
register	1111 011w : 11 010 reg

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
memory	1111 011w : mod 010 r/m
OR – Logical Inclusive OR	
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
OUT – Output to Port	
fixed port	1110 011w : port number
variable port	1110 111w
OUTS – Output to DX Port	0110 111w
POP – Pop a Word from the Stack	
register	1000 1111 : 11 000 reg
register (alternate encoding)	0101 1 reg
memory	1000 1111 : mod 000 r/m
POP – Pop a Segment Register from the Stack (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
POPA/POPAD – Pop All General Registers	0110 0001
POPF/POPFD – Pop Stack into FLAGS or EFLAGS Register	1001 1101
PUSH – Push Operand onto the Stack	
register	1111 1111 : 11 110 reg
register (alternate encoding)	0101 0 reg
memory	1111 1111 : mod 110 r/m
immediate	0110 10s0 : immediate data

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
PUSH – Push Segment Register onto the Stack	
segment register CS,DS,ES,SS	000 sreg2 110
segment register FS,GS	0000 1111: 10 sreg3 000
PUSHA/PUSHAD – Push All General Registers	0110 0000
PUSHF/PUSHFD – Push Flags Register onto the Stack	1001 1100
RCL – Rotate thru Carry Left	
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
RCR – Rotate thru Carry Right	
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
RDMSR – Read from Model-Specific Register	0000 1111 : 0011 0010
RDPMC – Read Performance Monitoring Counters	0000 1111 : 0011 0011
RDTSC – Read Time-Stamp Counter	0000 1111 : 0011 0001
REP INS – Input String	1111 0011 : 0110 110w
REP LODS – Load String	1111 0011 : 1010 110w
REP MOVS – Move String	1111 0011 : 1010 010w
REP OUTS – Output String	1111 0011 : 0110 111w
REP STOS – Store String	1111 0011 : 1010 101w

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
REPE CMPS – Compare String	1111 0011 : 1010 011w
REPE SCAS – Scan String	1111 0011 : 1010 111w
REPNE CMPS – Compare String	1111 0010 : 1010 011w
REPNE SCAS – Scan String	1111 0010 : 1010 111w
RET – Return from Procedure (to same segment)	
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
RET – Return from Procedure (to other segment)	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement
ROL – Rotate Left	
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
ROR – Rotate Right	
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
RSM – Resume from System Management Mode	0000 1111 : 1010 1010
SAHF – Store AH into Flags	1001 1110
SAL – Shift Arithmetic Left	same instruction as SHL

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
SAR – Shift Arithmetic Right	
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
SBB – Integer Subtraction with Borrow	
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
SCAS/SCASB/SCASW/SCASD – Scan String	1010 111w
SETcc – Byte Set on Condition	
register	0000 1111 : 1001 tttn : 11 000 reg
memory	0000 1111 : 1001 tttn : mod 000 r/m
SGDT – Store Global Descriptor Table Register	0000 1111 : 0000 0001 : mod^A 000 r/m
SHL – Shift Left	
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
SHLD – Double Precision Shift Left	
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

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
register by CL	0000 1111 : 1010 0101 : 11 reg2 reg1
memory by CL	0000 1111 : 1010 0101 : mod reg r/m
SHR – Shift Right	
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
SHRD – Double Precision Shift Right	
register by immediate count	0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8
memory by immediate count	0000 1111 : 1010 1100 : mod reg r/m : imm8
register by CL	0000 1111 : 1010 1101 : 11 reg2 reg1
memory by CL	0000 1111 : 1010 1101 : mod reg r/m
SIDT – Store Interrupt Descriptor Table Register	0000 1111 : 0000 0001 : mod^{A} 001 r/m
SLDT – Store Local Descriptor Table Register	
to register	0000 1111 : 0000 0000 : 11 000 reg
to memory	0000 1111 : 0000 0000 : mod 000 r/m
SMSW – Store Machine Status Word	
to register	0000 1111 : 0000 0001 : 11 100 reg
to memory	0000 1111 : 0000 0001 : mod 100 r/m
STC – Set Carry Flag	1111 1001
STD – Set Direction Flag	1111 1101
STI – Set Interrupt Flag	1111 1011
STOS/STOSB/STOSW/STOSD – Store String Data	1010 101w
STR – Store Task Register	
to register	0000 1111 : 0000 0000 : 11 001 reg
to memory	0000 1111 : 0000 0000 : mod 001 r/m

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
SUB – Integer Subtraction	
register1 to register2	0010 100w : 11 reg1 reg2
register2 to register1	0010 101w : 11 reg1 reg2
memory to register	0010 101w : mod reg r/m
register to memory	0010 100w : mod reg r/m
immediate to register	1000 00sw : 11 101 reg : immediate data
immediate to AL, AX, or EAX	0010 110w : immediate data
immediate to memory	1000 00sw : mod 101 r/m : immediate data
TEST – Logical Compare	
register1 and register2	1000 010w : 11 reg1 reg2
memory and register	1000 010w : mod reg r/m
immediate and register	1111 011w : 11 000 reg : immediate data
immediate and AL, AX, or EAX	1010 100w : immediate data
immediate and memory	1111 011w : mod 000 r/m : immediate data
UD2 – Undefined instruction	0000 FFFF : 0000 1011
VERR – Verify a Segment for Reading	
register	0000 1111 : 0000 0000 : 11 100 reg
memory	0000 1111 : 0000 0000 : mod 100 r/m
VERW – Verify a Segment for Writing	
register	0000 1111 : 0000 0000 : 11 101 reg
memory	0000 1111 : 0000 0000 : mod 101 r/m
WAIT – Wait	1001 1011
WBINVD – Writeback and Invalidate Data Cache	0000 1111 : 0000 1001
WRMSR – Write to Model-Specific Register	0000 1111 : 0011 0000
XADD – Exchange and Add	
register1, register2	0000 1111 : 1100 000w : 11 reg2 reg1
memory, reg	0000 1111 : 1100 000w : mod reg r/m
XCHG – Exchange Register/Memory with Register	
register1 with register2	1000 011w : 11 reg1 reg2

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

Instruction and Format	Encoding
AX or EAX with reg	1001 0 reg
memory with reg	1000 011w : mod reg r/m
XLAT/XLATB – Table Look-up Translation	1101 0111
XOR – Logical Exclusive OR	
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
Prefix Bytes	
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

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
ADC – ADD with Carry	
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
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
immediate8 to memory64	0100 10XB : 1000 0031 : mod 010 r/m : imm8
ADD – Add	
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
AND – Logical AND	
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
memory64 to qwordregister	0100 1RXB : 0010 0011 : mod qwordreg r/m
register to memory	0100 0RXB : 0010 000w : mod reg r/m

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
BSF – Bit Scan Forward	
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
BSR – Bit Scan Reverse	
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
BSWAP – Byte Swap	0000 1111 : 1100 1 reg
BSWAP – Byte Swap	0100 100B 0000 1111 : 1100 1 qwordreg

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
BT – Bit Test	
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
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
BTC – Bit Test and Complement	
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
BTR – Bit Test and Reset	
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
BTS – Bit Test and Set	
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
CALL – Call Procedure (in same segment)	
direct	1110 1000 : displacement32
register indirect	0100 WR00^{W} 1111 1111 : 11 010 reg
memory indirect	0100 W0XB^{W} 1111 1111 : mod 010 r/m
CALL – Call Procedure (in other segment)	
indirect	1111 1111 : mod 011 r/m
indirect	0100 10XB 0100 1000 1111 1111 : mod 011 r/m
CBW – Convert Byte to Word	1001 1000
CDQ – Convert Doubleword to Qword+	1001 1001
CDQE – RAX, Sign-Extend of EAX	0100 1000 1001 1001
CLC – Clear Carry Flag	1111 1000
CLD – Clear Direction Flag	1111 1100
CLI – Clear Interrupt Flag	1111 1010
CLTS – Clear Task-Switched Flag in CR0	0000 1111 : 0000 0110
CMC – Complement Carry Flag	1111 0101
CMP – Compare Two Operands	
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
CMPS/CMPSB/CMPSW/CMPSD/CMPSQ – Compare String Operands	
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
CMPXCHG – Compare and Exchange	
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
CPUID – CPU Identification	0000 1111 : 1010 0010
CQO – Sign-Extend RAX	0100 1000 1001 1001
CWD – Convert Word to Doubleword	1001 1001
CWDE – Convert Word to Doubleword	1001 1000
DEC – Decrement by 1	
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
DIV – Unsigned Divide	
AL, AX, or EAX by register	0100 000B 1111 011w : 11 110 reg

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
ENTER – Make Stack Frame for High Level Procedure	1100 1000 : 16-bit displacement : 8-bit level (L)
HLT – Halt	1111 0100
IDIV – Signed Divide	
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
IMUL – Signed Multiply	
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 withmemory64	0100 1RXB 0000 1111 : 1010 1111 : mod qwordreg r/m
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
IN – Input From Port	
fixed port	1110 010w : port number
variable port	1110 110w
INC – Increment by 1	
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
INS – Input from DX Port	0110 110w
INT n – Interrupt Type n	1100 1101 : type
INT – Single-Step Interrupt 3	1100 1100
INTO – Interrupt 4 on Overflow	1100 1110
INVD – Invalidate Cache	0000 1111 : 0000 1000
INVLPG – Invalidate TLB Entry	0000 1111 : 0000 0001 : mod 111 r/m
IRETO – Interrupt Return	1100 1111
***J*cc – Jump if Condition is Met**	
8-bit displacement	0111 tttn : 8-bit displacement
displacements (excluding 16-bit relative offsets)	0000 1111 : 1000 tttn : displacement32
JCXZ/JECXZ – Jump on CX/ECX Zero Address-size prefix differentiates JCXZ and JECXZ	1110 0011 : 8-bit displacement
JMP – Unconditional Jump (to same segment)	
short	1110 1011 : 8-bit displacement
direct	1110 1001 : displacement32
register indirect	0100 W00B^{w} : 1111 1111 : 11 100 reg
memory indirect	0100 W0XB^{w} : 1111 1111 : mod 100 r/m
JMP – Unconditional Jump (to other segment)	
indirect intersegment	0100 00XB : 1111 1111 : mod 101 r/m
64-bit indirect intersegment	0100 10XB : 1111 1111 : mod 101 r/m

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
LAR – Load Access Rights Byte	
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
LEA – Load Effective Address	
in wordregister/dwordregister	0100 0RXB : 1000 1101 : mod^{A} reg r/m
in qwordregister	0100 1RXB : 1000 1101 : mod^{A} qwordreg r/m
LEAVE – High Level Procedure Exit	1100 1001
LFS – Load Pointer to FS	
FS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0100 : mod^{A} reg r/m
FS:r64 with far pointer from memory	0100 1RXB : 0000 1111 : 1011 0100 : mod^{A} qwordreg r/m
LGDT – Load Global Descriptor Table Register	0100 10XB : 0000 1111 : 0000 0001 : mod^{A} 010 r/m
LGS – Load Pointer to GS	
GS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0101 : mod^{A} reg r/m
GS:r64 with far pointer from memory	0100 1RXB : 0000 1111 : 1011 0101 : mod^{A} qwordreg r/m
LIDT – Load Interrupt Descriptor Table Register	0100 10XB : 0000 1111 : 0000 0001 : mod^{A} 011 r/m
LLDT – Load Local Descriptor Table Register	
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
LMSW – Load Machine Status Word	
from register	0100 000B : 0000 1111 : 0000 0001 : 11 110 reg
from memory	0100 00XB :0000 1111 : 0000 0001 : mod 110 r/m
LOCK – Assert LOCK# Signal Prefix	1111 0000
LODS/LODSB/LODSW/LODSD/LODSQ – Load String Operand	
at DS:(E)SI to AL/EAX/EAX	1010 110w
at (R)SI to RAX	0100 1000 1010 1101
LOOP – Loop Count	
if count != 0, 8-bit displacement	1110 0010
if count !=0, RIP + 8-bit displacement sign-extended to 64-bits	0100 1000 1110 0010
LOOPE – Loop Count while Zero/Equal	
if count != 0 & ZF =1, 8-bit displacement	1110 0001
if count !=0 & ZF = 1, RIP + 8-bit displacement sign-extended to 64-bits	0100 1000 1110 0001
LOOPNE/LOOPNZ – Loop Count while not Zero/Equal	
if count != 0 & ZF = 0, 8-bit displacement	1110 0000
if count !=0 & ZF = 0, RIP + 8-bit displacement sign-extended to 64-bits	0100 1000 1110 0000
LSL – Load Segment Limit	
from register	0000 1111 : 0000 0011 : 11 reg1 reg2
from qwordregister	0100 1R00 0000 1111 : 0000 0011 : 11 qwordreg1 reg2
from memory16	0000 1111 : 0000 0011 : mod reg r/m
from memory64	0100 1RXB 0000 1111 : 0000 0011 : mod qwordreg r/m
LSS – Load Pointer to SS	
SS:r16/r32 with far pointer from memory	0100 0RXB : 0000 1111 : 1011 0010 : mod[A] reg r/m

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
SS:r64 with far pointer from memory	0100 1WXB : 0000 1111 : 1011 0010 : mod[A] qwordreg r/m
LTR – Load Task Register	
from register	0100 0R00 : 0000 1111 : 0000 0000 : 11 011 reg
from memory	0100 00XB : 0000 1111 : 0000 0000 : mod 011 r/m
MOV – Move Data	
register1 to register2	0100 0R0B : 1000 100w : 11 reg1 reg2
qwordregister1 to qwordregister2	0100 1R0B 1000 1001 : 11 qwordeg1 qwordreg2
register2 to register1	0100 0R0B : 1000 101w : 11 reg1 reg2
qwordregister2 to qwordregister1	0100 1R0B 1000 1011 : 11 qwordreg1 qwordreg2
memory to reg	0100 0RXB : 1000 101w : mod reg r/m
memory64 to qwordregister	0100 1RXB 1000 1011 : mod qwordreg r/m
reg to memory	0100 0RXB : 1000 100w : mod reg r/m
qwordregister to memory64	0100 1RXB 1000 1001 : mod qwordreg r/m
immediate to register	0100 000B : 1100 011w : 11 000 reg : imm
immediate32 to qwordregister (zero extend)	0100 100B 1100 0111 : 11 000 qwordreg : imm32
immediate to register (alternate encoding)	0100 000B : 1011 w reg : imm
immediate64 to qwordregister (alternate encoding)	0100 100B 1011 1000 reg : imm64
immediate to memory	0100 00XB : 1100 011w : mod 000 r/m : imm
immediate32 to memory64 (zero extend)	0100 10XB 1100 0111 : mod 000 r/m : imm32
memory to AL, AX, or EAX	0100 0000 : 1010 000w : displacement
memory64 to RAX	0100 1000 1010 0001 : displacement64
AL, AX, or EAX to memory	0100 0000 : 1010 001w : displacement
RAX to memory64	0100 1000 1010 0011 : displacement64
MOV – Move to/from Control Registers	
CR0-CR4 from register	0100 0R0B : 0000 1111 : 0010 0010 : 11 eee reg (eee = CR#)

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
CRx from qwordregister	0100 1R0B : 0000 1111 : 0010 0010 : 11 eee qwordreg (Reee = CR#)
register from CR0-CR4	0100 0R0B : 0000 1111 : 0010 0000 : 11 eee reg (eee = CR#)
qwordregister from CRx	0100 1R0B 0000 1111 : 0010 0000 : 11 eee qwordreg (Reee = CR#)
MOV – Move to/from Debug Registers	
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#)
MOV – Move to/from Segment Registers	
register to segment register	0100 W00B^{W} : 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
MOVS/MOVSB/MOVSW/MOVSD/MOVSQ – Move Data from String to String	
Move data from string to string	1010 010w
Move data from string to string (qword)	0100 1000 1010 0101
MOVSX/MOVSXD – Move with Sign-Extend	
register2 to register1	0100 0R0B : 0000 1111 : 1011 111w : 11 reg1 reg2

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
MOVZX – Move with Zero-Extend	
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
MUL – Unsigned Multiply	
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
NEG – Two's Complement Negation	
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
NOP – No Operation	1001 0000

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
NOT – One's Complement Negation	
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
OR – Logical Inclusive OR	
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
register to memory	0000 100w : mod reg r/m
byteregister to memory8	0100 0RXB 0000 1000 : mod bytereg r/m
qwordregister to memory64	0100 1RXB 0000 1001 : mod qwordreg r/m
immediate to register	1000 00sw : 11 001 reg : imm
immediate8 to byteregister	0100 000B 1000 0000 : 11 001 bytereg : imm8
immediate32 to qwordregister	0100 000B 1000 0001 : 11 001 qwordreg : imm32
immediate8 to qwordregister	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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
OUT – Output to Port	
fixed port	1110 011w : port number
variable port	1110 111w
OUTS – Output to DX Port	
output to DX Port	0110 111w
POP – Pop a Value from the Stack	
wordregister	0101 0101 : 0100 000B : 1000 1111 : 11 000 reg16
qwordregister	0100 W00B[S] : 1000 1111 : 11 000 reg64
wordregister (alternate encoding)	0101 0101 : 0100 000B : 0101 1 reg16
qwordregister (alternate encoding)	0100 W00B : 0101 1 reg64
memory64	0100 W0XB[S] : 1000 1111 : mod 000 r/m
memory16	0101 0101 : 0100 00XB 1000 1111 : mod 000 r/m
POP – Pop a Segment Register from the Stack (Note: CS cannot be sreg2 in this usage.)	
segment register FS, GS	0000 1111: 10 sreg3 001
POPF/POPFQ – Pop Stack into FLAGS/RFLAGS Register	
pop stack to FLAGS register	0101 0101 : 1001 1101
pop Stack to RFLAGS register	0100 1000 1001 1101
PUSH – Push Operand onto the Stack	
wordregister	0101 0101 : 0100 000B : 1111 1111 : 11 110 reg16
qwordregister	0100 W00B[S] : 1111 1111 : 11 110 reg64
wordregister (alternate encoding)	0101 0101 : 0100 000B : 0101 0 reg16
qwordregister (alternate encoding)	0100 W00B[S] : 0101 0 reg64
memory16	0101 0101 : 0100 000B : 1111 1111 : mod 110 r/m

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
memory64	0100 W00B^{S} : 1111 1111 : mod 110 r/m
immediate8	0110 1010 : imm8
immediate16	0101 0101 : 0110 1000 : imm16
immediate64	0110 1000 : imm64
PUSH – Push Segment Register onto the Stack	
segment register FS,GS	0000 1111: 10 sreg3 000
PUSHF/PUSHFD – Push Flags Register onto the Stack	1001 1100
RCL – Rotate thru Carry Left	
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
RCR – Rotate thru Carry Right	
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
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
RDMSR – Read from Model-Specific Register	
load ECX-specified register into EDX:EAX	0000 1111 : 0011 0010
RDPMC – Read Performance Monitoring Counters	
load ECX-specified performance counter into EDX:EAX	0000 1111 : 0011 0011
RDTSC – Read Time-Stamp Counter	
read time-stamp counter into EDX:EAX	0000 1111 : 0011 0001
REP INS – Input String	
REP LODS – Load String	
REP MOVS – Move String	
REP OUTS – Output String	
REP STOS – Store String	
REPE CMPS – Compare String	
REPE SCAS – Scan String	
REPNE CMPS – Compare String	
REPNE SCAS – Scan String	
RET – Return from Procedure (to same segment)	
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
RET – Return from Procedure (to other segment)	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
ROL – Rotate Left	
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
ROR – Rotate Right	
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
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
RSM – Resume from System Management Mode	0000 1111 : 1010 1010
SAL – Shift Arithmetic Left	same instruction as SHL
SAR – Shift Arithmetic Right	
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
SBB – Integer Subtraction with Borrow	
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
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
immediate8 to memory64	0100 10XB 1000 0011 : mod 011 r/m : imm8
SCAS/SCASB/SCASW/SCASD – Scan String	
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
SETcc – Byte Set on Condition	
register	0100 000B 0000 1111 : 1001 tttn : 11 000 reg
register	0100 0000 0000 1111 : 1001 tttn : 11 000 reg
memory	0100 00XB 0000 1111 : 1001 tttn : mod 000 r/m
memory	0100 0000 0000 1111 : 1001 tttn : mod 000 r/m
SGDT – Store Global Descriptor Table Register	0000 1111 : 0000 0001 : mod^{A} 000 r/m
SHL – Shift Left	
register by 1	0100 000B 1101 000w : 11 100 reg
byteregister by 1	0100 000B 1101 0000 : 11 100 bytereg
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
SHLD – Double Precision Shift Left	
register by immediate count	0100 0R0B 0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8
qwordregister by immediate8	0100 1R0B 0000 1111 : 1010 0100 : 11 qworddreg2 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
SHR – Shift Right	
register by 1	0100 000B 1101 000w : 11 101 reg
byteregister by 1	0100 000B 1101 0000 : 11 101 bytereg
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
SHRD – Double Precision Shift Right	
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
SIDT – Store Interrupt Descriptor Table Register	0000 1111 : 0000 0001 : mod[A] 001 r/m
SLDT – Store Local Descriptor Table Register	
to register	0100 000B 0000 1111 : 0000 0000 : 11 000 reg
to memory	0100 00XB 0000 1111 : 0000 0000 : mod 000 r/m

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
SMSW – Store Machine Status Word	
to register	0100 000B 0000 1111 : 0000 0001 : 11 100 reg
to memory	0100 00XB 0000 1111 : 0000 0001 : mod 100 r/m
STC – Set Carry Flag	1111 1001
STD – Set Direction Flag	1111 1101
STI – Set Interrupt Flag	1111 1011
STOS/STOSB/STOSW/STOSD/STOSQ – Store String Data	
store string data	1010 101w
store string data (RAX at address RDI)	0100 1000 1010 1011
STR – Store Task Register	
to register	0100 000B 0000 1111 : 0000 0000 : 11 001 reg
to memory	0100 00XB 0000 1111 : 0000 0000 : mod 001 r/m
SUB – Integer Subtraction	
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
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
SWAPGS – Swap GS Base Register	
GS base register value for value in MSR C0000102H	0000 1111 0000 0001 [this one incomplete]
SYSCALL – Fast System Call	
fast call to privilege level 0 system procedures	0000 1111 0000 0101
SYSRET – Return From Fast System Call	
return from fast system call	0000 1111 0000 0111
TEST – Logical Compare	
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
UD2 – Undefined instruction	0000 FFFF : 0000 1011
VERR – Verify a Segment for Reading	
register	0100 000B 0000 1111 : 0000 0000 : 11 100 reg
memory	0100 00XB 0000 1111 : 0000 0000 : mod 100 r/m
VERW – Verify a Segment for Writing	
register	0100 000B 0000 1111 : 0000 0000 : 11 101 reg
memory	0100 00XB 0000 1111 : 0000 0000 : mod 101 r/m
WAIT – Wait	1001 1011
WBINVD – Writeback and Invalidate Data Cache	0000 1111 : 0000 1001
WRMSR – Write to Model-Specific Register	
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
XADD – Exchange and Add	
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
XCHG – Exchange Register/Memory with Register	
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
XLAT/XLATB – Table Look-up Translation	
AL to byte DS:[(E)BX + unsigned AL]	1101 0111
AL to byte DS:[RBX + unsigned AL]	0100 1000 1101 0111
XOR – Logical Exclusive OR	
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
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

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

Instruction and Format	Encoding
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
Prefix Bytes	
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
CMPXCHG8B – Compare and Exchange 8 Bytes	
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
CMPXCHG8B/CMPXCHG16B – Compare and Exchange Bytes	
EDX:EAX with memory64	0000 1111 : 1100 0111 : mod 001 r/m
RDX:RAX with memory128	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-32 lists special encodings (instructions that do not follow the rules below).

1. 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
2. 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.
3. 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.
4. 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.

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)

gg	Granularity of Data
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
EMMS – Empty MMX technology state	0000 1111:01110111
MOVD – Move doubleword	
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
MOVQ – Move quadword	
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
PACKSSDW[1] – Pack dword to word data (signed with saturation)	
mmxreg2 to mmxreg1	0000 1111:0110 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 1011: mod mmxreg r/m
PACKSSWB[1] – Pack word to byte data (signed with saturation)	
mmxreg2 to mmxreg1	0000 1111:0110 0011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0011: mod mmxreg r/m
PACKUSWB[1] – Pack word to byte data (unsigned with saturation)	
mmxreg2 to mmxreg1	0000 1111:0110 0111: 11 mmxreg1 mmxreg2

Table B-19. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
memory to mmxreg	0000 1111:0110 0111: mod mmxreg r/m
PADD – Add with wrap-around	
mmxreg2 to mmxreg1	0000 1111: 1111 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1111 11gg: mod mmxreg r/m
PADDS – Add signed with saturation	
mmxreg2 to mmxreg1	0000 1111: 1110 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1110 11gg: mod mmxreg r/m
PADDUS – Add unsigned with saturation	
mmxreg2 to mmxreg1	0000 1111: 1101 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1101 11gg: mod mmxreg r/m
PAND – Bitwise And	
mmxreg2 to mmxreg1	0000 1111:1101 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 1011: mod mmxreg r/m
PANDN – Bitwise AndNot	
mmxreg2 to mmxreg1	0000 1111:1101 1111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 1111: mod mmxreg r/m
PCMPEQ – Packed compare for equality	
mmxreg1 with mmxreg2	0000 1111:0111 01gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:0111 01gg: mod mmxreg r/m
PCMPGT – Packed compare greater (signed)	
mmxreg1 with mmxreg2	0000 1111:0110 01gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:0110 01gg: mod mmxreg r/m
PMADDWD – Packed multiply add	
mmxreg2 to mmxreg1	0000 1111:1111 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1111 0101: mod mmxreg r/m
PMULHUW – Packed multiplication, store high word (unsigned)	
mmxreg2 to mmxreg1	0000 1111: 1110 0100: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1110 0100: mod mmxreg r/m

Table B-19. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
PMULHW – Packed multiplication, store high word	
mmxreg2 to mmxreg1	0000 1111:1110 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 0101: mod mmxreg r/m
PMULLW – Packed multiplication, store low word	
mmxreg2 to mmxreg1	0000 1111:1101 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 0101: mod mmxreg r/m
POR – Bitwise Or	
mmxreg2 to mmxreg1	0000 1111:1110 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 1011: mod mmxreg r/m
PSLL[2] – Packed shift left logical	
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
PSRA[2] – Packed shift right arithmetic	
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
PSRL[2] – Packed shift right logical	
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
PSUB – Subtract with wrap-around	
mmxreg2 from mmxreg1	0000 1111:1111 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1111 10gg: mod mmxreg r/m
PSUBS – Subtract signed with saturation	
mmxreg2 from mmxreg1	0000 1111:1110 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1110 10gg: mod mmxreg r/m

Table B-19. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
PSUBUS – Subtract unsigned with saturation	
mmxreg2 from mmxreg1	0000 1111:1101 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1101 10gg: mod mmxreg r/m
PUNPCKH – Unpack high data to next larger type	
mmxreg2 to mmxreg1	0000 1111:0110 10gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 10gg: mod mmxreg r/m
PUNPCKL – Unpack low data to next larger type	
mmxreg2 to mmxreg1	0000 1111:0110 00gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 00gg: mod mmxreg r/m
PXOR – Bitwise Xor	
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
CMOVcc – Conditional Move	
register2 to register1	0000 1111: 0100 tttn : 11 reg1 reg2
memory to register	0000 1111 : 0100 tttn : mod reg r/m
FCMOVcc – Conditional Move on EFLAG Register Condition Codes	
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)
FCOMI – Compare Real and Set EFLAGS	11011 011 : 11 110 ST(i)
FXRSTOR – Restore x87 FPU, MMX, SSE, and SSE2 State[1]	0000 1111:1010 1110: mod^{A} 001 r/m
FXSAVE – Save x87 FPU, MMX, SSE, and SSE2 State[1]	0000 1111:1010 1110: mod^{A} 000 r/m
SYSENTER – Fast System Call	0000 1111:0011 0100
SYSEXIT – 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
ADDPS—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
ADDSS—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
ANDNPS—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
ANDPS—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
CMPPS—Compare Packed Single-Precision Floating-Point Values	
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
CMPSS—Compare Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	1111 0011:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8

Table B-21. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
mem to xmmreg, imm8	1111 0011:0000 1111:1100 0010: mod xmmreg r/m: imm8
COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS	
xmmreg to xmmreg	0000 1111:0010 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0010 1111: mod xmmreg r/m
CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values	
mmreg to xmmreg	0000 1111:0010 1010:11 xmmreg1 mmreg1
mem to xmmreg	0000 1111:0010 1010: mod xmmreg r/m
CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0000 1111:0010 1101:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1101: mod mmreg r/m
CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value	
r32 to xmmreg1	1111 0011:0000 1111:00101010:11 xmmreg r32
mem to xmmreg	1111 0011:0000 1111:00101010: mod xmmreg r/m
CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer	
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
CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0000 1111:0010 1100:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1100: mod mmreg r/m
CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	1111 0011:0000 1111:0010 1100:11 r32 xmmreg1

Table B-21. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
mem to r32	1111 0011:0000 1111:0010 1100: mod r32 r/m
DIVPS—Divide Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1110: mod xmmreg r/m
DIVSS—Divide Scalar Single-Precision Floating-Point Values	
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
LDMXCSR—Load MXCSR Register State	
m32 to MXCSR	0000 1111:1010 1110:mod^{A} 010 mem
MAXPS—Return Maximum Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1111: mod xmmreg r/m
MAXSS—Return Maximum Scalar Double-Precision Floating-Point Value	
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
MINPS—Return Minimum Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1101: mod xmmreg r/m
MINSS—Return Minimum Scalar Double-Precision Floating-Point Value	
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

Table B-21. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0010 1000:11 xmmreg2 xmmreg1
mem to xmmreg1	0000 1111:0010 1000: mod xmmreg r/m
xmmreg1 to xmmreg2	0000 1111:0010 1001:11 xmmreg1 xmmreg2
xmmreg1 to mem	0000 1111:0010 1001: mod xmmreg r/m
MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low	
xmmreg to xmmreg	0000 1111:0001 0010:11 xmmreg1 xmmreg2
MOVHPS—Move High Packed Single-Precision Floating-Point Values	
mem to xmmreg	0000 1111:0001 0110: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0111: mod xmmreg r/m
MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High	
xmmreg to xmmreg	0000 1111:00010110:11 xmmreg1 xmmreg2
MOVLPS—Move Low Packed Single-Precision Floating-Point Values	
mem to xmmreg	0000 1111:0001 0010: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0011: mod xmmreg r/m
MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask	
xmmreg to r32	0000 1111:0101 0000:11 r32 xmmreg
MOVSS—Move Scalar Single-Precision Floating-Point Values	
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

Table B-21. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values	
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
MULPS—Multiply Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1001: mod xmmreg rm
MULSS—Multiply Scalar Single-Precision Floating-Point Values	
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
ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0110 mod xmmreg r/m
RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0011: mod xmmreg r/m
RCPSS—Compute Reciprocals of Scalar Single-Precision Floating-Point Value	
xmmreg to xmmreg	1111 0011:0000 1111:01010011:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:01010011: mod xmmreg r/m
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0010 mode xmmreg r/m

Table B-21. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
RSQRTSS—Compute Reciprocals of Square Roots of Scalar Single-Precision Floating-Point Value	
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
SHUFPS—Shuffle Packed Single-Precision Floating-Point Values	
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
SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 0001:11 xmmreg1 xmmreg 2
mem to xmmreg	0000 1111:0101 0001 mod xmmreg r/m
SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value	
xmmreg to xmmreg	1111 0011:0000 1111:0101 0001:11 xmmreg1 xmmreg 2
mem to xmmreg	1111 0011:0000 1111:0101 0001:mod xmmreg r/m
STMXCSR—Store MXCSR Register State	
MXCSR to mem	0000 1111:1010 1110:mod[A] 011 mem
SUBPS—Subtract Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1100:mod xmmreg r/m
SUBSS—Subtract Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg	1111 0011:0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1100:mod xmmreg r/m

Table B-21. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
UCOMISS—Unordered Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS	
xmmreg to xmmreg	0000 1111:0010 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0010 1110 mod xmmreg r/m
UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0001 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0001 0101 mod xmmreg r/m
UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0001 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0001 0100 mod xmmreg r/m
XORPS—Bitwise Logical XOR of Single-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0111 mod xmmreg r/m

Table B-22. Formats and Encodings of SSE Integer Instructions

Instruction and Format	Encoding
PAVGB/PAVGW—Average Packed Integers	
mmreg to mmreg	0000 1111:1110 0000:11 mmreg1 mmreg2
	0000 1111:1110 0011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 0000 mod mmreg r/m
	0000 1111:1110 0011 mod mmreg r/m
PEXTRW—Extract Word	
mmreg to reg32, imm8	0000 1111:1100 0101:11 r32 mmreg: imm8
PINSRW—Insert Word	
reg32 to mmreg, imm8	0000 1111:1100 0100:11 mmreg r32: imm8
m16 to mmreg, imm8	0000 1111:1100 0100 mod mmreg r/m: imm8

Table B-22. Formats and Encodings of SSE Integer Instructions (Contd.)

Instruction and Format	Encoding
PMAXSW—Maximum of Packed Signed Word Integers	
mmreg to mmreg	0000 1111:1110 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 1110 mod mmreg r/m
PMAXUB—Maximum of Packed Unsigned Byte Integers	
mmreg to mmreg	0000 1111:1101 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 1110 mod mmreg r/m
PMINSW—Minimum of Packed Signed Word Integers	
mmreg to mmreg	0000 1111:1110 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 1010 mod mmreg r/m
PMINUB—Minimum of Packed Unsigned Byte Integers	
mmreg to mmreg	0000 1111:1101 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 1010 mod mmreg r/m
PMOVMSKB—Move Byte Mask To Integer	
mmreg to reg32	0000 1111:1101 0111:11 r32 mmreg
PMULHUW—Multiply Packed Unsigned Integers and Store High Result	
mmreg to mmreg	0000 1111:1110 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 0100 mod mmreg r/m
PSADBW—Compute Sum of Absolute Differences	
mmreg to mmreg	0000 1111:1111 0110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 0110 mod mmreg r/m
PSHUFW—Shuffle Packed Words	
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
MASKMOVQ—Store Selected Bytes of Quadword	
mmreg to mmreg	0000 1111:1111 0111:11 mmreg1 mmreg2
MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg to mem	0000 1111:0010 1011: mod xmmreg r/m
MOVNTQ—Store Quadword Using Non-Temporal Hint	
mmreg to mem	0000 1111:1110 0111: mod mmreg r/m
PREFETCHT0—Prefetch Temporal to All Cache Levels	0000 1111:0001 1000:mod[A] 001 mem
PREFETCHT1—Prefetch Temporal to First Level Cache	0000 1111:0001 1000:mod[A] 010 mem
PREFETCHT2—Prefetch Temporal to Second Level Cache	0000 1111:0001 1000:mod[A] 011 mem
PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels	0000 1111:0001 1000:mod[A] 000 mem
SFENCE—Store Fence	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
ADDPD—Add Packed Double-Precision Floating-Point Values	
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
ADDSD—Add Scalar Double-Precision Floating-Point Values	
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
ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	0110 0110:0000 1111:0101 0101:11 xmmreg1 xmmreg2

Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0101 0101: mod xmmreg r/m
ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values	
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
CMPPD—Compare Packed Double-Precision Floating-Point Values	
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
CMPSD—Compare Scalar Double-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	1111 0010:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110 010:0000 1111:1100 0010: mod xmmreg r/m: imm8
COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS	
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
CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values	
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
CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
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

Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value	
r32 to xmmreg1	1111 0010:0000 1111:0010 1010:11 xmmreg r32
mem to xmmreg	1111 0010:0000 1111:0010 1010: mod xmmreg r/m
CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	1111 0010:0000 1111:0010 1101:11 r32 xmmreg
mem to r32	1111 0010:0000 1111:0010 1101: mod r32 r/m
CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0110 0110:0000 1111:0010 1100:11 mmreg xmmreg
mem to mmreg	0110 0110:0000 1111:0010 1100: mod mmreg r/m
CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	1111 0010:0000 1111:0010 1100:11 r32 xmmreg
mem to r32	1111 0010:0000 1111:0010 1100: mod r32 r/m
CVTPD2PS—Covert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	0110 0110:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1010: mod xmmreg r/m
CVTPS2PD—Covert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1010: mod xmmreg r/m

Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
CVTSD2SS—Covert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value	
xmmreg to xmmreg	1111 0010:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1010: mod xmmreg r/m
CVTSS2SD—Covert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	1111 0011:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:00001 111:0101 1010: mod xmmreg r/m
CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
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
CVTTPD2DQ—Convert With Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
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
CVTDQ2PD—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values	
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
CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to xmmreg	0110 0110:0000 1111:0101 1011:11 xmmreg1 xmmreg2

Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0101 1011: mod xmmreg r/m
CVTTPS2DQ—Convert With Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
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
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1011: mod xmmreg r/m
DIVPD—Divide Packed Double-Precision Floating-Point Values	
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
DIVSD—Divide Scalar Double-Precision Floating-Point Values	
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
MAXPD—Return Maximum Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	0110 0110:0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1111: mod xmmreg r/m
MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	1111 0010:0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1111: mod xmmreg r/m

Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
MINPD—Return Minimum Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	0110 0110:0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1101: mod xmmreg r/m
MINSD—Return Minimum Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	1111 0010:0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1101: mod xmmreg r/m
MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0010 1001:11 xmmreg2 xmmreg1
mem to xmmreg1	0110 0110:0000 1111:0010 1001: mod xmmreg r/m
xmmreg1 to xmmreg2	0110 0110:0000 1111:0010 1000:11 xmmreg1 xmmreg2
xmmreg1 to mem	0110 0110:0000 1111:0010 1000: mod xmmreg r/m
MOVHPD—Move High Packed Double-Precision Floating-Point Values	
mem to xmmreg	0110 0110:0000 1111:0001 0111: mod xmmreg r/m
xmmreg to mem	0110 0110:0000 1111:0001 0110: mod xmmreg r/m
MOVLPD—Move Low Packed Double-Precision Floating-Point Values	
mem to xmmreg	0110 0110:0000 1111:0001 0011: mod xmmreg r/m
xmmreg to mem	0110 0110:0000 1111:0001 0010: mod xmmreg r/m
MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask	
xmmreg to r32	0110 0110:0000 1111:0101 0000:11 r32 xmmreg
MOVSD—Move Scalar Double-Precision Floating-Point Values	
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

Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
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
MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values	
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
MULPD—Multiply Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	0110 0110:0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1001: mod xmmreg rm
MULSD—Multiply Scalar Double-Precision Floating-Point Values	
xmmreg to xmmreg	1111 0010:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:00001111:01011001: mod xmmreg r/m
ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values	
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
SHUFPD—Shuffle Packed Double-Precision Floating-Point Values	
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

Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values	
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
SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value	
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
SUBPD—Subtract Packed Double-Precision Floating-Point Values	
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
SUBSD—Subtract Scalar Double-Precision Floating-Point Values	
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
UCOMISD—Unordered Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS	
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
UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values	
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

Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values	
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
XORPD—Bitwise Logical OR of Double-Precision Floating-Point Values	
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

Table B-26. Formats and Encodings of SSE2 Integer Instructions

Instruction and Format	Encoding
MOVD—Move Doubleword	
reg to xmmeg	0110 0110:0000 1111:0110 1110: 11 xmmreg reg
reg from xmmreg	0110 0110:0000 1111:0111 1110: 11 xmmreg reg
mem to xmmreg	0110 0110:0000 1111:0110 1110: mod xmmreg r/m
mem from xmmreg	0110 0110:0000 1111:0111 1110: mod xmmreg r/m
MOVDQA—Move Aligned Double Quadword	
xmmreg to xmmreg	0110 0110:0000 1111:0110 1111:11 xmmreg1 xmmreg2
	0110 0110:0000 1111:0111 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 1111: mod xmmreg r/m
mem from xmmreg	0110 0110:0000 1111:0111 1111: mod xmmreg r/m
MOVDQU—Move Unaligned Double Quadword	
xmmreg to xmmreg	1111 0011:0000 1111:0110 1111:11 xmmreg1 xmmreg2
	1111 0011:0000 1111:0111 1111:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0110 1111: mod xmmreg r/m

Table B-26. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
mem from xmmreg	1111 0011:0000 1111:0111 1111: mod xmmreg r/m
MOVQ2DQ—Move Quadword from MMX to XMM Register	
mmreg to xmmreg	1111 0011:0000 1111:1101 0110:11 mmreg1 mmreg2
MOVDQ2Q—Move Quadword from XMM to MMX Register	
xmmreg to mmreg	1111 0010:0000 1111:1101 0110:11 mmreg1 mmreg2
MOVQ—Move Quadword	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0111 1110: 11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	0110 0110:0000 1111:1101 0110: 11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0111 1110: mod xmmreg r/m
mem from xmmreg	0110 0110:0000 1111:1101 0110: mod xmmreg r/m
PACKSSDW[1]—Pack Dword To Word Data (signed with saturation)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:0110 1011: mod xmmreg r/m
PACKSSWB—Pack Word To Byte Data (signed with saturation)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 0011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:0110 0011: mod xmmreg r/m
PACKUSWB—Pack Word To Byte Data (unsigned with saturation)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 0111: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:0110 0111: mod xmmreg r/m
PADDQ—Add Packed Quadword Integers	
mmreg to mmreg	0000 1111:1101 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 0100: mod mmreg r/m

Table B-26. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
xmmreg to xmmreg	0110 0110:0000 1111:1101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1101 0100: mod xmmreg r/m
PADD—Add With Wrap-around	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1111 11gg: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111: 1111 11gg: mod xmmreg r/m
PADDS—Add Signed With Saturation	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1110 11gg: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111: 1110 11gg: mod xmmreg r/m
PADDUS—Add Unsigned With Saturation	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1101 11gg: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111: 1101 11gg: mod xmmreg r/m
PAND—Bitwise And	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1101 1011: mod xmmreg r/m
PANDN—Bitwise AndNot	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1111: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1101 1111: mod xmmreg r/m
PAVGB—Average Packed Integers	
xmmreg to xmmreg	0110 0110:0000 1111:11100 000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100000 mod xmmreg r/m
PAVGW—Average Packed Integers	
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

Table B-26. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PCMPEQ—Packed Compare For Equality	
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
PCMPGT—Packed Compare Greater (signed)	
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
PEXTRW—Extract Word	
xmmreg to reg32, imm8	0110 0110:0000 1111:1100 0101:11 r32 xmmreg: imm8
PINSRW—Insert Word	
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
PMADDWD—Packed Multiply Add	
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
PMAXSW—Maximum of Packed Signed Word Integers	
xmmreg to xmmreg	0110 0110:0000 1111:1110 1110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101110 mod xmmreg r/m
PMAXUB—Maximum of Packed Unsigned Byte Integers	
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

Table B-26. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PMINSW—Minimum of Packed Signed Word Integers	
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
PMINUB—Minimum of Packed Unsigned Byte Integers	
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
PMOVMSKB—Move Byte Mask To Integer	
xmmreg to reg32	0110 0110:0000 1111:1101 0111:11 r32 xmmreg
PMULHUW—Packed multiplication, store high word (unsigned)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0100: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 0100: mod xmmreg r/m
PMULHW—Packed Multiplication, store high word	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0101: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 0101: mod xmmreg r/m
PMULLW—Packed Multiplication, store low word	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 0101: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1101 0101: mod xmmreg r/m
PMULUDQ—Multiply Packed Unsigned Doubleword Integers	
mmreg to mmreg	0000 1111:1111 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 0100: mod mmreg r/m
xmmreg to xmmreg	0110 0110:00001111:1111 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:00001111:1111 0100: mod xmmreg r/m

Table B-26. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
POR—Bitwise Or	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1011: 11 xmmreg1 xmmreg2
xmemory to xmmreg	0110 0110:0000 1111:1110 1011: mod xmmreg r/m
PSADBW—Compute Sum of Absolute Differences	
xmmreg to xmmreg	0110 0110:0000 1111:1111 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1111 0110: mod xmmreg r/m
PSHUFLW—Shuffle Packed Low Words	
xmmreg to xmmreg, imm8	1111 0010:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0010:0000 1111:0111 0000:11 mod xmmreg r/m: imm8
PSHUFHW—Shuffle Packed High Words	
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
PSHUFD—Shuffle Packed Doublewords	
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
PSLLDQ—Shift Double Quadword Left Logical	
xmmreg, imm8	0110 0110:0000 1111:0111 0011:11 111 xmmreg: imm8
PSLL—Packed Shift Left Logical	
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

Table B-26. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 110 xmmreg: imm8
PSRA—Packed Shift Right Arithmetic	
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
PSRLDQ—Shift Double Quadword Right Logical	
xmmreg, imm8	0110 0110:00001111:01110011:11 011 xmmreg: imm8
PSRL—Packed Shift Right Logical	
xmmxreg1 by xmmxreg2	0110 0110:0000 1111:1101 00gg: 11 xmmreg1 xmmreg2
xmmxreg by memory	0110 0110:0000 1111:1101 00gg: mod xmmreg r/m
xmmxreg by immediate	0110 0110:0000 1111:0111 00gg: 11 010 xmmreg: imm8
PSUBQ—Subtract Packed Quadword Integers	
mmreg to mmreg	0000 1111:11111 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
PSUB—Subtract With Wrap-around	
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
PSUBS—Subtract Signed With Saturation	
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

Table B-26. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PSUBUS—Subtract Unsigned With Saturation	
xmmreg2 from xmmreg1	0000 1111:1101 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0000 1111:1101 10gg: mod xmmreg r/m
PUNPCKH—Unpack High Data To Next Larger Type	
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
PUNPCKHQDQ—Unpack High Data	
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
PUNPCKL—Unpack Low Data To Next Larger Type	
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
PUNPCKLQDQ—Unpack Low Data	
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
PXOR—Bitwise Xor	
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

Table B-27. Format and Encoding of SSE2 Cacheability Instructions

Instruction and Format	Encoding
MASKMOVDQU—Store Selected Bytes of Double Quadword	
xmmreg to xmmreg	0110 0110:0000 1111:1111 0111:11 xmmreg1 xmmreg2
CLFLUSH—Flush Cache Line	
mem	0000 1111:1010 1110:mod r/m
MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg to mem	0110 0110:0000 1111:0010 1011: mod xmmreg r/m
MOVNTDQ—Store Double Quadword Using Non-Temporal Hint	
xmmreg to mem	0110 0110:0000 1111:1110 0111: mod xmmreg r/m
MOVNTI—Store Doubleword Using Non-Temporal Hint	
reg to mem	0000 1111:1100 0011: mod reg r/m
PAUSE—Spin Loop Hint	1111 0011:1001 0000
LFENCE—Load Fence	0000 1111:1010 1110: 11 101 000
MFENCE—Memory Fence	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.11.

Table B-28. Formats and Encodings of SSE3 Floating-Point Instructions

Instruction and Format	Encoding
ADDSUBPD—Add /Sub packed DP FP numbers from XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	01100110:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11010000: mod xmmreg r/m
ADDSUBPS—Add /Sub packed SP FP numbers from XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	11110010:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:11010000: mod xmmreg r/m
HADDPD—Add horizontally packed DP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	01100110:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111100: mod xmmreg r/m
HADDPS—Add horizontally packed SP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	11110010:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01111100: mod xmmreg r/m
HSUBPD—Sub horizontally packed DP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	01100110:00001111:01111101:11 xmmreg1 xmmreg2

Table B-28. Formats and Encodings of SSE3 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
mem to xmmreg	01100110:00001111:01111101: mod xmmreg r/m
HSUBPS—Sub horizontally packed SP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	11110010:00001111:01111101:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01111101: mod xmmreg r/m

Table B-29. Formats and Encodings for SSE3 Event Management Instructions

Instruction and Format	Encoding
MONITOR—Set up a linear address range to be monitored by hardware	
eax, ecx, edx	0000 1111 : 0000 0001:11 001 000
MWAIT—Wait until write-back store performed within the range specified by the instruction MONITOR	
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
FISTTP—Store ST in int16 (chop) and pop	
m16int	11011 111 : mod^A 001 r/m
FISTTP—Store ST in int32 (chop) and pop	
m32int	11011 011 : mod^A 001 r/m
FISTTP—Store ST in int64 (chop) and pop	
m64int	11011 101 : mod^A 001 r/m
LDDQU—Load unaligned integer 128-bit	
xmm, m128	11110010:00001111:11110000: mod^A xmmreg r/m
MOVDDUP—Move 64 bits representing one DP data from XMM2/Mem to XMM1 and duplicate	
xmmreg2 to xmmreg1	11110010:00001111:00010010:11 xmmreg1 xmmreg2

Table B-30. Formats and Encodings for SSE3 Integer and Move Instructions (Contd.)

Instruction and Format	Encoding
mem to xmmreg	11110010:00001111:00010010: mod xmmreg r/m
MOVSHDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate high	
xmmreg2 to xmmreg1	11110011:00001111:00010110:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:00010110: mod xmmreg r/m
MOVSLDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate low	
xmmreg2 to xmmreg1	11110011:00001111:00010010:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:00010010: mod xmmreg r/m

B.10 SSSE3 FORMATS AND ENCODING TABLE

The tables in this section provide SSSE3 formats and encodings. Some SSSE3 instructions require a mandatory prefix (66H) as part of the three-byte opcode. These prefixes are included in the table below.

Table B-31. Formats and Encodings for SSSE3 Instructions

Instruction and Format	Encoding
PABSB—Packed Absolute Value Bytes	
mmreg to mmreg	0000 1111:0011 1000: 0001 1100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0001 1100: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1100: mod xmmreg r/m
PABSD—Packed Absolute Value Double Words	
mmreg to mmreg	0000 1111:0011 1000: 0001 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0001 1110: mod mmreg r/m

Table B-31. Formats and Encodings for SSSE3 Instructions (Contd.)

Instruction and Format	Encoding
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1110: mod xmmreg r/m
PABSW—Packed Absolute Value Words	
mmreg to mmreg	0000 1111:0011 1000: 0001 1101:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0001 1101: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1101: mod xmmreg r/m
PALIGNR—Packed Align Right	
mmreg to mmreg	0000 1111:0011 1010: 0000 1111:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1010: 0000 1111: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1010: 0000 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0000 1111: mod xmmreg r/m
PHADDD—Packed Horizontal Add Double Words	
mmreg to mmreg	0000 1111:0011 1000: 0000 0010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0010: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0010: mod xmmreg r/m
PHADDSW—Packed Horizontal Add and Saturate	
mmreg to mmreg	0000 1111:0011 1000: 0000 0011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0011: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0011: mod xmmreg r/m

Table B-31. Formats and Encodings for SSSE3 Instructions (Contd.)

Instruction and Format	Encoding
PHADDW—Packed Horizontal Add Words	
mmreg to mmreg	0000 1111:0011 1000: 0000 0001:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0001: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0001: mod xmmreg r/m
PHSUBD—Packed Horizontal Subtract Double Words	
mmreg to mmreg	0000 1111:0011 1000: 0000 0110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0110: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0110: mod xmmreg r/m
PHSUBSW—Packed Horizontal Subtract and Saturate	
mmreg to mmreg	0000 1111:0011 1000: 0000 0111:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0111: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0111: mod xmmreg r/m
PHSUBW—Packed Horizontal Subtract Words	
mmreg to mmreg	0000 1111:0011 1000: 0000 0101:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0101: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0101: mod xmmreg r/m
PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes	
mmreg to mmreg	0000 1111:0011 1000: 0000 0100:11 mmreg1 mmreg2

Table B-31. Formats and Encodings for SSSE3 Instructions (Contd.)

Instruction and Format	Encoding
mem to mmreg	0000 1111:0011 1000: 0000 0100: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0100: mod xmmreg r/m
PMULHRSW—Packed Multiply HIgn with Round and Scale	
mmreg to mmreg	0000 1111:0011 1000: 0000 1011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1011: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1011: mod xmmreg r/m
PSHUFB—Packed Shuffle Bytes	
mmreg to mmreg	0000 1111:0011 1000: 0000 0000:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0000: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0000: mod xmmreg r/m
PSIGNB—Packed Sign Bytes	
mmreg to mmreg	0000 1111:0011 1000: 0000 1000:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1000: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1000: mod xmmreg r/m
PSIGND—Packed Sign Double Words	
mmreg to mmreg	0000 1111:0011 1000: 0000 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1010: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1010: mod xmmreg r/m

Table B-31. Formats and Encodings for SSSE3 Instructions (Contd.)

Instruction and Format	Encoding
PSIGNW—Packed Sign Words	
mmreg to mmreg	0000 1111:0011 1000: 0000 1001:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1001: mod mmreg r/m
xmmreg to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1001: mod xmmreg r/m

B.11 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-32. Special Case Instructions Promoted Using REX.W

Instruction and Format	Encoding
CMOVcc—Conditional Move	
register2 to register1	0100 0R0B 0000 1111: 0100 tttn : 11 reg1 reg2
qwordregister2 to qwordregister1	0100 1R0B 0000 1111: 0100 tttn : 11 qwordreg1 qwordreg2
memory to register	0100 0RXB 0000 1111 : 0100 tttn : mod reg r/m
memory64 to qwordregister	0100 1RXB 0000 1111 : 0100 tttn : mod qwordreg r/m
CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	0100 0R0B 1111 0010:0000 1111:0010 1101:11 r32 xmmreg
xmmreg to r64	0100 1R0B 1111 0010:0000 1111:0010 1101:11 r64 xmmreg
mem64 to r32	0100 0R0XB 1111 0010:0000 1111:0010 1101: mod r32 r/m
mem64 to r64	0100 1RXB 1111 0010:0000 1111:0010 1101: mod r64 r/m

Table B-32. Special Case Instructions Promoted Using REX.W (Contd.)

Instruction and Format	Encoding
CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value	
r32 to xmmreg1	0100 0R0B 1111 0011:0000 1111:0010 1010:11 xmmreg r32
r64 to xmmreg1	0100 1R0B 1111 0011:0000 1111:0010 1010:11 xmmreg r64
mem to xmmreg	0100 0RXB 1111 0011:0000 1111:0010 1010: mod xmmreg r/m
mem64 to xmmreg	0100 1RXB 1111 0011:0000 1111:0010 1010: mod xmmreg r/m
CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value	
r32 to xmmreg1	0100 0R0B 1111 0010:0000 1111:0010 1010:11 xmmreg r32
r64 to xmmreg1	0100 1R0B 1111 0010:0000 1111:0010 1010:11 xmmreg r64
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
CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer	
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
CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	0100 0R0B 11110010:00001111:00101100:11 r32 xmmreg

Table B-32. Special Case Instructions Promoted Using REX.W (Contd.)

Instruction and Format	Encoding
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
CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer	
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
MOVD/MOVQ—Move doubleword	
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
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

Table B-32. Special Case Instructions Promoted Using REX.W (Contd.)

Instruction and Format	Encoding
MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask	
xmmreg to r32	0100 0R0B 0000 1111:0101 0000:11 r32 xmmreg
xmmreg to r64	0100 1R0B 00001111:01010000:11 r64 xmmreg
PEXTRW—Extract Word	
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
PINSRW—Insert Word	
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
m16 to xmmreg, imm8	0100 1RXB 0110 0110 0000 1111:1100 0100 mod xmmreg r/m: imm8
PMOVMSKB—Move Byte Mask To Integer	
mmreg to reg32	0100 0RXB 0000 1111:1101 0111:11 r32 mmreg

Table B-32. Special Case Instructions Promoted Using REX.W (Contd.)

Instruction and Format	Encoding
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.12 FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS

Table B-33 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-33. General Floating-Point Instruction Formats

<table>
<tr><td></td><td colspan="10">Instruction</td><td colspan="2" rowspan="2">Optional Fields</td></tr>
<tr><td></td><td colspan="4">First Byte</td><td colspan="6">Second Byte</td></tr>
<tr><td>1</td><td>11011</td><td colspan="2">OPA</td><td>1</td><td colspan="2">mod</td><td>1</td><td colspan="2">OPB</td><td>r/m</td><td>s-i-b</td><td>disp</td></tr>
<tr><td>2</td><td>11011</td><td colspan="2">MF</td><td>OPA</td><td colspan="2">mod</td><td colspan="3">OPB</td><td>r/m</td><td>s-i-b</td><td>disp</td></tr>
<tr><td>3</td><td>11011</td><td>d</td><td>P</td><td>OPA</td><td>1</td><td>1</td><td>OPB</td><td colspan="2">R</td><td>ST(i)</td><td></td><td></td></tr>
<tr><td>4</td><td>11011</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="3">OP</td><td></td><td></td></tr>
<tr><td>5</td><td>11011</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="3">OP</td><td></td><td></td></tr>
<tr><td></td><td>15–11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2 1 0</td><td></td><td></td></tr>
</table>

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

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-34 shows the formats and encodings of the floating-point instructions.

Table B-34. Floating-Point Instruction Formats and Encodings

Instruction and Format	Encoding
F2XM1 – Compute $2^{ST(0)} - 1$	11011 001 : 1111 0000
FABS – Absolute Value	11011 001 : 1110 0001
FADD – Add	
ST(0) ← ST(0) + 32-bit memory	11011 000 : mod 000 r/m
ST(0) ← ST(0) + 64-bit memory	11011 100 : mod 000 r/m
ST(d) ← ST(0) + ST(i)	11011 d00 : 11 000 ST(i)
FADDP – Add and Pop	
ST(0) ← ST(0) + ST(i)	11011 110 : 11 000 ST(i)
FBLD – Load Binary Coded Decimal	11011 111 : mod 100 r/m
FBSTP – Store Binary Coded Decimal and Pop	11011 111 : mod 110 r/m
FCHS – Change Sign	11011 001 : 1110 0000
FCLEX – Clear Exceptions	11011 011 : 1110 0010
FCOM – Compare Real	
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)
FCOMP – Compare Real and Pop	
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)
FCOMPP – Compare Real and Pop Twice	11011 110 : 11 011 001
FCOMIP – Compare Real, Set EFLAGS, and Pop	11011 111 : 11 110 ST(i)
FCOS – Cosine of ST(0)	11011 001 : 1111 1111
FDECSTP – Decrement Stack-Top Pointer	11011 001 : 1111 0110
FDIV – Divide	
ST(0) ← ST(0) ÷ 32-bit memory	11011 000 : mod 110 r/m

Table B-34. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
ST(0) ← ST(0) ÷ 64-bit memory	11011 100 : mod 110 r/m
ST(d) ← ST(0) ÷ ST(i)	11011 d00 : 1111 R ST(i)
FDIVP – Divide and Pop	
ST(0) ← ST(0) ÷ ST(i)	11011 110 : 1111 1 ST(i)
FDIVR – Reverse Divide	
ST(0) ← 32-bit memory ÷ ST(0)	11011 000 : mod 111 r/m
ST(0) ← 64-bit memory ÷ ST(0)	11011 100 : mod 111 r/m
ST(d) ← ST(i) ÷ ST(0)	11011 d00 : 1111 R ST(i)
FDIVRP – Reverse Divide and Pop	
ST(0) ¨ ST(i) ÷ ST(0)	11011 110 : 1111 0 ST(i)
FFREE – Free ST(i) Register	11011 101 : 1100 0 ST(i)
FIADD – Add Integer	
ST(0) ← ST(0) + 16-bit memory	11011 110 : mod 000 r/m
ST(0) ← ST(0) + 32-bit memory	11011 010 : mod 000 r/m
FICOM – Compare Integer	
16-bit memory	11011 110 : mod 010 r/m
32-bit memory	11011 010 : mod 010 r/m
FICOMP – Compare Integer and Pop	
16-bit memory	11011 110 : mod 011 r/m
32-bit memory	11011 010 : mod 011 r/m
FIDIV	
ST(0) ← ST(0) ÷ 16-bit memory	11011 110 : mod 110 r/m
ST(0) ← ST(0) ÷ 32-bit memory	11011 010 : mod 110 r/m
FIDIVR	
ST(0) ← 16-bit memory ÷ ST(0)	11011 110 : mod 111 r/m
ST(0) ← 32-bit memory ÷ ST(0)	11011 010 : mod 111 r/m
FILD – Load Integer	
16-bit memory	11011 111 : mod 000 r/m
32-bit memory	11011 011 : mod 000 r/m
64-bit memory	11011 111 : mod 101 r/m

Table B-34. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
FIMUL	
$ST(0) \leftarrow ST(0) \times$ 16-bit memory	11011 110 : mod 001 r/m
$ST(0) \leftarrow ST(0) \times$ 32-bit memory	11011 010 : mod 001 r/m
FINCSTP – Increment Stack Pointer	11011 001 : 1111 0111
FINIT – Initialize Floating-Point Unit	
FIST – Store Integer	
16-bit memory	11011 111 : mod 010 r/m
32-bit memory	11011 011 : mod 010 r/m
FISTP – Store Integer and Pop	
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
FISUB	
$ST(0) \leftarrow ST(0)$ - 16-bit memory	11011 110 : mod 100 r/m
$ST(0) \leftarrow ST(0)$ - 32-bit memory	11011 010 : mod 100 r/m
FISUBR	
$ST(0) \leftarrow$ 16-bit memory $-$ ST(0)	11011 110 : mod 101 r/m
$ST(0) \leftarrow$ 32-bit memory $-$ ST(0)	11011 010 : mod 101 r/m
FLD – Load Real	
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)
FLD1 – Load $+$1.0 into ST(0)	11011 001 : 1110 1000
FLDCW – Load Control Word	11011 001 : mod 101 r/m
FLDENV – Load FPU Environment	11011 001 : mod 100 r/m
FLDL2E – Load $\log_2(\varepsilon)$ into ST(0)	11011 001 : 1110 1010
FLDL2T – Load $\log_2(10)$ into ST(0)	11011 001 : 1110 1001
FLDLG2 – Load $\log_{10}(2)$ into ST(0)	11011 001 : 1110 1100
FLDLN2 – Load $\log_\varepsilon(2)$ into ST(0)	11011 001 : 1110 1101
FLDPI – Load π into ST(0)	11011 001 : 1110 1011

Table B-34. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
FLDZ – Load +0.0 into ST(0)	11011 001 : 1110 1110
FMUL – Multiply	
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)
FMULP – Multiply	
ST(i) ← ST(0) × ST(i)	11011 110 : 1100 1 ST(i)
FNOP – No Operation	11011 001 : 1101 0000
FPATAN – Partial Arctangent	11011 001 : 1111 0011
FPREM – Partial Remainder	11011 001 : 1111 1000
FPREM1 – Partial Remainder (IEEE)	11011 001 : 1111 0101
FPTAN – Partial Tangent	11011 001 : 1111 0010
FRNDINT – Round to Integer	11011 001 : 1111 1100
FRSTOR – Restore FPU State	11011 101 : mod 100 r/m
FSAVE – Store FPU State	11011 101 : mod 110 r/m
FSCALE – Scale	11011 001 : 1111 1101
FSIN – Sine	11011 001 : 1111 1110
FSINCOS – Sine and Cosine	11011 001 : 1111 1011
FSQRT – Square Root	11011 001 : 1111 1010
FST – Store Real	
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)
FSTCW – Store Control Word	11011 001 : mod 111 r/m
FSTENV – Store FPU Environment	11011 001 : mod 110 r/m
FSTP – Store Real and Pop	
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)
FSTSW – Store Status Word into AX	11011 111 : 1110 0000

Table B-34. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
FSTSW – Store Status Word into Memory	11011 101 : mod 111 r/m
FSUB – Subtract	
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)
FSUBP – Subtract and Pop	
ST(0) ← ST(0) – ST(i)	11011 110 : 1110 1 ST(i)
FSUBR – Reverse Subtract	
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)
FSUBRP – Reverse Subtract and Pop	
ST(i) ← ST(i) – ST(0)	11011 110 : 1110 0 ST(i)
FTST – Test	11011 001 : 1110 0100
FUCOM – Unordered Compare Real	11011 101 : 1110 0 ST(i)
FUCOMP – Unordered Compare Real and Pop	11011 101 : 1110 1 ST(i)
FUCOMPP – Unordered Compare Real and Pop Twice	11011 010 : 1110 1001
FUCOMI – Unorderd Compare Real and Set EFLAGS	11011 011 : 11 101 ST(i)
FUCOMIP – Unorderd Compare Real, Set EFLAGS, and Pop	11011 111 : 11 101 ST(i)
FXAM – Examine	11011 001 : 1110 0101
FXCH – Exchange ST(0) and ST(i)	11011 001 : 1100 1 ST(i)
FXTRACT – Extract Exponent and Significand	11011 001 : 1111 0100
FYL2X – ST(1) $\times$ $\log_2$(ST(0))	11011 001 : 1111 0001
FYL2XP1 – ST(1) $\times$ $\log_2$(ST(0) + 1.0)	11011 001 : 1111 1001
FWAIT – Wait until FPU Ready	1001 1011

B.13 VMX INSTRUCTIONS

Table B-35 describes virtual-machine extensions (VMX).

Table B-35. Encodings for VMX Instructions

Instruction and Format	Encoding
VMCALL—Call to VM Monitor	
Call VMM: causes VM exit.	00001111 00000001 11000001
VMCLEAR—Clear Virtual-Machine Control Structure	
mem32:VMCS_data_ptr	01100110 00001111 11000111: mod 110 r/m
mem64:VMCS_data_ptr	01100110 00001111 11000111: mod 110 r/m
VMLAUNCH—Launch Virtual Machine	
Launch VM managed by Current_VMCS	00001111 00000001 11000010
VMRESUME—Resume Virtual Machine	
Resume VM managed by Current_VMCS	00001111 00000001 11000011
VMPTRLD—Load Pointer to Virtual-Machine Control Structure	
mem32 to Current_VMCS_ptr	00001111 11000111: mod 110 r/m
mem64 to Current_VMCS_ptr	00001111 11000111: mod 110 r/m
VMPTRST—Store Pointer to Virtual-Machine Control Structure	
Current_VMCS_ptr to mem32	00001111 11000111: mod 111 r/m
Current_VMCS_ptr to mem64	00001111 11000111: mod 111 r/m
VMREAD—Read Field from Virtual-Machine Control Structure	
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
VMWRITE—Write Field to Virtual-Machine Control Structure	
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

Table B-35. Encodings for VMX Instructions

Instruction and Format	Encoding
VMXOFF—Leave VMX Operation	
Leave VMX.	00001111 00000001 11000100
VMXON—Enter VMX Operation	
Enter VMX.	11110011 000011111 11000111: mod 110 r/m

APPENDIX C
INTEL® 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, SSE3, and SSSE3 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:

<intrin_op>	Indicates the intrinsics basic operation; for example, add for addition and sub for subtraction
<suffix>	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
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.

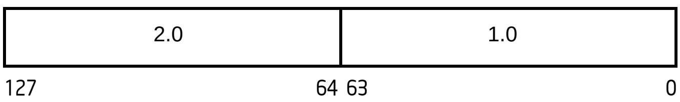
The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

```
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:

data_type	Is the return data type, which can be either void, int, __m64, __m128, __m128d, or __m128i. Only the _mm_empty intrinsic returns void.
intrinsic_name	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.
parameters	Represents the parameters required by each intrinsic.

C.1 SIMPLE INTRINSICS

NOTE

For detailed descriptions of the intrinsics in Table C-1, see the corresponding mnemonic in Chapter 3 in the “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A”, or Chapter 4, “Instruction Set Reference, N-Z” in the “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B”.

Table C-1. Simple Intrinsics

Mnemonic	Intrinsic
ADDPD	__m128d _mm_add_pd(__m128d a, __m128d b)
ADDPS	__m128 _mm_add_ps(__m128 a, __m128 b)
ADDSD	__m128d _mm_add_sd(__m128d a, __m128d b)
ADDSS	__m128 _mm_add_ss(__m128 a, __m128 b)
ADDSUBPD	__m128d _mm_addsub_pd(__m128d a, __m128d b)
ADDSUBPS	__m128 _mm_addsub_ps(__m128 a, __m128 b)
ANDNPD	__m128d _mm_andnot_pd(__m128d a, __m128d b)
ANDNPS	__m128 _mm_andnot_ps(__m128 a, __m128 b)
ANDPD	__m128d _mm_and_pd(__m128d a, __m128d b)
ANDPS	__m128 _mm_and_ps(__m128 a, __m128 b)
CLFLUSH	void _mm_clflush(void const *p)
CMPPD	__m128d _mm_cmpeq_pd(__m128d a, __m128d b)
	__m128d _mm_cmplt_pd(__m128d a, __m128d b)
	__m128d _mm_cmple_pd(__m128d a, __m128d b)
	__m128d _mm_cmpgt_pd(__m128d a, __m128d b)
	__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)
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)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	__m128 _mm_cmpord_ps(__m128 a, __m128 b)
	__m128 _mm_cmpunord_ps(__m128 a, __m128 b)
	__m128 _mm_cmpnle_ps(__m128 a, __m128 b)
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)
	__m128 _mm_cmpneq_sd(__m128d a, __m128d b)
	__m128 _mm_cmpnlt_sd(__m128d a, __m128d b)
	__m128d _mm_cmpnle_sd(__m128d a, __m128d b)
	__m128d _mm_cmpngt_sd(__m128d a, __m128d b)
	__m128d _mm_cmpnge_sd(__m128d a, __m128d b)
	__m128d _mm_cmpord_sd(__m128d a, __m128d b)
	__m128d _mm_cmpunord_sd(__m128d a, __m128d b)
CMPSS	__m128 _mm_cmpeq_ss(__m128 a, __m128 b)
	__m128 _mm_cmplt_ss(__m128 a, __m128 b)
	__m128 _mm_cmple_ss(__m128 a, __m128 b)
	__m128 _mm_cmpgt_ss(__m128 a, __m128 b)
	__m128 _mm_cmpge_ss(__m128 a, __m128 b)
	__m128 _mm_cmpneq_ss(__m128 a, __m128 b)
	__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)
	__m128 _mm_cmpnle_ss(__m128 a, __m128 b)
	__m128 _mm_cmpngt_ss(__m128 a, __m128 b)
	__m128 _mm_cmpnge_ss(__m128 a, __m128 b)
	__m128 _mm_cmpord_ss(__m128 a, __m128 b)
	__m128 _mm_cmpunord_ss(__m128 a, __m128 b)
COMISD	int _mm_comieq_sd(__m128d a, __m128d b)
	int _mm_comilt_sd(__m128d a, __m128d b)
	int _mm_comile_sd(__m128d a, __m128d b)
	int _mm_comigt_sd(__m128d a, __m128d b)
	int _mm_comige_sd(__m128d a, __m128d b)
	int _mm_comineq_sd(__m128d a, __m128d b)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
COMISS	int _mm_comieq_ss(__m128 a, __m128 b)
	int _mm_comilt_ss(__m128 a, __m128 b)
	int _mm_comile_ss(__m128 a, __m128 b)
	int _mm_comigt_ss(__m128 a, __m128 b)
	int _mm_comige_ss(__m128 a, __m128 b)
	int _mm_comineq_ss(__m128 a, __m128 b)
CVTDQ2PD	__m128d _mm_cvtepi32_pd(__m128i a)
CVTDQ2PS	__m128 _mm_cvtepi32_ps(__m128i a)
CVTPD2DQ	__m128i _mm_cvtpd_epi32(__m128d a)
CVTPD2PI	__m64 _mm_cvtpd_pi32(__m128d a)
CVTPD2PS	__m128 _mm_cvtpd_ps(__m128d a)
CVTPI2PD	__m128d _mm_cvtpi32_pd(__m64 a)
CVTPI2PS	__m128 _mm_cvt_pi2ps(__m128 a, __m64 b) __m128 _mm_cvtpi32_ps(__m128 a, __m64 b)
CVTPS2DQ	__m128i _mm_cvtps_epi32(__m128 a)
CVTPS2PD	__m128d _mm_cvtps_pd(__m128 a)
CVTPS2PI	__m64 _mm_cvt_ps2pi(__m128 a) __m64 _mm_cvtps_pi32(__m128 a)
CVTSD2SI	int _mm_cvtsd_si32(__m128d a)
CVTSD2SS	__m128 _mm_cvtsd_ss(__m128 a, __m128d b)
CVTSI2SD	__m128d _mm_cvtsi32_sd(__m128d a, int b)
CVTSI2SS	__m128 _mm_cvt_si2ss(__m128 a, int b) __m128 _mm_cvtsi32_ss(__m128a, int b)
CVTSS2SD	__m128d _mm_cvtss_sd(__m128d a, __m128 b)
CVTSS2SI	int _mm_cvt_ss2si(__m128 a) int _mm_cvtss_si32(__m128 a)
CVTTPD2DQ	__m128i _mm_cvttpd_epi32(__m128d a)
CVTTPD2PI	__m64 _mm_cvttpd_pi32(__m128d a)
CVTTPS2DQ	__m128i _mm_cvttps_epi32(__m128 a)
CVTTPS2PI	__m64 _mm_cvtt_ps2pi(__m128 a) __m64 _mm_cvttps_pi32(__m128 a)
CVTTSD2SI	int _mm_cvttsd_si32(__m128d a)
CVTTSS2SI	int _mm_cvtt_ss2si(__m128 a) int _mm_cvttss_si32(__m128 a)
	__m64 _mm_cvtsi32_si64(int i)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
	int _mm_cvtsi64_si32(__m64 m)
DIVPD	__m128d _mm_div_pd(__m128d a, __m128d b)
DIVPS	__m128 _mm_div_ps(__m128 a, __m128 b)
DIVSD	__m128d _mm_div_sd(__m128d a, __m128d b)
DIVSS	__m128 _mm_div_ss(__m128 a, __m128 b)
EMMS	void _mm_empty()
HADDPD	__m128d _mm_hadd_pd(__m128d a, __m128d b)
HADDPS	__m128 _mm_hadd_ps(__m128 a, __m128 b)
HSUBPD	__m128d _mm_hsub_pd(__m128d a, __m128d b)
HSUBPS	__m128 _mm_hsub_ps(__m128 a, __m128 b)
LDDQU	__m128i _mm_lddqu_si128(__m128i const *p)
LDMXCSR	__mm_setcsr(unsigned int i)
LFENCE	void _mm_lfence(void)
MASKMOVDQU	void _mm_maskmoveu_si128(__m128i d, __m128i n, char *p)
MASKMOVQ	void _mm_maskmove_si64(__m64 d, __m64 n, char *p)
MAXPD	__m128d _mm_max_pd(__m128d a, __m128d b)
MAXPS	__m128 _mm_max_ps(__m128 a, __m128 b)
MAXSD	__m128d _mm_max_sd(__m128d a, __m128d b)
MAXSS	__m128 _mm_max_ss(__m128 a, __m128 b)
MFENCE	void _mm_mfence(void)
MINPD	__m128d _mm_min_pd(__m128d a, __m128d b)
MINPS	__m128 _mm_min_ps(__m128 a, __m128 b)
MINSD	__m128d _mm_min_sd(__m128d a, __m128d b)
MINSS	__m128 _mm_min_ss(__m128 a, __m128 b)
MONITOR	void _mm_monitor(void const *p, unsigned extensions, unsigned hints)
MOVAPD	__m128d _mm_load_pd(double * p)
	void_mm_store_pd(double *p, __m128d a)
MOVAPS	__m128 _mm_load_ps(float * p)
	void_mm_store_ps(float *p, __m128 a)
MOVD	__m128i _mm_cvtsi32_si128(int a)
	int _mm_cvtsi128_si32(__m128i a)
	__m64 _mm_cvtsi32_si64(int a)
	int _mm_cvtsi64_si32(__m64 a)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
MOVDDUP	__m128d _mm_movedup_pd(__m128d a) __m128d _mm_loaddup_pd(double const * dp)
MOVDQA	__m128i _mm_load_si128(__m128i * p)
	void_mm_store_si128(__m128i *p, __m128i a)
MOVDQU	__m128i _mm_loadu_si128(__m128i * p)
	void_mm_storeu_si128(__m128i *p, __m128i a)
MOVDQ2Q	__m64 _mm_movepi64_pi64(__m128i a)
MOVHLPS	__m128 _mm_movehl_ps(__m128 a, __m128 b)
MOVHPD	__m128d _mm_loadh_pd(__m128d a, double * p)
	void _mm_storeh_pd(double * p, __m128d a)
MOVHPS	__m128 _mm_loadh_pi(__m128 a, __m64 * p)
	void _mm_storeh_pi(__m64 * p, __m128 a)
MOVLPD	__m128d _mm_loadl_pd(__m128d a, double * p)
	void _mm_storel_pd(double * p, __m128d a)
MOVLPS	__m128 _mm_loadl_pi(__m128 a, __m64 *p)
	void_mm_storel_pi(__m64 * p, __m128 a)
MOVLHPS	__m128 _mm_movelh_ps(__m128 a, __m128 b)
MOVMSKPD	int _mm_movemask_pd(__m128d a)
MOVMSKPS	int _mm_movemask_ps(__m128 a)
MOVNTDQ	void_mm_stream_si128(__m128i * p, __m128i a)
MOVNTPD	void_mm_stream_pd(double * p, __m128d a)
MOVNTPS	void_mm_stream_ps(float * p, __m128 a)
MOVNTI	void_mm_stream_si32(int * p, int a)
MOVNTQ	void_mm_stream_pi(__m64 * p, __m64 a)
MOVQ	__m128i _mm_loadl_epi64(__m128i * p)
	void_mm_storel_epi64(_m128i * p, __m128i a)
	__m128i _mm_move_epi64(__m128i a)
MOVQ2DQ	__m128i _mm_movpi64_epi64(__m64 a)
MOVSD	__m128d _mm_load_sd(double * p)
	void_mm_store_sd(double * p, __m128d a)
	__m128d _mm_move_sd(__m128d a, __m128d b)
MOVSHDUP	__m128 _mm_movehdup_ps(__m128 a)
MOVSLDUP	__m128 _mm_moveldup_ps(__m128 a)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
MOVSS	__m128 _mm_load_ss(float * p)
	void_mm_store_ss(float * p, __m128 a)
	__m128 _mm_move_ss(__m128 a, __m128 b)
MOVUPD	__m128d _mm_loadu_pd(double * p)
	void_mm_storeu_pd(double *p, __m128d a)
MOVUPS	__m128 _mm_loadu_ps(float * p)
	void_mm_storeu_ps(float *p, __m128 a)
MULPD	__m128d _mm_mul_pd(__m128d a, __m128d b)
MULPS	__m128 _mm_mul_ss(__m128 a, __m128 b)
MULSD	__m128d _mm_mul_sd(__m128d a, __m128d b)
MULSS	__m128 _mm_mul_ss(__m128 a, __m128 b)
MWAIT	void _mm_mwait(unsigned extensions, unsigned hints)
ORPD	__m128d _mm_or_pd(__m128d a, __m128d b)
ORPS	__m128 _mm_or_ps(__m128 a, __m128 b)
PABSB	__m64 _mm_abs_pi8 (__m64 a)
	__m128i _mm_abs_epi8 (__m128i a)
PABSD	__m64 _mm_abs_pi32 (__m64 a)
	__m128i _mm_abs_epi32 (__m128i a)
PABSW	__m64 _mm_abs_pi16 (__m64 a)
	__m128i _mm_abs_epi16 (__m128i a)
PACKSSWB	__m128i _mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSWB	__m64 _mm_packs_pi16(__m64 m1, __m64 m2)
PACKSSDW	__m128i _mm_packs_epi32 (__m128i m1, __m128i m2)
PACKSSDW	__m64 _mm_packs_pi32 (__m64 m1, __m64 m2)
PACKUSWB	__m128i _mm_packus_epi16(__m128i m1, __m128i m2)
PACKUSWB	__m64 _mm_packs_pu16(__m64 m1, __m64 m2)
PADDB	__m128i _mm_add_epi8(__m128i m1, __m128i m2)
PADDB	__m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDW	__m128i _mm_addw_epi16(__m128i m1, __m128i m2)
PADDW	__m64 _mm_addw_pi16(__m64 m1, __m64 m2)
PADDD	__m128i _mm_add_epi32(__m128i m1, __m128i m2)
PADDD	__m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDQ	__m128i _mm_add_epi64(__m128i m1, __m128i m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PADDQ	__m64 _mm_add_si64(__m64 m1, __m64 m2)
PADDSB	__m128i _mm_adds_epi8(__m128i m1, __m128i m2)
PADDSB	__m64 _mm_adds_pi8(__m64 m1, __m64 m2)
PADDSW	__m128i _mm_adds_epi16(__m128i m1, __m128i m2)
PADDSW	__m64 _mm_adds_pi16(__m64 m1, __m64 m2)
PADDUSB	__m128i _mm_adds_epu8(__m128i m1, __m128i m2)
PADDUSB	__m64 _mm_adds_pu8(__m64 m1, __m64 m2)
PADDUSW	__m128i _mm_adds_epu16(__m128i m1, __m128i m2)
PADDUSW	__m64 _mm_adds_pu16(__m64 m1, __m64 m2)
PALIGNR	__m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)
	__m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)
PAND	__m128i _mm_and_si128(__m128i m1, __m128i m2)
PAND	__m64 _mm_and_si64(__m64 m1, __m64 m2)
PANDN	__m128i _mm_andnot_si128(__m128i m1, __m128i m2)
PANDN	__m64 _mm_andnot_si64(__m64 m1, __m64 m2)
PAUSE	void _mm_pause(void)
PAVGB	__m128i _mm_avg_epu8(__m128i a, __m128i b)
PAVGB	__m64 _mm_avg_pu8(__m64 a, __m64 b)
PAVGW	__m128i _mm_avg_epu16(__m128i a, __m128i b)
PAVGW	__m64 _mm_avg_pu16(__m64 a, __m64 b)
PCMPEQB	__m128i _mm_cmpeq_epi8(__m128i m1, __m128i m2)
PCMPEQB	__m64 _mm_cmpeq_pi8(__m64 m1, __m64 m2)
PCMPEQW	__m128i _mm_cmpeq_epi16 (__m128i m1, __m128i m2)
PCMPEQW	__m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD	__m128i _mm_cmpeq_epi32(__m128i m1, __m128i m2)
PCMPEQD	__m64 _mm_cmpeq_pi32(__m64 m1, __m64 m2)
PCMPGTB	__m128i _mm_cmpgt_epi8 (__m128i m1, __m128i m2)
PCMPGTB	__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW	__m128i _mm_cmpgt_epi16(__m128i m1, __m128i m2)
PCMPGTW	__m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)
PCMPGTD	__m128i _mm_cmpgt_epi32(__m128i m1, __m128i m2)
PCMPGTD	__m64 _mm_cmpgt_pi32(__m64 m1, __m64 m2)
PEXTRW	int _mm_extract_epi16(__m128i a, int n)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PEXTRW	int _mm_extract_pi16(__m64 a, int n)
PHADDD	__m64 _mm_hadd_pi32 (__m64 a, __m64 b)
	__m128i _mm_hadd_epi32 (__m128i a, __m128i b)
PHADDSW	__m64 _mm_hadds_pi16 (__m64 a, __m64 b)
	__m128i _mm_hadds_epi16 (__m128i a, __m128i b)
PHADDW	__m64 _mm_hadd_pi16 (__m64 a, __m64 b)
	__m128i _mm_hadd_epi16 (__m128i a, __m128i b)
PHSUBD	__m64 _mm_hsub_pi32 (__m64 a, __m64 b)
	__m128i _mm_hsub_epi32 (__m128i a, __m128i b)
PHSUBSW	__m64 _mm_hsubs_pi16 (__m64 a, __m64 b)
	__m128i _mm_hsubs_epi16 (__m128i a, __m128i b)
PHSUBW	__m64 _mm_hsub_pi16 (__m64 a, __m64 b)
	__m128i _mm_hsub_epi16 (__m128i a, __m128i b)
PINSRW	__m128i _mm_insert_epi16(__m128i a, int d, int n)
PINSRW	__m64 _mm_insert_pi16(__m64 a, int d, int n)
PMADDUBSW	__m64 _mm_maddubs_pi16 (__m64 a, __m64 b)
	__m128i _mm_maddubs_epi16 (__m128i a, __m128i b)
PMADDWD	__m128i _mm_madd_epi16(__m128i m1 __m128i m2)
PMADDWD	__m64 _mm_madd_pi16(__m64 m1, __m64 m2)
PMAXSW	__m128i _mm_max_epi16(__m128i a, __m128i b)
PMAXSW	__m64 _mm_max_pi16(__m64 a, __m64 b)
PMAXUB	__m128i _mm_max_epu8(__m128i a, __m128i b)
PMAXUB	__m64 _mm_max_pu8(__m64 a, __m64 b)
PMINSW	__m128i _mm_min_epi16(__m128i a, __m128i b)
PMINSW	__m64 _mm_min_pi16(__m64 a, __m64 b)
PMINUB	__m128i _mm_min_epu8(__m128i a, __m128i b)
PMINUB	__m64 _mm_min_pu8(__m64 a, __m64 b)
PMOVMSKB	int _mm_movemask_epi8(__m128i a)
PMOVMSKB	int _mm_movemask_pi8(__m64 a)
PMULHRSW	__m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
	__m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
PMULHUW	__m128i _mm_mulhi_epu16(__m128i a, __m128i b)
PMULHUW	__m64 _mm_mulhi_pu16(__m64 a, __m64 b)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PMULHW	__m128i _mm_mulhi_epi16(__m128i m1, __m128i m2)
PMULHW	__m64 _mm_mulhi_pi16(__m64 m1, __m64 m2)
PMULLW	__m128i _mm_mullo_epi16(__m128i m1, __m128i m2)
PMULLW	__m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
PMULUDQ	__m64 _mm_mul_su32(__m64 m1, __m64 m2)
	__m128i _mm_mul_epu32(__m128i m1, __m128i m2)
POR	__m64 _mm_or_si64(__m64 m1, __m64 m2)
POR	__m128i _mm_or_si128(__m128i m1, __m128i m2)
PREFETCHh	void _mm_prefetch(char *a, int sel)
PSADBW	__m128i _mm_sad_epu8(__m128i a, __m128i b)
PSADBW	__m64 _mm_sad_pu8(__m64 a, __m64 b)
PSHUFB	__m64 _mm_shuffle_pi8 (__m64 a, __m64 b)
	__m128i _mm_shuffle_epi8 (__m128i a, __m128i b)
PSHUFD	__m128i _mm_shuffle_epi32(__m128i a, int n)
PSHUFHW	__m128i _mm_shufflehi_epi16(__m128i a, int n)
PSHUFLW	__m128i _mm_shufflelo_epi16(__m128i a, int n)
PSHUFW	__m64 _mm_shuffle_pi16(__m64 a, int n)
PSIGNB	__m64 _mm_sign_pi8 (__m64 a, __m64 b)
	__m128i _mm_sign_epi8 (__m128i a, __m128i b)
PSIGND	__m64 _mm_sign_pi32 (__m64 a, __m64 b)
	__m128i _mm_sign_epi32 (__m128i a, __m128i b)
PSIGNW	__m64 _mm_sign_pi16 (__m64 a, __m64 b)
	__m128i _mm_sign_epi16 (__m128i a, __m128i b)
PSLLW	__m128i _mm_sll_epi16(__m128i m, __m128i count)
PSLLW	__m128i _mm_slli_epi16(__m128i m, int count)
PSLLW	__m64 _mm_sll_pi16(__m64 m, __m64 count)
	__m64 _mm_slli_pi16(__m64 m, int count)
PSLLD	__m128i _mm_slli_epi32(__m128i m, int count)
	__m128i _mm_sll_epi32(__m128i m, __m128i count)
PSLLD	__m64 _mm_slli_pi32(__m64 m, int count)
	__m64 _mm_sll_pi32(__m64 m, __m64 count)
PSLLQ	__m64 _mm_sll_si64(__m64 m, __m64 count)
	__m64 _mm_slli_si64(__m64 m, int count)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PSLLQ	__m128i _mm_sll_epi64(__m128i m, __m128i count)
	__m128i _mm_slli_epi64(__m128i m, int count)
PSLLDQ	__m128i _mm_slli_si128(__m128i m, int imm)
PSRAW	__m128i _mm_sra_epi16(__m128i m, __m128i count)
	__m128i _mm_srai_epi16(__m128i m, int count)
PSRAW	__m64 _mm_sra_pi16(__m64 m, __m64 count)
	__m64 _mm_srai_pi16(__m64 m, int count)
PSRAD	__m128i _mm_sra_epi32 (__m128i m, __m128i count)
	__m128i _mm_srai_epi32 (__m128i m, int count)
PSRAD	__m64 _mm_sra_pi32 (__m64 m, __m64 count)
	__m64 _mm_srai_pi32 (__m64 m, int count)
PSRLW	_m128i _mm_srl_epi16 (__m128i m, __m128i count)
	__m128i _mm_srli_epi16 (__m128i m, int count)
	__m64 _mm_srl_pi16 (__m64 m, __m64 count)
	__m64 _mm_srli_pi16(__m64 m, int count)
PSRLD	__m128i _mm_srl_epi32 (__m128i m, __m128i count)
	__m128i _mm_srli_epi32 (__m128i m, int count)
PSRLD	__m64 _mm_srl_pi32 (__m64 m, __m64 count)
	__m64 _mm_srli_pi32 (__m64 m, int count)
PSRLQ	__m128i _mm_srl_epi64 (__m128i m, __m128i count)
	__m128i _mm_srli_epi64 (__m128i m, int count)
PSRLQ	__m64 _mm_srl_si64 (__m64 m, __m64 count)
	__m64 _mm_srli_si64 (__m64 m, int count)
PSRLDQ	__m128i _mm_srli_si128(__m128i m, int imm)
PSUBB	__m128i _mm_sub_epi8(__m128i m1, __m128i m2)
PSUBB	__m64 _mm_sub_pi8(__m64 m1, __m64 m2)
PSUBW	__m128i _mm_sub_epi16(__m128i m1, __m128i m2)
PSUBW	__m64 _mm_sub_pi16(__m64 m1, __m64 m2)
PSUBD	__m128i _mm_sub_epi32(__m128i m1, __m128i m2)
PSUBD	__m64 _mm_sub_pi32(__m64 m1, __m64 m2)
PSUBQ	__m128i _mm_sub_epi64(__m128i m1, __m128i m2)
PSUBQ	__m64 _mm_sub_si64(__m64 m1, __m64 m2)
PSUBSB	__m128i _mm_subs_epi8(__m128i m1, __m128i m2)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
PSUBSB	__m64 _mm_subs_pi8(__m64 m1, __m64 m2)
PSUBSW	__m128i _mm_subs_epi16(__m128i m1, __m128i m2)
PSUBSW	__m64 _mm_subs_pi16(__m64 m1, __m64 m2)
PSUBUSB	__m128i _mm_subs_epu8(__m128i m1, __m128i m2)
PSUBUSB	__m64 _mm_subs_pu8(__m64 m1, __m64 m2)
PSUBUSW	__m128i _mm_subs_epu16(__m128i m1, __m128i m2)
PSUBUSW	__m64 _mm_subs_pu16(__m64 m1, __m64 m2)
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 m1, __m128i m2)
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)
PXOR	__m64 _mm_xor_si64(__m64 m1, __m64 m2)
PXOR	__m128i _mm_xor_si128(__m128i m1, __m128i m2)
RCPPS	__m128 _mm_rcp_ps(__m128 a)
RCPSS	__m128 _mm_rcp_ss(__m128 a)
RSQRTPS	__m128 _mm_rsqrt_ps(__m128 a)
RSQRTSS	__m128 _mm_rsqrt_ss(__m128 a)
SFENCE	void_mm_sfence(void)
SHUFPD	__m128d _mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)
SHUFPS	__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)
SQRTPD	__m128d _mm_sqrt_pd(__m128d a)
SQRTPS	__m128 _mm_sqrt_ps(__m128 a)
SQRTSD	__m128d _mm_sqrt_sd(__m128d a)

Table C-1. Simple Intrinsics (Contd.)

Mnemonic	Intrinsic
SQRTSS	__m128 _mm_sqrt_ss(__m128 a)
STMXCSR	_mm_getcsr(void)
SUBPD	__m128d _mm_sub_pd(__m128d a, __m128d b)
SUBPS	__m128 _mm_sub_ps(__m128 a, __m128 b)
SUBSD	__m128d _mm_sub_sd(__m128d a, __m128d b)
SUBSS	__m128 _mm_sub_ss(__m128 a, __m128 b)
UCOMISD	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)
UCOMISS	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)
UNPCKHPD	__m128d _mm_unpackhi_pd(__m128d a, __m128d b)
UNPCKHPS	__m128 _mm_unpackhi_ps(__m128 a, __m128 b)
UNPCKLPD	__m128d _mm_unpacklo_pd(__m128d a, __m128d b)
UNPCKLPS	__m128 _mm_unpacklo_ps(__m128 a, __m128 b)
XORPD	__m128d _mm_xor_pd(__m128d a, __m128d b)
XORPS	__m128 _mm_xor_ps(__m128 a, __m128 b)

C.2 COMPOSITE INTRINSICS

Table C-2. Composite Intrinsics

Mnemonic	Intrinsic
(composite)	__m128i _mm_set_epi64(__m64 q1, __m64 q0)
(composite)	__m128i _mm_set_epi32(int i3, int i2, int i1, int i0)
(composite)	__m128i _mm_set_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w1,short w0)
(composite)	__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)
(composite)	__m128i _mm_set1_epi64(__m64 q)
(composite)	__m128i _mm_set1_epi32(int a)
(composite)	__m128i _mm_set1_epi16(short a)
(composite)	__m128i _mm_set1_epi8(char a)
(composite)	__m128i _mm_setr_epi64(__m64 q1, __m64 q0)
(composite)	__m128i _mm_setr_epi32(int i3, int i2, int i1, int i0)
(composite)	__m128i _mm_setr_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w, short w0)
(composite)	__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)
(composite)	__m128i _mm_setzero_si128()
(composite)	__m128 _mm_set_ps1(float w) __m128 _mm_set1_ps(float w)
(composite)	__m128cmm_set1_pd(double w)
(composite)	__m128d _mm_set_sd(double w)
(composite)	__m128d _mm_set_pd(double z, double y)
(composite)	__m128 _mm_set_ps(float z, float y, float x, float w)
(composite)	__m128d _mm_setr_pd(double z, double y)
(composite)	__m128 _mm_setr_ps(float z, float y, float x, float w)
(composite)	__m128d _mm_setzero_pd(void)
(composite)	__m128 _mm_setzero_ps(void)

Table C-2. Composite Intrinsics (Contd.)

Mnemonic	Intrinsic
MOVSD + shuffle	__m128d _mm_load_pd(double * p) __m128d _mm_load1_pd(double *p)
MOVSS + shuffle	__m128 _mm_load_ps1(float * p) __m128 _mm_load1_ps(float *p)
MOVAPD + shuffle	__m128d _mm_loadr_pd(double * p)
MOVAPS + shuffle	__m128 _mm_loadr_ps(float * p)
MOVSD + shuffle	void _mm_store1_pd(double *p, __m128d a)
MOVSS + shuffle	void _mm_store_ps1(float * p, __m128 a) void _mm_store1_ps(float *p, __m128 a)
MOVAPD + shuffle	_mm_storer_pd(double * p, __m128d a)
MOVAPS + shuffle	_mm_storer_ps(float * p, __m128 a)

INDEX FOR VOLUMES 2A & 2B

D

E

F

N

O

P

Z

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.
Rm 709, Shaanxi
Zhongda Int'l Bldg
No.30 Nandajie Street
Xian AX710002
China
Fax:(86 29) 7203356

Intel Corp.
Rm 2710, Metropolian
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
Crowne 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
21 Nehru Place
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 Cossettini 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

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
Rexcdale 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