

AAA—ASCII Adjust After Addition

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
37	AAA	NP	Invalid	Valid	ASCII adjust AL after addition.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Adjusts the sum of two unpacked BCD values to create an unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two unpacked BCD values and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the addition produces a decimal carry, the AH register increments by 1, and the CF and AF flags are set. If there was no decimal carry, the CF and AF flags are cleared and the AH register is unchanged. In either case, bits 4 through 7 of the AL register are set to 0.

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;
  ELSE
    IF ((AL AND 0FH) > 9) or (AF = 1)
      THEN
        AL ← AL + 6;
        AH ← AH + 1;
        AF ← 1;
        CF ← 1;
        AL ← AL AND 0FH;
      ELSE
        AF ← 0;
        CF ← 0;
        AL ← AL AND 0FH;
      FI;
    FI;
  
```

Flags Affected

The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are set to 0. The OF, SF, ZF, and PF flags are undefined.

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

AAD—ASCII Adjust AX Before Division

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
D5 0A	AAD	NP	Invalid	Valid	ASCII adjust AX before division.
D5 <i>ib</i>	AAD imm8	NP	Invalid	Valid	Adjust AX before division to number base <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Adjusts two unpacked BCD digits (the least-significant digit in the AL register and the most-significant digit in the AH register) so that a division operation performed on the result will yield a correct unpacked BCD value. The AAD instruction is only useful when it precedes a DIV instruction that divides (binary division) the adjusted value in the AX register by an unpacked BCD value.

The AAD instruction sets the value in the AL register to $(AL + (10 * AH))$, and then clears the AH register to 00H. The value in the AX register is then equal to the binary equivalent of the original unpacked two-digit (base 10) number in registers AH and AL.

The generalized version of this instruction allows adjustment of two unpacked digits of any number base (see the "Operation" section below), by setting the *imm8* byte to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAD mnemonic is interpreted by all assemblers to mean adjust ASCII (base 10) values. To adjust values in another number base, the instruction must be hand coded in machine code (D5 *imm8*).

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;
  ELSE
    tempAL ← AL;
    tempAH ← AH;
    AL ← (tempAL + (tempAH * imm8)) AND FFH;
    (* imm8 is set to 0AH for the AAD mnemonic.*)
    AH ← 0;
FI;
```

The immediate value (*imm8*) is taken from the second byte of the instruction.

Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register; the OF, AF, and CF flags are undefined.

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

AAM—ASCII Adjust AX After Multiply

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
D4 0A	AAM	NP	Invalid	Valid	ASCII adjust AX after multiply.
D4 <i>ib</i>	AAM imm8	NP	Invalid	Valid	Adjust AX after multiply to number base <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked (base 10) BCD values. The AX register is the implied source and destination operand for this instruction. The AAM instruction is only useful when it follows an MUL instruction that multiplies (binary multiplication) two unpacked BCD values and stores a word result in the AX register. The AAM instruction then adjusts the contents of the AX register to contain the correct 2-digit unpacked (base 10) BCD result.

The generalized version of this instruction allows adjustment of the contents of the AX to create two unpacked digits of any number base (see the “Operation” section below). Here, the *imm8* byte is set to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAM mnemonic is interpreted by all assemblers to mean adjust to ASCII (base 10) values. To adjust to values in another number base, the instruction must be hand coded in machine code (D4 *imm8*).

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;
ELSE
  tempAL ← AL;
  AH ← tempAL / imm8; (* imm8 is set to 0AH for the AAM mnemonic *)
  AL ← tempAL MOD imm8;
FI;
```

The immediate value (*imm8*) is taken from the second byte of the instruction.

Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register. The OF, AF, and CF flags are undefined.

Protected Mode Exceptions

- #DE If an immediate value of 0 is used.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

AAS—ASCII Adjust AL After Subtraction

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
3F	AAS	NP	Invalid	Valid	ASCII adjust AL after subtraction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one unpacked BCD value from another and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the subtraction produced a decimal carry, the AH register decrements by 1, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top four bits set to 0.

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;
  ELSE
    IF ((AL AND OFH) > 9) or (AF = 1)
      THEN
        AX ← AX - 6;
        AH ← AH - 1;
        AF ← 1;
        CF ← 1;
        AL ← AL AND OFH;
      ELSE
        CF ← 0;
        AF ← 0;
        AL ← AL AND OFH;
      FI;
    FI;
  
```

Flags Affected

The AF and CF flags are set to 1 if there is a decimal borrow; otherwise, they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

ADC—Add with Carry

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
14 <i>ib</i>	ADC AL, <i>imm8</i>	I	Valid	Valid	Add with carry <i>imm8</i> to AL.
15 <i>iw</i>	ADC AX, <i>imm16</i>	I	Valid	Valid	Add with carry <i>imm16</i> to AX.
15 <i>id</i>	ADC EAX, <i>imm32</i>	I	Valid	Valid	Add with carry <i>imm32</i> to EAX.
REX.W + 15 <i>id</i>	ADC RAX, <i>imm32</i>	I	Valid	N.E.	Add with carry <i>imm32</i> sign extended to 64-bits to RAX.
80 /2 <i>ib</i>	ADC r/m8, <i>imm8</i>	MI	Valid	Valid	Add with carry <i>imm8</i> to r/m8.
REX + 80 /2 <i>ib</i>	ADC r/m8*, <i>imm8</i>	MI	Valid	N.E.	Add with carry <i>imm8</i> to r/m8.
81 /2 <i>iw</i>	ADC r/m16, <i>imm16</i>	MI	Valid	Valid	Add with carry <i>imm16</i> to r/m16.
81 /2 <i>id</i>	ADC r/m32, <i>imm32</i>	MI	Valid	Valid	Add with CF <i>imm32</i> to r/m32.
REX.W + 81 /2 <i>id</i>	ADC r/m64, <i>imm32</i>	MI	Valid	N.E.	Add with CF <i>imm32</i> sign extended to 64-bits to r/m64.
83 /2 <i>ib</i>	ADC r/m16, <i>imm8</i>	MI	Valid	Valid	Add with CF sign-extended <i>imm8</i> to r/m16.
83 /2 <i>ib</i>	ADC r/m32, <i>imm8</i>	MI	Valid	Valid	Add with CF sign-extended <i>imm8</i> into r/m32.
REX.W + 83 /2 <i>ib</i>	ADC r/m64, <i>imm8</i>	MI	Valid	N.E.	Add with CF sign-extended <i>imm8</i> into r/m64.
10 /r	ADC r/m8, r8	MR	Valid	Valid	Add with carry byte register to r/m8.
REX + 10 /r	ADC r/m8*, r8*	MR	Valid	N.E.	Add with carry byte register to r/m64.
11 /r	ADC r/m16, r16	MR	Valid	Valid	Add with carry r16 to r/m16.
11 /r	ADC r/m32, r32	MR	Valid	Valid	Add with CF r32 to r/m32.
REX.W + 11 /r	ADC r/m64, r64	MR	Valid	N.E.	Add with CF r64 to r/m64.
12 /r	ADC r8, r/m8	RM	Valid	Valid	Add with carry r/m8 to byte register.
REX + 12 /r	ADC r8*, r/m8*	RM	Valid	N.E.	Add with carry r/m64 to byte register.
13 /r	ADC r16, r/m16	RM	Valid	Valid	Add with carry r/m16 to r16.
13 /r	ADC r32, r/m32	RM	Valid	Valid	Add with CF r/m32 to r32.
REX.W + 13 /r	ADC r64, r/m64	RM	Valid	N.E.	Add with CF r/m64 to 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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AE/X/AE/RAX	imm8	NA	NA

Description

Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag 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, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADC 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 carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC 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

$\text{DEST} \leftarrow \text{DEST} + \text{SRC} + \text{CF};$

Intel C/C++ Compiler Intrinsic Equivalent

```
ADC:     extern unsigned char _addcarry_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *sum_out);
ADC:     extern unsigned char _addcarry_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *sum_out);
ADC:     extern unsigned char _addcarry_u32(unsigned char c_in, unsigned int src1, unsigned char int, unsigned int *sum_out);
ADC:     extern unsigned char _addcarry_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);
```

Flags Affected

The OF, SF, ZF, AF, CF, 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 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 <i>ib</i>	ADD AL, <i>imm8</i>	I	Valid	Valid	Add <i>imm8</i> to AL.
05 <i>iw</i>	ADD AX, <i>imm16</i>	I	Valid	Valid	Add <i>imm16</i> to AX.
05 <i>id</i>	ADD EAX, <i>imm32</i>	I	Valid	Valid	Add <i>imm32</i> to EAX.
REX.W + 05 <i>id</i>	ADD RAX, <i>imm32</i>	I	Valid	N.E.	Add <i>imm32 sign-extended to 64-bits</i> to RAX.
80 /0 <i>ib</i>	ADD <i>r/m8, imm8</i>	MI	Valid	Valid	Add <i>imm8</i> to <i>r/m8</i> .
REX + 80 /0 <i>ib</i>	ADD <i>r/m8</i> [*] , <i>imm8</i>	MI	Valid	N.E.	Add <i>sign-extended imm8</i> to <i>r/m64</i> .
81 /0 <i>iw</i>	ADD <i>r/m16, imm16</i>	MI	Valid	Valid	Add <i>imm16</i> to <i>r/m16</i> .
81 /0 <i>id</i>	ADD <i>r/m32, imm32</i>	MI	Valid	Valid	Add <i>imm32</i> to <i>r/m32</i> .
REX.W + 81 /0 <i>id</i>	ADD <i>r/m64, imm32</i>	MI	Valid	N.E.	Add <i>imm32 sign-extended to 64-bits</i> to <i>r/m64</i> .
83 /0 <i>ib</i>	ADD <i>r/m16, imm8</i>	MI	Valid	Valid	Add <i>sign-extended imm8</i> to <i>r/m16</i> .
83 /0 <i>ib</i>	ADD <i>r/m32, imm8</i>	MI	Valid	Valid	Add <i>sign-extended imm8</i> to <i>r/m32</i> .
REX.W + 83 /0 <i>ib</i>	ADD <i>r/m64, imm8</i>	MI	Valid	N.E.	Add <i>sign-extended imm8</i> to <i>r/m64</i> .
00 / <i>r</i>	ADD <i>r/m8, r8</i>	MR	Valid	Valid	Add <i>r8</i> to <i>r/m8</i> .
REX + 00 / <i>r</i>	ADD <i>r/m8</i> [*] , <i>r8</i> [*]	MR	Valid	N.E.	Add <i>r8</i> to <i>r/m8</i> .
01 / <i>r</i>	ADD <i>r/m16, r16</i>	MR	Valid	Valid	Add <i>r16</i> to <i>r/m16</i> .
01 / <i>r</i>	ADD <i>r/m32, r32</i>	MR	Valid	Valid	Add <i>r32</i> to <i>r/m32</i> .
REX.W + 01 / <i>r</i>	ADD <i>r/m64, r64</i>	MR	Valid	N.E.	Add <i>r64</i> to <i>r/m64</i> .
02 / <i>r</i>	ADD <i>r8, r/m8</i>	RM	Valid	Valid	Add <i>r/m8</i> to <i>r8</i> .
REX + 02 / <i>r</i>	ADD <i>r8</i> [*] , <i>r/m8</i> [*]	RM	Valid	N.E.	Add <i>r/m8</i> to <i>r8</i> .
03 / <i>r</i>	ADD <i>r16, r/m16</i>	RM	Valid	Valid	Add <i>r/m16</i> to <i>r16</i> .
03 / <i>r</i>	ADD <i>r32, r/m32</i>	RM	Valid	Valid	Add <i>r/m32</i> to <i>r32</i> .
REX.W + 03 / <i>r</i>	ADD <i>r64, r/m64</i>	RM	Valid	N.E.	Add <i>r/m64</i> to <i>r64</i> .

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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AH/EAX/RAX	imm8	NA	NA

Description

Adds the destination operand (first operand) and the source operand (second operand) and then 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, a register, or a 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 ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

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 or 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

$\text{DEST} \leftarrow \text{DEST} + \text{SRC};$

Flags Affected

The OF, SF, ZF, AF, CF, 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 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ADDPD—Add Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 58 /r ADDPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Add packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 58 /r VADDPD <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Add packed double-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG 58 /r VADDPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Add packed double-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed double-precision floating-point results 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).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of SIMD double-precision floating-point operation.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

ADDPD (128-bit Legacy SSE version)

```
DEST[63:0] ← DEST[63:0] + SRC[63:0];
DEST[127:64] ← DEST[127:64] + SRC[127:64];
DEST[VLMAX-1:128] (Unmodified)
```

VADDPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[127:64] ← SRC1[127:64] + SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```

VADDPD (VEX.256 encoded version)

DEST[63:0] \leftarrow SRC1[63:0] + SRC2[63:0]
DEST[127:64] \leftarrow SRC1[127:64] + SRC2[127:64]
DEST[191:128] \leftarrow SRC1[191:128] + SRC2[191:128]
DEST[255:192] \leftarrow SRC1[255:192] + SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

ADDPD: `__m128d _mm_add_pd (__m128d a, __m128d b)`
VADDPD: `__m256d _mm256_add_pd (__m256d a, __m256d b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

ADDPS—Add Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 58 /r ADDPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Add packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> and stores result in <i>xmm1</i> .
VEX.NDS.128.0F.WIG 58 /r VADDPS <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Add packed single-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.0F.WIG 58 /r VADDPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Add packed single-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r))	NA

Description

Performs a SIMD add of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results 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).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of SIMD single-precision floating-point operation.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

ADDPS (128-bit Legacy SSE version)

```
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];
DEST[VLMAX-1:128] (Unmodified)
```

VADDPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] + SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

VADDPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] + \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] + \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] + \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] + \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] + \text{SRC2}[255:224]$

Intel C/C++ Compiler Intrinsic Equivalent

ADDPS: `_m128 _mm_add_ps(_m128 a, _m128 b)`
 VADDPS: `_m256 _mm256_add_ps (_m256 a, _m256 b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

ADDSD—Add Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 58 /r ADDSD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Add the low double-precision floating-point value from <i>xmm2/m64</i> to <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 58 /r VADDSD <i>xmm1, xmm2, xmm3/m64</i>		RVM V/V	AVX	Add the low double-precision floating-point value from <i>xmm3/mem</i> to <i>xmm2</i> and store the result in <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r))	NA

Description

Adds the low double-precision floating-point values from the source operand (second operand) and 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. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview 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).

128-bit Legacy SSE version: Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

ADDSD (128-bit Legacy SSE version)

DEST[63:0] \leftarrow DEST[63:0] + SRC[63:0]
DEST[VLMAX-1:64] (Unmodified)

VADDSD (VEX.128 encoded version)

DEST[63:0] \leftarrow SRC1[63:0] + SRC2[63:0]
DEST[127:64] \leftarrow SRC1[127:64]
DEST[VLMAX-1:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

ADDSD: __m128d _mm_add_sd (m128d a, m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

ADDSS—Add Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 58 /r ADDSS <i>xmm1, xmm2/m32</i>		RM V/V	SSE	Add the low single-precision floating-point value from <i>xmm2/m32</i> to <i>xmm1</i> .
VEX.NDS.LIG.F3.OF.WIG 58 /r VADDSS <i>xmm1,xmm2,xmm3/m32</i>	RVM	V/V	AVX	Add the low single-precision floating-point value from <i>xmm3/mem</i> to <i>xmm2</i> and store the result in <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds the low single-precision floating-point values from the source operand (second operand) and 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. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview 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).

128-bit Legacy SSE version: Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation**ADDSS DEST, SRC (128-bit Legacy SSE version)**

```
DEST[31:0] ← DEST[31:0] + SRC[31:0];
DEST[VLMAX-1:32] (Unmodified)
```

VADDSS DEST, SRC1, SRC2 (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

ADDSS: `__m128_mm_add_ss(__m128 a, __m128 b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

ADDSUBPD—Packed Double-FP Add/Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F D0 /r ADDSUBPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Add/subtract double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG D0 /r VADDSDUPD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Add/subtract packed double-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG D0 /r VADDSDUPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Add / subtract packed double-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds odd-numbered double-precision floating-point values of the first source operand (second operand) with the corresponding double-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered double-precision floating-point values from the second source operand from the corresponding double-precision floating values in the first source operand; stores the result into the even-numbered values 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).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-3.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

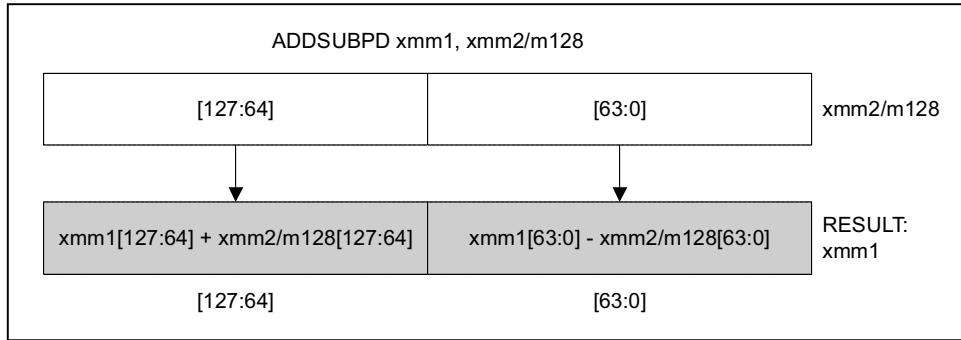


Figure 3-3. ADDSUBPD—Packed Double-FP Add/Subtract

Operation

ADDSUBPD (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] - \text{SRC}[63:0]$
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64] + \text{SRC}[127:64]$
 $\text{DEST}[VLMAX-1:128] \text{ (Unmodified)}$

VADDSUBPD (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] - \text{SRC2}[63:0]$
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] + \text{SRC2}[127:64]$
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

VADDSUBPD (VEX.256 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] - \text{SRC2}[63:0]$
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] + \text{SRC2}[127:64]$
 $\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128] - \text{SRC2}[191:128]$
 $\text{DEST}[255:192] \leftarrow \text{SRC1}[255:192] + \text{SRC2}[255:192]$

Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPD: `__m128d _mm_addsub_pd(__m128d a, __m128d b)`
 VADDSUBPD: `__m256d _mm256_addsub_pd (__m256d a, __m256d b)`

Exceptions

When the source operand is a memory operand, it must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

ADDSUBPS—Packed Single-FP Add/Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF D0 /r ADDSUBPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE3	Add/subtract single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.F2.0F.WIG D0 /r VADDSSUBPS <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Add/subtract single-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.F2.0F.WIG D0 /r VADDSSUBPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Add / subtract single-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

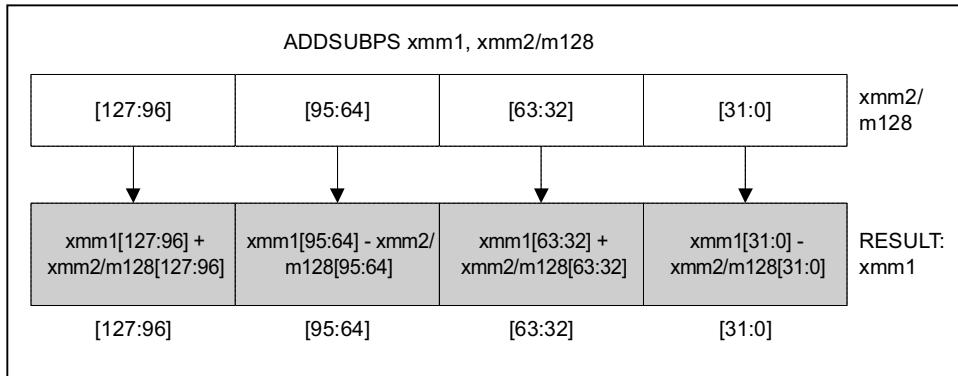
Adds odd-numbered single-precision floating-point values of the first source operand (second operand) with the corresponding single-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered single-precision floating-point values from the second source operand from the corresponding single-precision floating values in the first source operand; stores the result into the even-numbered values 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).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-4.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.



OM15992

Figure 3-4. ADDSUBPS—Packed Single-FP Add/Subtract

Operation

ADDSUBPS (128-bit Legacy SSE version)

```

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]
DEST[VLMAX-1:128] (Unmodified)

```

VADDSUBPS (VEX.128 encoded version)

```

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[VLMAX-1:128] ← 0

```

VADDSUBPS (VEX.256 encoded version)

```

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[159:128] ← SRC1[159:128] - SRC2[159:128]
DEST[191:160] ← SRC1[191:160] + SRC2[191:160]
DEST[223:192] ← SRC1[223:192] - SRC2[223:192]
DEST[255:224] ← SRC1[255:224] + SRC2[255:224].

```

Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPS: `__m128_mm_addsub_ps(__m128 a, __m128 b)`

VADDSUBPS: `__m256_mm256_addsub_ps (__m256 a, __m256 b)`

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

AESDEC—Perform One Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DE /r AESDEC xmm1, xmm2/m128		RM V/V	AES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs a single round of the AES decryption flow using the Equivalent Inverse Cipher, with the round key from the second source operand, operating on a 128-bit data (state) from the first source operand, and store the result in the destination operand.

Use the AESDEC instruction for all but the last decryption round. For the last decryption round, use the AESDEC-CLAST instruction.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

```

AESDEC
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
STATE ← InvMixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] (Unmodified)

```

VAESDEC

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
STATE ← InvMixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDEC: __m128i _mm_aesdec (__m128i, __m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

AESDECLAST—Perform Last Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DF /r AESDECLAST xmm1, xmm2/m128	RM	V/V	AES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs the last round of the AES decryption flow using the Equivalent Inverse Cipher, with the round key from the second source operand, operating on a 128-bit data (state) from the first source operand, and store the result in the destination operand.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation**AESDECLAST**

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] (Unmodified)
```

VAESDECLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDECLAST: __m128i _mm_aesdeclast (__m128i, __m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

AESENC—Perform One Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DC /r AESENC xmm1, xmm2/m128	RM	V/V	AES	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from the xmm3/m128; store the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs a single round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

Use the AESENC instruction for all but the last encryption rounds. For the last encryption round, use the AESENC-CLAST instruction.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation**AESENC**

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
STATE ← MixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] (Unmodified)
```

VAESENC

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
STATE ← MixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENCL __m128i _mm_aesenc (__m128i, __m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

AESENCLAST—Perform Last Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DD /r AESENCLAST xmm1, xmm2/m128	RM	V/V	AES	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128 bit round key from xmm3/m128; store the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs the last round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation**AESENCLAST**

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] (Unmodified)
```

VAESENCLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENCLAST: __m128i _mm_aesenclast (__m128i, __m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

AESIMC—Perform the AES InvMixColumn Transformation

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DB /r AESIMC xmm1, xmm2/m128	RM	V/V	AES	Perform the InvMixColumn transformation on a 128-bit round key from xmm2/m128 and store the result in xmm1.
VEX.128.66.0F38.WIG DB /r VAESIMC xmm1, xmm2/m128	RM	V/V	Both AES and AVX flags	Perform the InvMixColumn transformation on a 128-bit round key from xmm2/m128 and store the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Perform the InvMixColumns transformation on the source operand and store the result in the destination operand. The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location.

Note: the AESIMC instruction should be applied to the expanded AES round keys (except for the first and last round key) in order to prepare them for decryption using the "Equivalent Inverse Cipher" (defined in FIPS 197).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation**AESIMC**

DEST[127:0] \leftarrow InvMixColumns(SRC);
DEST[VLMAX-1:128] (Unmodified)

VAESIMC

DEST[127:0] \leftarrow InvMixColumns(SRC);
DEST[VLMAX-1:128] \leftarrow 0;

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESIMC: _m128i _mm_aesimc (_m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv != 1111B.

AESKEYGENASSIST—AES Round Key Generation Assist

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A DF /r ib AESKEYGENASSIST xmm1, xmm2/m128, imm8	RMI	V/V	AES	Assist in AES round key generation using an 8 bits Round Constant (RCON) specified in the immediate byte, operating on 128 bits of data specified in xmm2/m128 and stores the result in xmm1.
VEX.128.66.0F3A.WIG DF /r ib VAESKEYGENASSIST xmm1, xmm2/m128, imm8	RMI	V/V	Both AES and AVX flags	Assist in AES round key generation using 8 bits Round Constant (RCON) specified in the immediate byte, operating on 128 bits of data specified in xmm2/m128 and stores the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Assist in expanding the AES cipher key, by computing steps towards generating a round key for encryption, using 128-bit data specified in the source operand and an 8-bit round constant specified as an immediate, store the result in the destination operand.

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

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

AESKEYGENASSIST

```
X3[31:0] ← SRC [127: 96];
X2[31:0] ← SRC [95: 64];
X1[31:0] ← SRC [63: 32];
X0[31:0] ← SRC [31: 0];
RCON[31:0] ← ZeroExtend(Imm8[7:0]);
DEST[31:0] ← SubWord(X1);
DEST[63:32 ] ← RotWord( SubWord(X1) ) XOR RCON;
DEST[95:64] ← SubWord(X3);
DEST[127:96] ← RotWord( SubWord(X3) ) XOR RCON;
DEST[VLMAX-1:128] (Unmodified)
```

VAESKEYGENASSIST

```

X3[31:0] ← SRC [127: 96];
X2[31:0] ← SRC [95: 64];
X1[31:0] ← SRC [63: 32];
X0[31:0] ← SRC [31: 0];
RCON[31:0] ← ZeroExtend(Imm8[7:0]);
DEST[31:0] ← SubWord(X1);
DEST[63:32 ] ← RotWord( SubWord(X1) ) XOR RCON;
DEST[95:64] ← SubWord(X3);
DEST[127:96] ← RotWord( SubWord(X3) ) XOR RCON;
DEST[VLMAX-1:128] ← 0;

```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESKEYGENASSIST: `__m128i _mm_aesimc (__m128i, const int)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv != 1111B.

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 <i>ib</i>	AND AL, <i>imm8</i>	I	Valid	Valid	AL AND <i>imm8</i> .
25 <i>iw</i>	AND AX, <i>imm16</i>	I	Valid	Valid	AX AND <i>imm16</i> .
25 <i>id</i>	AND EAX, <i>imm32</i>	I	Valid	Valid	EAX AND <i>imm32</i> .
REX.W + 25 <i>id</i>	AND RAX, <i>imm32</i>	I	Valid	N.E.	RAX AND <i>imm32</i> sign-extended to 64-bits.
80 /4 <i>ib</i>	AND r/m8, <i>imm8</i>	MI	Valid	Valid	r/m8 AND <i>imm8</i> .
REX + 80 /4 <i>ib</i>	AND r/m8*, <i>imm8</i>	MI	Valid	N.E.	r/m8 AND <i>imm8</i> .
81 /4 <i>iw</i>	AND r/m16, <i>imm16</i>	MI	Valid	Valid	r/m16 AND <i>imm16</i> .
81 /4 <i>id</i>	AND r/m32, <i>imm32</i>	MI	Valid	Valid	r/m32 AND <i>imm32</i> .
REX.W + 81 /4 <i>id</i>	AND r/m64, <i>imm32</i>	MI	Valid	N.E.	r/m64 AND <i>imm32</i> sign extended to 64-bits.
83 /4 <i>ib</i>	AND r/m16, <i>imm8</i>	MI	Valid	Valid	r/m16 AND <i>imm8</i> (sign-extended).
83 /4 <i>ib</i>	AND r/m32, <i>imm8</i>	MI	Valid	Valid	r/m32 AND <i>imm8</i> (sign-extended).
REX.W + 83 /4 <i>ib</i>	AND r/m64, <i>imm8</i>	MI	Valid	N.E.	r/m64 AND <i>imm8</i> (sign-extended).
20 /r	AND r/m8, r8	MR	Valid	Valid	r/m8 AND r8.
REX + 20 /r	AND r/m8*, r8*	MR	Valid	N.E.	r/m64 AND r8 (sign-extended).
21 /r	AND r/m16, r16	MR	Valid	Valid	r/m16 AND r16.
21 /r	AND r/m32, r32	MR	Valid	Valid	r/m32 AND r32.
REX.W + 21 /r	AND r/m64, r64	MR	Valid	N.E.	r/m64 AND r32.
22 /r	AND r8, r/m8	RM	Valid	Valid	r8 AND r/m8.
REX + 22 /r	AND r8*, r/m8*	RM	Valid	N.E.	r/m64 AND r8 (sign-extended).
23 /r	AND r16, r/m16	RM	Valid	Valid	r16 AND r/m16.
23 /r	AND r32, r/m32	RM	Valid	Valid	r32 AND r/m32.
REX.W + 23 /r	AND r64, r/m64	RM	Valid	N.E.	r64 AND 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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AH/EAX/RAX	imm8	NA	NA

Description

Performs a bitwise AND 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 set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

This instruction can be used with a LOCK prefix to allow the it 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

$\text{DEST} \leftarrow \text{DEST AND SRC};$

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ANDN – Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.0F38.W0 F2 /r ANDN r32a, r32b, r/m32	RVM	V/V	BMI1	Bitwise AND of inverted r32b with r/m32, store result in r32a.
VEX.NDS.LZ. OF38.W1 F2 /r ANDN r64a, r64b, r/m64	RVM	V/NE	BMI1	Bitwise AND of inverted r64b with r/m64, store result in r64a.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND of inverted second operand (the first source operand) with the third operand (the second source operand). The result is stored in the first operand (destination operand).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
DEST ← (NOT SRC1) bitwiseAND SRC2;
SF ← DEST[OperandSize -1];
ZF ← (DEST = 0);
```

Flags Affected

SF and ZF are updated based on result. OF and CF flags are cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.

ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 54 /r ANDPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Return the bitwise logical AND of packed double-precision floating-point values in <i>xmm1</i> and <i>xmm2/m128</i> .
VEX.NDS.128.66.0F.WIG 54 /r VANDPD <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 54 /r VANDPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND 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.

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation**ANDPD (128-bit Legacy SSE version)**

```
DEST[63:0] ← DEST[63:0] BITWISE AND SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE AND SRC[127:64]
DEST[VLMAX-1:128] (Unmodified)
```

VANDPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```

VANDPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE AND SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE AND SRC2[255:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

ANDPD: `__m128d _mm_and_pd(__m128d a, __m128d b)`

VANDPD: `__m256d _mm256_and_pd (__m256d a, __m256d b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 54 /r ANDPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Bitwise logical AND of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.0F.WIG 54 /r VANDPS <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG 54 /r VANDPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND of the four or eight packed single-precision floating-point values from the first source operand and the second source operand, and stores the result 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).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

ANDPS (128-bit Legacy SSE version)

```
DEST[31:0] ← DEST[31:0] BITWISE AND SRC[31:0]
DEST[63:32] ← DEST[63:32] BITWISE AND SRC[63:32]
DEST[95:64] ← DEST[95:64] BITWISE AND SRC[95:64]
DEST[127:96] ← DEST[127:96] BITWISE AND SRC[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

VANDPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

VANDPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] \text{ BITWISE AND SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] \text{ BITWISE AND SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] \text{ BITWISE AND SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] \text{ BITWISE AND SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] \text{ BITWISE AND SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] \text{ BITWISE AND SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] \text{ BITWISE AND SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] \text{ BITWISE AND SRC2}[255:224].$

Intel C/C++ Compiler Intrinsic Equivalent

ANDPS: `__m128 _mm_and_ps(__m128 a, __m128 b)`
 VANDPS: `__m256 _mm256_and_ps (__m256 a, __m256 b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 55 /r ANDNPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Bitwise logical AND NOT of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 55 /r VANDNPD <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 55/r VANDNPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND NOT of the two or four packed double-precision floating-point values from the first source operand and the second source operand, and stores the result 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).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation**ANDNPD (128-bit Legacy SSE version)**

```
DEST[63:0] ← (NOT(DEST[63:0])) BITWISE AND SRC[63:0]
DEST[127:64] ← (NOT(DEST[127:64])) BITWISE AND SRC[127:64]
DEST[VLMAX-1:128] (Unmodified)
```

VANDNPD (VEX.128 encoded version)

```
DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```

VANDNPD (VEX.256 encoded version)

```
DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[191:128] ← (NOT(SRC1[191:128])) BITWISE AND SRC2[191:128]
DEST[255:192] ← (NOT(SRC1[255:192])) BITWISE AND SRC2[255:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

ANDNPD: `__m128d _mm_andnot_pd(__m128d a, __m128d b)`

VANDNPD: `__m256d _mm256_andnot_pd (__m256d a, __m256d b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 55 /r ANDNPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Bitwise logical AND NOT of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.0F.WIG 55 /r VANDNPS <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG 55 /r VANDNPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Inverts the bits of the four packed single-precision floating-point values in the destination operand (first operand), performs a bitwise logical AND of the four packed single-precision floating-point values in the source operand (second operand) and the temporary inverted result, and stores the result 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).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation**ANDNPS (128-bit Legacy SSE version)**

```
DEST[31:0] ← (NOT(DEST[31:0])) BITWISE AND SRC[31:0]
DEST[63:32] ← (NOT(DEST[63:32])) BITWISE AND SRC[63:32]
DEST[95:64] ← (NOT(DEST[95:64])) BITWISE AND SRC[95:64]
DEST[127:96] ← (NOT(DEST[127:96])) BITWISE AND SRC[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

VANDNPS (VEX.128 encoded version)

```
DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

VANDNPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow (\text{NOT}(\text{SRC1}[31:0])) \text{ BITWISE AND SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow (\text{NOT}(\text{SRC1}[63:32])) \text{ BITWISE AND SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow (\text{NOT}(\text{SRC1}[95:64])) \text{ BITWISE AND SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow (\text{NOT}(\text{SRC1}[127:96])) \text{ BITWISE AND SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow (\text{NOT}(\text{SRC1}[159:128])) \text{ BITWISE AND SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow (\text{NOT}(\text{SRC1}[191:160])) \text{ BITWISE AND SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow (\text{NOT}(\text{SRC1}[223:192])) \text{ BITWISE AND SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow (\text{NOT}(\text{SRC1}[255:224])) \text{ BITWISE AND SRC2}[255:224]$.

Intel C/C++ Compiler Intrinsic Equivalent

ANDNPS: $\text{__m128 _mm_andnot_ps}(\text{__m128 a, __m128 b})$
 VANDNPS: $\text{__m256 _mm256_andnot_ps}(\text{__m256 a, __m256 b})$

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

ARPL—Adjust RPL Field of Segment Selector

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
63 /r	ARPL r/m16, r16	NP	N. E.	Valid	Adjust RPL of r/m16 to not less than RPL of r16.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Compares the RPL fields of two segment selectors. The first operand (the destination operand) contains one segment selector and the second operand (source operand) contains the other. (The RPL field is located in bits 0 and 1 of each operand.) If the RPL field of the destination operand is less than the RPL field of the source operand, the ZF flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the ZF flag is cleared and no change is made to the destination operand. (The destination operand can be a word register or a memory location; the source operand must be a word register.)

The ARPL instruction is provided for use by operating-system procedures (however, it can also be used by applications). It is generally used to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. Here the segment selector passed to the operating system is placed in the destination operand and segment selector for the application program's code segment is placed in the source operand. (The RPL field in the source operand represents the privilege level of the application program.) Execution of the ARPL instruction then ensures that the RPL of the segment selector received by the operating system is no lower (does not have a higher privilege) than the privilege level of the application program (the segment selector for the application program's code segment can be read from the stack following a procedure call).

This instruction executes as described in compatibility mode and legacy mode. It is not encodable in 64-bit mode.

See "Checking Caller Access Privileges" in Chapter 3, "Protected-Mode Memory Management," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information about the use of this instruction.

Operation

```

IF 64-BIT MODE
  THEN
    See MOVSXD;
  ELSE
    IF DEST[RPL] < SRC[RPL]
      THEN
        ZF ← 1;
        DEST[RPL] ← SRC[RPL];
      ELSE
        ZF ← 0;
    Fl;
Fl;

```

Flags Affected

The ZF flag is set to 1 if the RPL field of the destination operand is less than that of the source operand; otherwise, it is set to 0.

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.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #UD The ARPL instruction is not recognized in real-address mode.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #UD The ARPL instruction is not recognized in virtual-8086 mode.
If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Not applicable.

BLENDPD — Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 0D /r ib BLENDPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select packed DP-FP values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 0D /r ib VBLENDPD <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Select packed double-precision floating-point Values from <i>xmm2</i> and <i>xmm3/m128</i> from mask in <i>imm8</i> and store the values in <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG 0D /r ib VBLENDPD <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Select packed double-precision floating-point Values from <i>ymm2</i> and <i>ymm3/m256</i> from mask in <i>imm8</i> and store the values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[3:0]

Description

Double-precision floating-point values from the second source operand (third operand) are conditionally merged with values from the first source operand (second operand) and written to the destination operand (first operand). The immediate bits [3:0] determine whether the corresponding double-precision floating-point value in the destination is copied from the second source or first source. If a bit in the mask, corresponding to a word, is "1", then the double-precision floating-point value in the second source operand is copied, else the value in the first source operand is copied.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation**BLENDPD (128-bit Legacy SSE version)**

```
IF (IMM8[0] = 0) THEN DEST[63:0] ← DEST[63:0]
    ELSE DEST [63:0] ← SRC[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← DEST[127:64]
    ELSE DEST [127:64] ← SRC[127:64] FI
DEST[VLMAX-1:128] (Unmodified)
```

VBLENDPD (VEX.128 encoded version)

```
IF (IMM8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST [63:0] ← SRC2[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← SRC1[127:64]
    ELSE DEST [127:64] ← SRC2[127:64] FI
DEST[VLMAX-1:128] ← 0
```

VBLENDPD (VEX.256 encoded version)

```

IF (IMM8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
ELSE DEST [63:0] ← SRC2[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← SRC1[127:64]
ELSE DEST [127:64] ← SRC2[127:64] FI
IF (IMM8[2] = 0) THEN DEST[191:128] ← SRC1[191:128]
ELSE DEST [191:128] ← SRC2[191:128] FI
IF (IMM8[3] = 0) THEN DEST[255:192] ← SRC1[255:192]
ELSE DEST [255:192] ← SRC2[255:192] FI

```

Intel C/C++ Compiler Intrinsic Equivalent

BLENDPD: __m128d _mm_blend_pd (__m128d v1, __m128d v2, const int mask);
 VBLENDPD: __m256d _mm256_blend_pd (__m256d a, __m256d b, const int mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

BEXTR – Bit Field Extract

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS ¹ .LZ.0F38.W0 F7 /r BEXTR r32a, r/m32, r32b	RMV	V/V	BMI1	Contiguous bitwise extract from r/m32 using r32b as control; store result in r32a.
VEX.NDS ¹ .LZ.0F38.W1 F7 /r BEXTR r64a, r/m64, r64b	RMV	V/N.E.	BMI1	Contiguous bitwise extract from r/m64 using r64b as control; store result in r64a

NOTES:

1. ModRM:r/m is used to encode the first source operand (second operand) and VEX.vvvv encodes the second source operand (third operand).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

Description

Extracts contiguous bits from the first source operand (the second operand) using an index value and length value specified in the second source operand (the third operand). Bit 7:0 of the second source operand specifies the starting bit position of bit extraction. A START value exceeding the operand size will not extract any bits from the second source operand. Bit 15:8 of the second source operand specifies the maximum number of bits (LENGTH) beginning at the START position to extract. Only bit positions up to (OperandSize -1) of the first source operand are extracted. The extracted bits are written to the destination register, starting from the least significant bit. All higher order bits in the destination operand (starting at bit position LENGTH) are zeroed. The destination register is cleared if no bits are extracted.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```

START ← SRC2[7:0];
LEN ← SRC2[15:8];
TEMP ← ZERO_EXTEND_TO_512(SRC1);
DEST ← ZERO_EXTEND(TEMP[START+LEN -1: START]);
ZF ← (DEST = 0);

```

Flags Affected

ZF is updated based on the result. AF, SF, and PF are undefined. All other flags are cleared.

Intel C/C++ Compiler Intrinsic Equivalent

```

BEXTR:     unsigned __int32 _bextr_u32(unsigned __int32 src, unsigned __int32 start, unsigned __int32 len);
BEXTR:     unsigned __int64 _bextr_u64(unsigned __int64 src, unsigned __int32 start, unsigned __int32 len);

```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally
#UD If VEX.W = 1.

BLENDPS — Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A OC /r ib BLENDPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select packed single precision floating-point values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG OC /r ib VBLENDPS <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Select packed single-precision floating-point values from <i>xmm2</i> and <i>xmm3/m128</i> from mask in <i>imm8</i> and store the values in <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG OC /r ib VBLENDPS <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Select packed single-precision floating-point values from <i>ymm2</i> and <i>ymm3/m256</i> from mask in <i>imm8</i> and store the values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Packed single-precision floating-point values from the second source operand (third operand) are conditionally merged with values from the first source operand (second operand) and written to the destination operand (first operand). The immediate bits [7:0] determine whether the corresponding single precision floating-point value in the destination is copied from the second source or first source. If a bit in the mask, corresponding to a word, is "1", then the single-precision floating-point value in the second source operand is copied, else the value in the first source operand is copied.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation**BLENDPS (128-bit Legacy SSE version)**

```

IF (IMM8[0] = 0) THEN DEST[31:0] ← DEST[31:0]
    ELSE DEST [31:0] ← SRC[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← DEST[63:32]
    ELSE DEST [63:32] ← SRC[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← DEST[95:64]
    ELSE DEST [95:64] ← SRC[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← DEST[127:96]
    ELSE DEST [127:96] ← SRC[127:96] FI
DEST[VLMAX-1:128] (Unmodified)

```

VBLENDPS (VEX.128 encoded version)

```

IF (IMM8[0] = 0) THEN DEST[31:0] ← SRC1[31:0]
    ELSE DEST [31:0] ← SRC2[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← SRC1[63:32]
    ELSE DEST [63:32] ← SRC2[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← SRC1[95:64]
    ELSE DEST [95:64] ← SRC2[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← SRC1[127:96]
    ELSE DEST [127:96] ← SRC2[127:96] FI
DEST[VLMAX-1:128] ← 0

```

VBLENDPS (VEX.256 encoded version)

```

IF (IMM8[0] = 0) THEN DEST[31:0] ← SRC1[31:0]
    ELSE DEST [31:0] ← SRC2[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← SRC1[63:32]
    ELSE DEST [63:32] ← SRC2[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← SRC1[95:64]
    ELSE DEST [95:64] ← SRC2[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← SRC1[127:96]
    ELSE DEST [127:96] ← SRC2[127:96] FI
IF (IMM8[4] = 0) THEN DEST[159:128] ← SRC1[159:128]
    ELSE DEST [159:128] ← SRC2[159:128] FI
IF (IMM8[5] = 0) THEN DEST[191:160] ← SRC1[191:160]
    ELSE DEST [191:160] ← SRC2[191:160] FI
IF (IMM8[6] = 0) THEN DEST[223:192] ← SRC1[223:192]
    ELSE DEST [223:192] ← SRC2[223:192] FI
IF (IMM8[7] = 0) THEN DEST[255:224] ← SRC1[255:224]
    ELSE DEST [255:224] ← SRC2[255:224] FI.

```

Intel C/C++ Compiler Intrinsic Equivalent

BLENDPS: __m128 _mm_blend_ps (__m128 v1, __m128 v2, const int mask);
 VBLENDPS: __m256 _mm256_blend_ps (__m256 a, __m256 b, const int mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

BLENDVPD – Variable Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 15 /r BLENDVPD <i>xmm1, xmm2/m128, <XMM0></i>	RM0	V/V	SSE4_1	Select packed DP FP values from <i>xmm1</i> and <i>xmm2</i> from mask specified in <i>XMM0</i> and store the values in <i>xmm1</i> .
VEX.NDS.128.66.0F3A.W0 4B /r /is4 VBLENDVPD <i>xmm1, xmm2, xmm3/m128, xmm4</i>	RVMR	V/V	AVX	Conditionally copy double-precision floating-point values from <i>xmm2</i> or <i>xmm3/m128</i> to <i>xmm1</i> , based on mask bits in the mask operand, <i>xmm4</i> .
VEX.NDS.256.66.0F3A.W0 4B /r /is4 VBLENDVPD <i>ymm1, ymm2, ymm3/m256, ymm4</i>	RVMR	V/V	AVX	Conditionally copy double-precision floating-point values from <i>ymm2</i> or <i>ymm3/m256</i> to <i>ymm1</i> , based on mask bits in the mask operand, <i>ymm4</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM0	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	NA
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

Conditionally copy each quadword data element of double-precision floating-point value from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each quadword element of the mask register.

Each quadword element of the destination operand is copied from:

- the corresponding quadword element in the second source operand, If a mask bit is "1"; or
- the corresponding quadword element in the first source operand, If a mask bit is "0"

The register assignment of the implicit mask operand for BLENDVPD is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPD with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (VLMAX-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and destination operand are YMM registers. The second source operand can be a YMM register or a 256-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. VEX.W must be 0, otherwise, the instruction will #UD.

VBLENDVPD permits the mask to be any XMM or YMM register. In contrast, BLENDVPD treats XMM0 implicitly as the mask and do not support non-destructive destination operation.

Operation

BLENDVPD (128-bit Legacy SSE version)

```

MASK ← XMM0
IF (MASK[63] = 0) THEN DEST[63:0] ← DEST[63:0]
    ELSE DEST [63:0] ← SRC[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← DEST[127:64]
    ELSE DEST [127:64] ← SRC[127:64] FI
DEST[VLMAX-1:128] (Unmodified)

```

VBLENDVPD (VEX.128 encoded version)

```

MASK ← SRC3
IF (MASK[63] = 0) THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST [63:0] ← SRC2[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← SRC1[127:64]
    ELSE DEST [127:64] ← SRC2[127:64] FI
DEST[VLMAX-1:128] ← 0

```

VBLENDVPD (VEX.256 encoded version)

```

MASK ← SRC3
IF (MASK[63] = 0) THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST [63:0] ← SRC2[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← SRC1[127:64]
    ELSE DEST [127:64] ← SRC2[127:64] FI
IF (MASK[191] = 0) THEN DEST[191:128] ← SRC1[191:128]
    ELSE DEST [191:128] ← SRC2[191:128] FI
IF (MASK[255] = 0) THEN DEST[255:192] ← SRC1[255:192]
    ELSE DEST [255:192] ← SRC2[255:192] FI

```

Intel C/C++ Compiler Intrinsic Equivalent

```

BLENDVPD:   __m128d _mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3);
VBLENDVPD:  __m128 _mm_blendv_pd (__m128d a, __m128d b, __m128d mask);
VBLENDVPD:  __m256 _mm256_blendv_pd (__m256d a, __m256d b, __m256d mask);

```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally
#UD If VEX.W = 1.

BLENDVPS – Variable Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 14 /r BLENDVPS <i>xmm1</i> , <i>xmm2/m128</i> , < <i>XMM0</i> >	RMO	V/V	SSE4_1	Select packed single precision floating-point values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>XMM0</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.W0 4A /r /is4 VBLENDVPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>xmm4</i>	VMR	V/V	AVX	Conditionally copy single-precision floating-point values from <i>xmm2</i> or <i>xmm3/m128</i> to <i>xmm1</i> , based on mask bits in the specified mask operand, <i>xmm4</i> .
VEX.NDS.256.66.0F3A.W0 4A /r /is4 VBLENDVPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>ymm4</i>	VMR	V/V	AVX	Conditionally copy single-precision floating-point values from <i>ymm2</i> or <i>ymm3/m256</i> to <i>ymm1</i> , based on mask bits in the specified mask register, <i>ymm4</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMO	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	NA
VMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

Conditionally copy each dword data element of single-precision floating-point value from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each dword element of the mask register.

Each quadword element of the destination operand is copied from:

- the corresponding dword element in the second source operand, If a mask bit is "1"; or
- the corresponding dword element in the first source operand, If a mask bit is "0"

The register assignment of the implicit mask operand for BLENDVPS is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPS with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (VLMAX-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and destination operand are YMM registers. The second source operand can be a YMM register or a 256-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. VEX.W must be 0, otherwise, the instruction will #UD.

VBLENDVPS permits the mask to be any XMM or YMM register. In contrast, BLENDVPS treats XMM0 implicitly as the mask and do not support non-destructive destination operation.

Operation

BLENDVPS (128-bit Legacy SSE version)

```

MASK ← XMM0
IF (MASK[31] = 0) THEN DEST[31:0] ← DEST[31:0]
    ELSE DEST [31:0] ← SRC[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← DEST[63:32]
    ELSE DEST [63:32] ← SRC[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← DEST[95:64]
    ELSE DEST [95:64] ← SRC[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← DEST[127:96]
    ELSE DEST [127:96] ← SRC[127:96] FI
DEST[VLMAX-1:128] (Unmodified)

```

VBLENDVPS (VEX.128 encoded version)

```

MASK ← SRC3
IF (MASK[31] = 0) THEN DEST[31:0] ← SRC1[31:0]
    ELSE DEST [31:0] ← SRC2[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← SRC1[63:32]
    ELSE DEST [63:32] ← SRC2[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← SRC1[95:64]
    ELSE DEST [95:64] ← SRC2[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← SRC1[127:96]
    ELSE DEST [127:96] ← SRC2[127:96] FI
DEST[VLMAX-1:128] ← 0

```

VBLENDVPS (VEX.256 encoded version)

```

MASK ← SRC3
IF (MASK[31] = 0) THEN DEST[31:0] ← SRC1[31:0]
    ELSE DEST [31:0] ← SRC2[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← SRC1[63:32]
    ELSE DEST [63:32] ← SRC2[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← SRC1[95:64]
    ELSE DEST [95:64] ← SRC2[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← SRC1[127:96]
    ELSE DEST [127:96] ← SRC2[127:96] FI
IF (MASK[159] = 0) THEN DEST[159:128] ← SRC1[159:128]
    ELSE DEST [159:128] ← SRC2[159:128] FI
IF (MASK[191] = 0) THEN DEST[191:160] ← SRC1[191:160]
    ELSE DEST [191:160] ← SRC2[191:160] FI
IF (MASK[223] = 0) THEN DEST[223:192] ← SRC1[223:192]
    ELSE DEST [223:192] ← SRC2[223:192] FI
IF (MASK[255] = 0) THEN DEST[255:224] ← SRC1[255:224]
    ELSE DEST [255:224] ← SRC2[255:224] FI

```

Intel C/C++ Compiler Intrinsic Equivalent

```

BLENDVPS:   __m128_mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3);
VBLENDVPS:  __m128_mm_blendv_ps (__m128 a, __m128 b, __m128 mask);
VBLENDVPS:  __m256_mm256_blendv_ps (__m256 a, __m256 b, __m256 mask);

```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.W = 1.

BLSI – Extract Lowest Set Isolated Bit

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.0F38.W0 F3 /3 BLSI r32, r/m32	VM	V/V	BMI1	Extract lowest set bit from r/m32 and set that bit in r32.
VEX.NDD.LZ.0F38.W1 F3 /3 BLSI r64, r/m64	VM	V/N.E.	BMI1	Extract lowest set bit from r/m64, and set that bit in r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
VM	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Extracts the lowest set bit from the source operand and set the corresponding bit in the destination register. All other bits in the destination operand are zeroed. If no bits are set in the source operand, BLSI sets all the bits in the destination to 0 and sets ZF and CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```

temp ← (-SRC) bitwiseAND (SRC);
SF ← temp[OperandSize - 1];
ZF ← (temp = 0);
IF SRC = 0
    CF ← 0;
ELSE
    CF ← 1;
FI
DEST ← temp;
```

Flags Affected

ZF and SF are updated based on the result. CF is set if the source is not zero. OF flags are cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

```

BLSI:     unsigned __int32 _blsi_u32(unsigned __int32 src);
BLSI:     unsigned __int64 _blsi_u64(unsigned __int64 src);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.

BLSMSK – Get Mask Up to Lowest Set Bit

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.0F38.W0 F3 /2 BLSMSK r32, r/m32	VM	V/V	BMI1	Set all lower bits in r32 to “1” starting from bit 0 to lowest set bit in r/m32.
VEX.NDD.LZ.0F38.W1 F3 /2 BLSMSK r64, r/m64	VM	V/N.E.	BMI1	Set all lower bits in r64 to “1” starting from bit 0 to lowest set bit in r/m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
VM	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Sets all the lower bits of the destination operand to “1” up to and including lowest set bit (=1) in the source operand. If source operand is zero, BLSMSK sets all bits of the destination operand to 1 and also sets CF to 1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
temp ← (SRC-1) XOR (SRC);
SF ← temp[OperandSize -1];
ZF ← 0;
IF SRC = 0
    CF ← 1;
ELSE
    CF ← 0;
FI
DEST ← temp;
```

Flags Affected

SF is updated based on the result. CF is set if the source is zero. ZF and OF flags are cleared. AF and PF flag are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

```
BLSMSK:     unsigned __int32 _blsmask_u32(unsigned __int32 src);
BLSMSK:     unsigned __int64 _blsmask_u64(unsigned __int64 src);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.

BLSR – Reset Lowest Set Bit

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.0F38.W0 F3 /1 BLSR r32, r/m32	VM	V/V	BMI1	Reset lowest set bit of r/m32, keep all other bits of r/m32 and write result to r32.
VEX.NDD.LZ.0F38.W1 F3 /1 BLSR r64, r/m64	VM	V/N.E.	BMI1	Reset lowest set bit of r/m64, keep all other bits of r/m64 and write result to r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
VM	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Copies all bits from the source operand to the destination operand and resets (=0) the bit position in the destination operand that corresponds to the lowest set bit of the source operand. If the source operand is zero BLSR sets CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
temp ← (SRC-1) bitwiseAND ( SRC );
SF ← temp[OperandSize - 1];
ZF ← (temp = 0);
IF SRC = 0
    CF ← 1;
ELSE
    CF ← 0;
FI
DEST ← temp;
```

Flags Affected

ZF and SF flags are updated based on the result. CF is set if the source is zero. OF flag is cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

```
BLSR:     unsigned __int32 _blsr_u32(unsigned __int32 src);
BLSR:     unsigned __int64 _blsr_u64(unsigned __int64 src);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally
#UD If VEX.W = 1.

BOUND—Check Array Index Against Bounds

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
62 /r	BOUND r16, m16&16	RM	Invalid	Valid	Check if <i>r16</i> (array index) is within bounds specified by <i>m16&16</i> .
62 /r	BOUND r32, m32&32	RM	Invalid	Valid	Check if <i>r32</i> (array index) is within bounds specified by <i>m16&16</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

BOUND determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). The array index is a signed integer located in a register. The bounds operand is a memory location that contains a pair of signed doubleword-integers (when the operand-size attribute is 32) or a pair of signed word-integers (when the operand-size attribute is 16). The first doubleword (or word) is the lower bound of the array and the second doubleword (or word) is the upper bound of the array. The array index must be greater than or equal to the lower bound and less than or equal to the upper bound plus the operand size in bytes. If the index is not within bounds, a BOUND range exceeded exception (#BR) is signaled. When this exception is generated, the saved return instruction pointer points to the BOUND instruction.

The bounds limit data structure (two words or doublewords containing the lower and upper limits of the array) is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array. Because the address of the array already will be present in a register, this practice avoids extra bus cycles to obtain the effective address of the array bounds.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```

IF 64bit Mode
  THEN
    #UD;
  ELSE
    IF (ArrayIndex < LowerBound OR ArrayIndex > UpperBound)
      (* Below lower bound or above upper bound *)
      THEN #BR; Fl;
    Fl;
  
```

Flags Affected

None.

Protected Mode Exceptions

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#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

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#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

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#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

#UD	If in 64-bit mode.
-----	--------------------

BSF—Bit Scan Forward

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF BC /r	BSF r16, r/m16	RM	Valid	Valid	Bit scan forward on r/m16.
OF BC /r	BSF r32, r/m32	RM	Valid	Valid	Bit scan forward on r/m32.
REX.W + OF BC /r	BSF r64, r/m64	RM	Valid	N.E.	Bit scan forward on r/m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content of the source operand is 0, the content of the destination operand is undefined.

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 SRC = 0
  THEN
    ZF ← 1;
    DEST is undefined;
  ELSE
    ZF ← 0;
    temp ← 0;
    WHILE Bit(SRC, temp) = 0
      DO
        temp ← temp + 1;
      OD;
    DEST ← temp;
  FI;
```

Flags Affected

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are 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.
#UD	If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

BSR—Bit Scan Reverse

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF BD /r	BSR r16, r/m16	RM	Valid	Valid	Bit scan reverse on r/m16.
OF BD /r	BSR r32, r/m32	RM	Valid	Valid	Bit scan reverse on r/m32.
REX.W + OF BD /r	BSR r64, r/m64	RM	Valid	N.E.	Bit scan reverse on r/m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content source operand is 0, the content of the destination operand is undefined.

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 SRC = 0
  THEN
    ZF ← 1;
    DEST is undefined;
  ELSE
    ZF ← 0;
    temp ← OperandSize - 1;
    WHILE Bit(SRC, temp) = 0
      DO
        temp ← temp - 1;
      OD;
    DEST ← temp;
  FI;
```

Flags Affected

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are 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.
#UD	If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

BSWAP—Byte Swap

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF C8+rd	BSWAP r32	O	Valid*	Valid	Reverses the byte order of a 32-bit register.
REX.W + OF C8+rd	BSWAP r64	O	Valid	N.E.	Reverses the byte order of a 64-bit register.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
0	opcode + rd (r, w)	NA	NA	NA

Description

Reverses the byte order of a 32-bit or 64-bit (destination) register. This instruction is provided for converting little-endian values to big-endian format and vice versa. To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

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.

IA-32 Architecture Legacy Compatibility

The BSWAP instruction is not supported on IA-32 processors earlier than the Intel486™ processor family. For compatibility with this instruction, software should include functionally equivalent code for execution on Intel processors earlier than the Intel486 processor family.

Operation

```

TEMP ← DEST
IF 64-bit mode AND OperandSize = 64
    THEN
        DEST[7:0] ← TEMP[63:56];
        DEST[15:8] ← TEMP[55:48];
        DEST[23:16] ← TEMP[47:40];
        DEST[31:24] ← TEMP[39:32];
        DEST[39:32] ← TEMP[31:24];
        DEST[47:40] ← TEMP[23:16];
        DEST[55:48] ← TEMP[15:8];
        DEST[63:56] ← TEMP[7:0];
    ELSE
        DEST[7:0] ← TEMP[31:24];
        DEST[15:8] ← TEMP[23:16];
        DEST[23:16] ← TEMP[15:8];
        DEST[31:24] ← TEMP[7:0];
    Fl;

```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

BT—Bit Test

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF A3 /r	BT r/m16, r16	MR	Valid	Valid	Store selected bit in CF flag.
OF A3 /r	BT r/m32, r32	MR	Valid	Valid	Store selected bit in CF flag.
REX.W + OF A3 /r	BT r/m64, r64	MR	Valid	N.E.	Store selected bit in CF flag.
OF BA /4 ib	BT r/m16, imm8	MI	Valid	Valid	Store selected bit in CF flag.
OF BA /4 ib	BT r/m32, imm8	MI	Valid	Valid	Store selected bit in CF flag.
REX.W + OF BA /4 ib	BT r/m64, imm8	MI	Valid	N.E.	Store selected bit in CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r)	imm8	NA	NA

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset (specified by the second operand) and stores the value of the bit in the CF flag. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode).
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-10.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 or 5 bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high order bits if they are not zero.

When accessing a bit in memory, the processor may access 4 bytes starting from the memory address for a 32-bit operand size, using by the following relationship:

$$\text{Effective Address} + (4 * (\text{BitOffset DIV } 32))$$

Or, it may access 2 bytes starting from the memory address for a 16-bit operand, using this relationship:

$$\text{Effective Address} + (2 * (\text{BitOffset DIV } 16))$$

It may do so even when only a single byte needs to be accessed to reach the given bit. When using this bit addressing mechanism, software should avoid referencing areas of memory close to address space holes. In particular, it should avoid references to memory-mapped I/O registers. Instead, software should use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

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 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

$CF \leftarrow \text{Bit}(\text{BitBase}, \text{BitOffset});$

Flags Affected

The CF flag contains the value of the selected bit. The ZF flag is unaffected. The OF, SF, AF, and PF flags are 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

BTC—Bit Test and Complement

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF BB /r	BTC <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag and complement.
OF BB /r	BTC <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + OF BB /r	BTC <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag and complement.
OF BA /7 <i>ib</i>	BTC <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and complement.
OF BA /7 <i>ib</i>	BTC <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + OF BA /7 <i>ib</i>	BTC <i>r/m64, imm8</i>	MI	Valid	N.E.	Store selected bit in CF flag and complement.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and complements the selected bit in the bit string. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-10.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" in this chapter for more information on this addressing mechanism.

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

```
CF ← Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) ← NOT Bit(BitBase, BitOffset);
```

Flags Affected

The CF flag contains the value of the selected bit before it is complemented. The ZF flag is unaffected. The OF, SF, AF, and PF flags are 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

BTR—Bit Test and Reset

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF B3 /r	BTR r/m16, r16	MR	Valid	Valid	Store selected bit in CF flag and clear.
OF B3 /r	BTR r/m32, r32	MR	Valid	Valid	Store selected bit in CF flag and clear.
REX.W + OF B3 /r	BTR r/m64, r64	MR	Valid	N.E.	Store selected bit in CF flag and clear.
OF BA /6 ib	BTR r/m16, imm8	MI	Valid	Valid	Store selected bit in CF flag and clear.
OF BA /6 ib	BTR r/m32, imm8	MI	Valid	Valid	Store selected bit in CF flag and clear.
REX.W + OF BA /6 ib	BTR r/m64, imm8	MI	Valid	N.E.	Store selected bit in CF flag and clear.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and clears the selected bit in the bit string to 0. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-10.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" in this chapter for more information on this addressing mechanism.

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

```
CF ← Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) ← 0;
```

Flags Affected

The CF flag contains the value of the selected bit before it is cleared. The ZF flag is unaffected. The OF, SF, AF, and PF flags are 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

BTS—Bit Test and Set

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF AB /r	BTS <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag and set.
OF AB /r	BTS <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag and set.
REX.W + OF AB /r	BTS <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag and set.
OF BA /5 ib	BTS <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and set.
OF BA /5 ib	BTS <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and set.
REX.W + OF BA /5 ib	BTS <i>r/m64, imm8</i>	MI	Valid	N.E.	Store selected bit in CF flag and set.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and sets the selected bit in the bit string to 1. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-10.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" in this chapter for more information on this addressing mechanism.

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

```
CF ← Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) ← 1;
```

Flags Affected

The CF flag contains the value of the selected bit before it is set. The ZF flag is unaffected. The OF, SF, AF, and PF flags are 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 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.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

BZHI — Zero High Bits Starting with Specified Bit Position

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS ¹ .LZ.0F38.W0 F5 /r BZHI r32a, r/m32, r32b	RMV	V/V	BMI2	Zero bits in r/m32 starting with the position in r32b, write result to r32a.
VEX.NDS ¹ .LZ.0F38.W1 F5 /r BZHI r64a, r/m64, r64b	RMV	V/N.E.	BMI2	Zero bits in r/m64 starting with the position in r64b, write result to r64a.

NOTES:

- ModRM:r/m is used to encode the first source operand (second operand) and VEX.vvvv encodes the second source operand (third operand).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

Description

BZHI copies the bits of the first source operand (the second operand) into the destination operand (the first operand) and clears the higher bits in the destination according to the INDEX value specified by the second source operand (the third operand). The INDEX is specified by bits 7:0 of the second source operand. The INDEX value is saturated at the value of OperandSize -1. CF is set, if the number contained in the 8 low bits of the third operand is greater than OperandSize -1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```

N ← SRC2[7:0]
DEST ← SRC1
IF (N < OperandSize)
    DEST[OperandSize-1:N] ← 0
FI
IF (N > OperandSize - 1)
    CF ← 1
ELSE
    CF ← 0
FI

```

Flags Affected

ZF, CF and SF flags are updated based on the result. OF flag is cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

```

BZHI:     unsigned __int32 _bzhi_u32(unsigned __int32 src, unsigned __int32 index);
BZHI:     unsigned __int64 _bzhi_u64(unsigned __int64 src, unsigned __int32 index);

```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally
#UD If VEX.W = 1.

CALL—Call Procedure

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
E8 <i>cw</i>	CALL <i>rel16</i>	M	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 <i>cd</i>	CALL <i>rel32</i>	M	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.
FF /2	CALL <i>r/m16</i>	M	N.E.	Valid	Call near, absolute indirect, address given in <i>r/m16</i> .
FF /2	CALL <i>r/m32</i>	M	N.E.	Valid	Call near, absolute indirect, address given in <i>r/m32</i> .
FF /2	CALL <i>r/m64</i>	M	Valid	N.E.	Call near, absolute indirect, address given in <i>r/m64</i> .
9A <i>cd</i>	CALL <i>ptr16:16</i>	D	Invalid	Valid	Call far, absolute, address given in operand.
9A <i>cp</i>	CALL <i>ptr16:32</i>	D	Invalid	Valid	Call far, absolute, address given in operand.
FF /3	CALL <i>m16:16</i>	M	Valid	Valid	Call far, absolute indirect address given in <i>m16:16</i> .
					In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction.
FF /3	CALL <i>m16:32</i>	M	Valid	Valid	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.
REX.W + FF /3	CALL <i>m16:64</i>	M	Valid	N.E.	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = 64-bit offset from far pointer referenced in the instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four types of calls:

- **Near Call** — A call to a procedure in the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intra-segment call.
- **Far Call** — A call to a procedure located in a different segment than the current code segment, sometimes referred to as an inter-segment call.
- **Inter-privilege-level far call** — A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- **Task switch** — A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See "Calling Procedures Using Call and RET" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for additional information on near, far, and inter-privilege-level calls. See Chapter 7, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on performing task switches with the CALL instruction.

Near Call. When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) on the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified by the target operand. The target operand specifies either an absolute offset in the code segment (an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register; this value points to the instruction following the CALL instruction). The CS register is not changed on near calls.

For a near call absolute, an absolute offset is specified indirectly in a general-purpose register or a memory location ($r/m16$, $r/m32$, or $r/m64$). The operand-size attribute determines the size of the target operand (16, 32 or 64 bits). When in 64-bit mode, the operand size for near call (and all near branches) is forced to 64-bits. Absolute offsets are loaded directly into the EIP(RIP) register. If the operand size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits. When accessing an absolute offset indirectly using the stack pointer [ESP] as the base register, the base value used is the value of the ESP before the instruction executes.

A relative offset ($rel/16$ or $rel/32$) is generally specified as a label in assembly code. But at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP(RIP) register. In 64-bit mode the relative offset is always a 32-bit immediate value which is sign extended to 64-bits before it is added to the value in the RIP register for the target calculation. As with absolute offsets, the operand-size attribute determines the size of the target operand (16, 32, or 64 bits). In 64-bit mode the target operand will always be 64-bits because the operand size is forced to 64-bits for near branches.

Far Calls in Real-Address or Virtual-8086 Mode. When executing a far call in real-address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers on the stack for use as a return-instruction pointer. The processor then performs a "far branch" to the code segment and offset specified with the target operand for the called procedure. The target operand specifies an absolute far address either directly with a pointer ($ptr16:16$ or $ptr16:32$) or indirectly with a memory location ($m16:16$ or $m16:32$). With the pointer method, the segment and offset of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

Far Calls in Protected Mode. When the processor is operating in protected mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level
- Far call to a different privilege level (inter-privilege level call)
- Task switch (far call to another task)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer ($ptr16:16$ or $ptr16:32$) or indirectly with a memory location ($m16:16$ or $m16:32$). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register; the offset from the instruction is loaded into the EIP register.

A call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. The target

operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, an optional set of parameters from the calling procedures stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction is similar to executing a call through a call gate. The target operand specifies the segment selector of the task gate for the new task activated by the switch (the offset in the target operand is ignored). The task gate in turn points to the TSS for the new task, which contains the segment selectors for the task's code and stack segments. Note that the TSS also contains the EIP value for the next instruction that was to be executed before the calling task was suspended. This instruction pointer value is loaded into the EIP register to re-start the calling task.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on the mechanics of a task switch.

When you execute a task switch with a CALL instruction, the nested task flag (NT) is set in the EFLAGS register and the new TSS's previous task link field is loaded with the old task's TSS selector. Code is expected to suspend this nested task by executing an IRET instruction which, because the NT flag is set, automatically uses the previous task link to return to the calling task. (See "Task Linking" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from JMP instruction. JMP does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

Mixing 16-Bit and 32-Bit Calls. When making far calls between 16-bit and 32-bit code segments, use a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset can be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values can be pushed on the stack. See Chapter 21, "Mixing 16-Bit and 32-Bit Code," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information.

Far Calls in Compatibility Mode. When the processor is operating in compatibility mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, remaining in compatibility mode
- Far call to the same privilege level, transitioning to 64-bit mode
- Far call to a different privilege level (inter-privilege level call), transitioning to 64-bit mode

Note that a CALL instruction can not be used to cause a task switch in compatibility mode since task switches are not supported in IA-32e mode.

In compatibility mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in compatibility mode is very similar to one carried out in protected mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register and the offset from the instruction is loaded into the EIP register. The difference is that 64-bit mode may be entered. This is specified by the L bit in the new code segment descriptor.

Note that a 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set, causing an entry to 64-bit mode.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. The full value of RSP is used for the offset, of which the upper 32-bits are undefined.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Near/(Far) Calls in 64-bit Mode. When the processor is operating in 64-bit mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, transitioning to compatibility mode
- Far call to the same privilege level, remaining in 64-bit mode
- Far call to a different privilege level (inter-privilege level call), remaining in 64-bit mode

Note that in this mode the CALL instruction can not be used to cause a task switch in 64-bit mode since task switches are not supported in IA-32e mode.

In 64-bit mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in 64-bit mode is very similar to one carried out in compatibility mode. The target operand specifies an absolute far address indirectly with a memory location (*m16:16*, *m16:32* or *m16:64*). The form of CALL with a direct specification of absolute far address is not defined in 64-bit mode. The operand-size attribute determines the size of the offset (16, 32, or 64 bits) in the far address. The new code segment selector and its descriptor are loaded into the CS register; the offset from the instruction is loaded into the EIP register. The new code segment may specify entry either into compatibility or 64-bit mode, based on the L bit value.

A 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can only specify the call gate segment selector indirectly with a memory location (*m16:16*, *m16:32* or *m16:64*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch.

Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. (The full value of RSP is used for the

offset.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Operation

```

IF near call
  THEN IF near relative call
    THEN
      IF OperandSize = 64
        THEN
          tempDEST ← SignExtend(DEST); (* DEST is rel32 *)
          tempRIP ← RIP + tempDEST;
          IF stack not large enough for a 8-byte return address
            THEN #SS(0); Fl;
          Push(RIP);
          RIP ← tempRIP;
        Fl;
      IF OperandSize = 32
        THEN
          tempEIP ← EIP + DEST; (* DEST is rel32 *)
          IF tempEIP is not within code segment limit THEN #GP(0); Fl;
          IF stack not large enough for a 4-byte return address
            THEN #SS(0); Fl;
          Push(EIP);
          EIP ← tempEIP;
        Fl;
      IF OperandSize = 16
        THEN
          tempEIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
          IF tempEIP is not within code segment limit THEN #GP(0); Fl;
          IF stack not large enough for a 2-byte return address
            THEN #SS(0); Fl;
          Push(IP);
          EIP ← tempEIP;
        Fl;
    ELSE (* Near absolute call *)
      IF OperandSize = 64
        THEN
          tempRIP ← DEST; (* DEST is r/m64 *)
          IF stack not large enough for a 8-byte return address
            THEN #SS(0); Fl;
          Push(RIP);
          RIP ← tempRIP;
        Fl;
      IF OperandSize = 32
        THEN
          tempEIP ← DEST; (* DEST is r/m32 *)
          IF tempEIP is not within code segment limit THEN #GP(0); Fl;
          IF stack not large enough for a 4-byte return address
            THEN #SS(0); Fl;
          Push(EIP);
          EIP ← tempEIP;
        Fl;
    ENDIF;
  ENDIF;
ENDIF;

```

INSTRUCTION SET REFERENCE, A-M

```
Fl;
IF OperandSize = 16
THEN
    tempEIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
    IF tempEIP is not within code segment limit THEN #GP(0); Fl;
    IF stack not large enough for a 2-byte return address
        THEN #SS(0); Fl;
    Push(IP);
    EIP ← tempEIP;

Fl;
Fl;rel/abs
Fl; near

IF far call and (PE = 0 or (PE = 1 and VM = 1)) (* Real-address or virtual-8086 mode *)
THEN
    IF OperandSize = 32
    THEN
        IF stack not large enough for a 6-byte return address
            THEN #SS(0); Fl;
        IF DEST[31:16] is not zero THEN #GP(0); Fl;
        Push(CS); (* Padded with 16 high-order bits *)
        Push(EIP);
        CS ← DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
        EIP ← DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
    ELSE (* OperandSize = 16 *)
        IF stack not large enough for a 4-byte return address
            THEN #SS(0); Fl;
        Push(CS);
        Push(IP);
        CS ← DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
        EIP ← DEST[15:0]; (* DEST is ptr16:16 or [m16:16]; clear upper 16 bits *)
    Fl;
Fl;

IF far call and (PE = 1 and VM = 0) (* Protected mode or IA-32e Mode, not virtual-8086 mode*)
THEN
    IF segment selector in target operand NULL
        THEN #GP(0); Fl;
    IF segment selector index not within descriptor table limits
        THEN #GP(new code segment selector); Fl;
    Read type and access rights of selected segment descriptor;
    IF IA32_EFER.LMA = 0
    THEN
        IF segment type is not a conforming or nonconforming code segment, call
        gate, task gate, or TSS
            THEN #GP(segment selector); Fl;
    ELSE
        IF segment type is not a conforming or nonconforming code segment or
        64-bit call gate,
            THEN #GP(segment selector); Fl;
    Fl;
Depending on type and access rights:
GO TO CONFORMING-CODE-SEGMENT;
GO TO NONCONFORMING-CODE-SEGMENT;
```

```

GO TO CALL-GATE;
GO TO TASK-GATE;
GO TO TASK-STATE-SEGMENT;
Fl;

CONFORMING-CODE-SEGMENT:
IF L bit = 1 and D bit = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); Fl;
IF DPL > CPL
    THEN #GP(new code segment selector); Fl;
IF segment not present
    THEN #NP(new code segment selector); Fl;
IF stack not large enough for return address
    THEN #SS(0); Fl;
tempEIP ← DEST(Offset);
IF OperandSize = 16
    THEN
        tempEIP ← tempEIP AND 0000FFFFH; Fl; (* Clear upper 16 bits *)
IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
segment limit)
    THEN #GP(0); Fl;
IF tempEIP is non-canonical
    THEN #GP(0); Fl;
IF OperandSize = 32
    THEN
        Push(CS); (* Padded with 16 high-order bits *)
        Push(EIP);
        CS ← DEST(CodeSegmentSelector);
        (* Segment descriptor information also loaded *)
        CS(RPL) ← CPL;
        EIP ← tempEIP;
    ELSE
        IF OperandSize = 16
            THEN
                Push(CS);
                Push(IP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                EIP ← tempEIP;
            ELSE (* OperandSize = 64 *)
                Push(CS); (* Padded with 48 high-order bits *)
                Push(RIP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                RIP ← tempEIP;
        Fl;
    Fl;
END;

```

```

NONCONFORMING-CODE-SEGMENT:
IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); Fl;

```

INSTRUCTION SET REFERENCE, A-M

```
IF (RPL > CPL) or (DPL ≠ CPL)
    THEN #GP(new code segment selector); Fl;
IF segment not present
    THEN #NP(new code segment selector); Fl;
IF stack not large enough for return address
    THEN #SS(0); Fl;
tempEIP ← DEST(Offset);
IF OperandSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH; Fl; (* Clear upper 16 bits *)
IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
segment limit)
    THEN #GP(0); Fl;
IF tempEIP is non-canonical
    THEN #GP(0); Fl;
IF OperandSize = 32
    THEN
        Push(CS); (* Padded with 16 high-order bits *)
        Push(EIP);
        CS ← DEST(CodeSegmentSelector);
        (* Segment descriptor information also loaded *)
        CS(RPL) ← CPL;
        EIP ← tempEIP;
    ELSE
        IF OperandSize = 16
            THEN
                Push(CS);
                Push(IP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                EIP ← tempEIP;
            ELSE (* OperandSize = 64 *)
                Push(CS); (* Padded with 48 high-order bits *)
                Push(RIP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                RIP ← tempEIP;
        Fl;
    Fl;
END;
```

CALL-GATE:

```
IF call gate (DPL < CPL) or (RPL > DPL)
    THEN #GP(call-gate selector); Fl;
IF call gate not present
    THEN #NP(call-gate selector); Fl;
IF call-gate code-segment selector is NULL
    THEN #GP(0); Fl;
IF call-gate code-segment selector index is outside descriptor table limits
    THEN #GP(call-gate code-segment selector); Fl;
Read call-gate code-segment descriptor;
IF call-gate code-segment descriptor does not indicate a code segment
or call-gate code-segment descriptor DPL > CPL
```

```

    THEN #GP(call-gate code-segment selector); Fl;
IF IA32_EFER.LMA = 1 AND (call-gate code-segment descriptor is
not a 64-bit code segment or call-gate code-segment descriptor has both L-bit and D-bit set)
    THEN #GP(call-gate code-segment selector); Fl;
IF call-gate code segment not present
    THEN #NP(call-gate code-segment selector); Fl;
IF call-gate code segment is non-conforming and DPL < CPL
    THEN go to MORE-PRIVILEGE;
    ELSE go to SAME-PRIVILEGE;
Fl;
END;

```

MORE-PRIVILEGE:

```

IF current TSS is 32-bit
    THEN
        TSSstackAddress ← (new code-segment DPL * 8) + 4;
        IF (TSSstackAddress + 5) > current TSS limit
            THEN #TS(current TSS selector); Fl;
        NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
        NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
    ELSE
        IF current TSS is 16-bit
            THEN
                TSSstackAddress ← (new code-segment DPL * 4) + 2
                IF (TSSstackAddress + 3) > current TSS limit
                    THEN #TS(current TSS selector); Fl;
                NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
            ELSE (* current TSS is 64-bit *)
                TSSstackAddress ← (new code-segment DPL * 8) + 4;
                IF (TSSstackAddress + 7) > current TSS limit
                    THEN #TS(current TSS selector); Fl;
                NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
                NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
        Fl;

```

```

IF IA32_EFER.LMA = 0 and NewSS is NULL
    THEN #TS(NewSS); Fl;
Read new code-segment descriptor and new stack-segment descriptor;
IF IA32_EFER.LMA = 0 and (NewSS RPL ≠ new code-segment DPL
or new stack-segment DPL ≠ new code-segment DPL or new stack segment is not a
writable data segment)
    THEN #TS(NewSS); Fl

```

```

IF IA32_EFER.LMA = 0 and new stack segment not present
    THEN #SS(NewSS); Fl;

```

```

IF CallGateSize = 32
    THEN

```

```

        IF new stack does not have room for parameters plus 16 bytes
            THEN #SS(NewSS); Fl;
        IF CallGate(InstructionPointer) not within new code-segment limit
            THEN #GP(0); Fl;
        SS ← newSS; (* Segment descriptor information also loaded *)
        ESP ← newESP;
        CS:EIP ← CallGate(CS:InstructionPointer);

```

INSTRUCTION SET REFERENCE, A-M

```
(* Segment descriptor information also loaded *)
Push(oldSS:oldESP); (* From calling procedure *)
temp ← parameter count from call gate, masked to 5 bits;
Push(parameters from calling procedure's stack, temp)
Push(oldCS:oldEIP); (* Return address to calling procedure *)
ELSE
  IF CallGateSize = 16
    THEN
      IF new stack does not have room for parameters plus 8 bytes
        THEN #SS(NewSS); Fl;
      IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
        THEN #GP(0); Fl;
      SS ← newSS; (* Segment descriptor information also loaded *)
      ESP ← newESP;
      CS:IP ← CallGate(CS:InstructionPointer);
      (* Segment descriptor information also loaded *)
      Push(oldSS:oldESP); (* From calling procedure *)
      temp ← parameter count from call gate, masked to 5 bits;
      Push(parameters from calling procedure's stack, temp)
      Push(oldCS:oldEIP); (* Return address to calling procedure *)
    ELSE (* CallGateSize = 64 *)
      IF pushing 32 bytes on the stack would use a non-canonical address
        THEN #SS(NewSS); Fl;
      IF (CallGate(InstructionPointer) is non-canonical)
        THEN #GP(0); Fl;
      SS ← NewSS; (* NewSS is NULL)
      RSP ← NewESP;
      CS:IP ← CallGate(CS:InstructionPointer);
      (* Segment descriptor information also loaded *)
      Push(oldSS:oldESP); (* From calling procedure *)
      Push(oldCS:oldEIP); (* Return address to calling procedure *)
    Fl;
  Fl;
CPL ← CodeSegment(DPL)
CS(RPL) ← CPL
END;
```

SAME-PRIVILEGE:

```
IF CallGateSize = 32
  THEN
    IF stack does not have room for 8 bytes
      THEN #SS(0); Fl;
    IF CallGate(InstructionPointer) not within code segment limit
      THEN #GP(0); Fl;
    CS:EIP ← CallGate(CS:EIP) (* Segment descriptor information also loaded *)
    Push(oldCS:oldEIP); (* Return address to calling procedure *)
  ELSE
    If CallGateSize = 16
      THEN
        IF stack does not have room for 4 bytes
          THEN #SS(0); Fl;
        IF CallGate(InstructionPointer) not within code segment limit
          THEN #GP(0); Fl;
        CS:IP ← CallGate(CS:instruction pointer);
```

```

(* Segment descriptor information also loaded *)
Push(oldCS:oldIP); (* Return address to calling procedure *)
ELSE (* CallGateSize = 64)
    IF pushing 16 bytes on the stack touches non-canonical addresses
        THEN #SS(0); Fl;
    IF RIP non-canonical
        THEN #GP(0); Fl;
    CS:IP ← CallGate(CS:instruction pointer);
(* Segment descriptor information also loaded *)
Push(oldCS:oldIP); (* Return address to calling procedure *)

Fl;
Fl;
CS(RPL) ← CPL
END;

TASK-GATE:
IF task gate DPL < CPL or RPL
    THEN #GP(task gate selector); Fl;
IF task gate not present
    THEN #NP(task gate selector); Fl;
Read the TSS segment selector in the task-gate descriptor;
IF TSS segment selector local/global bit is set to local
or index not within GDT limits
    THEN #GP(TSS selector); Fl;
Access TSS descriptor in GDT;
IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
    THEN #GP(TSS selector); Fl;
IF TSS not present
    THEN #NP(TSS selector); Fl;
SWITCH-TASKS (with nesting) to TSS;
IF EIP not within code segment limit
    THEN #GP(0); Fl;
END;

TASK-STATE-SEGMENT:
IF TSS DPL < CPL or RPL
or TSS descriptor indicates TSS not available
    THEN #GP(TSS selector); Fl;
IF TSS is not present
    THEN #NP(TSS selector); Fl;
SWITCH-TASKS (with nesting) to TSS;
IF EIP not within code segment limit
    THEN #GP(0); Fl;
END;

```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions

- #GP(0) If the target offset in destination operand is beyond the new code segment limit.
- If the segment selector in the destination operand is NULL.
- If the code segment selector in the gate is NULL.

	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 a code segment or gate or TSS selector index is outside descriptor table limits. If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment. If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment's segment selector is greater than the CPL. If the DPL for a conforming-code segment is greater than the CPL. If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector. If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment. If the segment selector from a call gate is beyond the descriptor table limits. If the DPL for a code-segment obtained from a call gate is greater than the CPL. If the segment selector for a TSS has its local/global bit set for local. If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.
#SS(selector)	If a memory operand effective address is outside the SS segment limit. If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs. If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present. If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.
#NP(selector)	If a code segment, data segment, stack segment, call gate, task gate, or TSS is not present.
#TS(selector)	If the new stack segment selector and ESP are beyond the end of the TSS. If the new stack segment selector is NULL. If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed. If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor. If the new stack segment is not a writable data segment. If segment-selector index for stack segment is outside descriptor table limits.
#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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the target offset is beyond the code segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

#GP(selector)	If a memory address accessed by the selector is in non-canonical space.
#GP(0)	If the target offset in the destination operand is non-canonical.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical. If target offset in destination operand is non-canonical. If the segment selector in the destination operand is NULL. If the code segment selector in the 64-bit gate is NULL.
#GP(selector)	If code segment or 64-bit call gate is outside descriptor table limits. If code segment or 64-bit call gate overlaps non-canonical space. If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, or 64-bit call gate. If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and has both the D-bit and the L-bit set. If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL. If the DPL for a conforming-code segment is greater than the CPL. If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate. If the upper type field of a 64-bit call gate is not 0x0. If the segment selector from a 64-bit call gate is beyond the descriptor table limits. If the DPL for a code-segment obtained from a 64-bit call gate is greater than the CPL. If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear. If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.
#SS(0)	If pushing the return offset or CS selector onto the stack exceeds the bounds of the stack segment when no stack switch occurs. If a memory operand effective address is outside the SS segment limit. If the stack address is in a non-canonical form.
#SS(selector)	If pushing the old values of SS selector, stack pointer, EFLAGS, CS selector, offset, or error code onto the stack violates the canonical boundary when a stack switch occurs.
#NP(selector)	If a code segment or 64-bit call gate is not present.
#TS(selector)	If the load of the new RSP exceeds the limit of the TSS.
#UD	(64-bit mode only) If a far call is direct to an absolute address in memory. If the LOCK prefix is used.
#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.

CBW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
98	CBW		NP	Valid	AX ← sign-extend of AL.
98	CWDE		NP	Valid	EAX ← sign-extend of AX.
REX.W + 98	CDQE		NP	Valid	RAX ← sign-extend of EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Double the size of the source operand by means of sign extension. The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to doubleword) instruction copies the sign (bit 15) of the word in the AX register into the high 16 bits of the EAX register.

CBW and CWDE reference the same opcode. The CBW instruction is intended for use when the operand-size attribute is 16; CWDE is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size. Others may treat these two mnemonics as synonyms (CBW/CWDE) and use the setting of the operand-size attribute to determine the size of values to be converted.

In 64-bit mode, the default operation size is the size of the destination register. Use of the REX.W prefix promotes this instruction (CDQE when promoted) to operate on 64-bit operands. In which case, CDQE copies the sign (bit 31) of the doubleword in the EAX register into the high 32 bits of RAX.

Operation

```

IF OperandSize = 16 (* Instruction = CBW *)
  THEN
    AX ← SignExtend(AL);
  ELSE IF (OperandSize = 32, Instruction = CWDE)
    EAX ← SignExtend(AX); Fl;
  ELSE (* 64-Bit Mode, OperandSize = 64, Instruction = CDQE*)
    RAX ← SignExtend(EAX);
Fl;

```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

CLC—Clear Carry Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
F8	CLC	NP	Valid	Valid	Clear CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Clears the CF flag in the EFLAGS register. Operation is the same in all non-64-bit modes and 64-bit mode.

Operation

$CF \leftarrow 0;$

Flags Affected

The CF flag is set to 0. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

CLD—Clear Direction Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
FC	CLD	NP	Valid	Valid	Clear DF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI). Operation is the same in all non-64-bit modes and 64-bit mode.

Operation

$DF \leftarrow 0;$

Flags Affected

The DF flag is set to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

CLFLUSH—Flush Cache Line

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF AE /7	CLFLUSH <i>m8</i>	M	Valid	Valid	Flushes cache line containing <i>m8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Invalidates the cache line that contains the linear address specified with the source operand from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the cache coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty) it is written to memory before invalidation. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (bit 19 of the EDX register, see “CPUID—CPU Identification” in this chapter). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCH_h instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH_h instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

CLFLUSH is only ordered by the MFENCE instruction. It is not guaranteed to be ordered by any other fencing or serializing instructions or by another CLFLUSH instruction. For example, software can use an MFENCE instruction to ensure that previous stores are included in the write-back.

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

CLFLUSH operation is the same in non-64-bit modes and 64-bit mode.

Operation

Flush_Cache_Line(SRC);

Intel C/C++ Compiler Intrinsic Equivalents

CLFLUSH: void _mm_clflush(void const *p)

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 CPUID.01H:EDX.CLFSH[bit 19] = 0.
If the LOCK prefix is used. |

If instruction prefix is 66H, F2H or F3H.

Real-Address Mode Exceptions

- #GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
- #UD If CPUID.01H:EDX.CLFSH[bit 19] = 0.
 - If the LOCK prefix is used.
- If instruction prefix is 66H, F2H or F3H.

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.
- #PF(fault-code) For a page fault.
- #UD If CPUID.01H:EDX.CLFSH[bit 19] = 0.
 - If the LOCK prefix is used.
- If instruction prefix is 66H, F2H or F3H.

CLI – Clear Interrupt Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
FA	CLI	NP	Valid	Valid	Clear interrupt flag; interrupts disabled when interrupt flag cleared.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

If protected-mode virtual interrupts are not enabled, CLI clears the IF flag in the EFLAGS register. No other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no affect on the generation of exceptions and NMI interrupts.

When protected-mode virtual interrupts are enabled, CPL is 3, and IOPL is less than 3; CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected. Table 3-6 indicates the action of the CLI instruction depending on the processor operating mode and the CPL/IOPL of the running program or procedure.

CLI operation is the same in non-64-bit modes and 64-bit mode.

Table 3-6. Decision Table for CLI Results

PE	VM	IOPL	CPL	PVI	VIF	VME	CLI Result
0	X	X	X	X	X	X	IF = 0
1	0	\geq CPL	X	X	X	X	IF = 0
1	0	< CPL	3	1	X	X	VIF = 0
1	0	< CPL	< 3	X	X	X	GP Fault
1	0	< CPL	X	0	X	X	GP Fault
1	1	3	X	X	X	X	IF = 0
1	1	< 3	X	X	X	1	VIF = 0
1	1	< 3	X	X	X	0	GP Fault

NOTES:

* X = This setting has no impact.

Operation

```

IF PE = 0
  THEN
    IF  $\leftarrow$  0; (* Reset Interrupt Flag *)
  ELSE
    IF VM = 0;
      THEN
        IF IOPL  $\geq$  CPL
          THEN
            IF  $\leftarrow$  0; (* Reset Interrupt Flag *)
          ELSE
            IF ((IOPL < CPL) and (CPL = 3) and (PVI = 1))
              THEN
                VIF  $\leftarrow$  0; (* Reset Virtual Interrupt Flag *)
              ELSE

```

```

        #GP(0);
    Fl;
    Fl;
ELSE (* VM = 1 *)
    IF IOPL = 3
        THEN
            IF ← 0; (* Reset Interrupt Flag *)
        ELSE
            IF (IOPL < 3) AND (VME = 1)
                THEN
                    VIF ← 0; (* Reset Virtual Interrupt Flag *)
                ELSE
                    #GP(0);
    Fl;
    Fl;
Fl;

```

Flags Affected

If protected-mode virtual interrupts are not enabled, IF is set to 0 if the CPL is equal to or less than the IOPL; otherwise, it is not affected. The other flags in the EFLAGS register are unaffected.

When protected-mode virtual interrupts are enabled, CPL is 3, and IOPL is less than 3; CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected.

Protected Mode Exceptions

- #GP(0) If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #GP(0) If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.
- #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #GP(0) If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.
- #UD If the LOCK prefix is used.

CLTS—Clear Task-Switched Flag in CRO

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
OF 06	CLTS	NP	Valid	Valid	Clears TS flag in CRO.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Clears the task-switched (TS) flag in the CR0 register. This instruction is intended for use in operating-system procedures. It is a privileged instruction that can only be executed at a CPL of 0. It is allowed to be executed in real-address mode to allow initialization for protected mode.

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the section titled “Control Registers” in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for more information about this flag.

CLTS operation is the same in non-64-bit modes and 64-bit mode.

See Chapter 25, “VMX Non-Root Operation,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

$\text{CR0.TS}[bit\ 3] \leftarrow 0;$

Flags Affected

The TS flag in CR0 register is cleared.

Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #GP(0) CLTS is not recognized in virtual-8086 mode.
- #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #GP(0) If the CPL is greater than 0.
- #UD If the LOCK prefix is used.

CMC—Complement Carry Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
F5	CMC	NP	Valid	Valid	Complement CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Complements the CF flag in the EFLAGS register. CMC operation is the same in non-64-bit modes and 64-bit mode.

Operation

EFLAGS.CF[bit 0] ← NOT EFLAGS.CF[bit 0];

Flags Affected

The CF flag contains the complement of its original value. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

CMOVcc—Conditional Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 47 /r	CMOVA r16, r/m16	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
0F 47 /r	CMOVA r32, r/m32	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
REX.W + 0F 47 /r	CMOVA r64, r/m64	RM	Valid	N.E.	Move if above (CF=0 and ZF=0).
0F 43 /r	CMOVAE r16, r/m16	RM	Valid	Valid	Move if above or equal (CF=0).
0F 43 /r	CMOVAE r32, r/m32	RM	Valid	Valid	Move if above or equal (CF=0).
REX.W + 0F 43 /r	CMOVAE r64, r/m64	RM	Valid	N.E.	Move if above or equal (CF=0).
0F 42 /r	CMOVB r16, r/m16	RM	Valid	Valid	Move if below (CF=1).
0F 42 /r	CMOVB r32, r/m32	RM	Valid	Valid	Move if below (CF=1).
REX.W + 0F 42 /r	CMOVB r64, r/m64	RM	Valid	N.E.	Move if below (CF=1).
0F 46 /r	CMOVBE r16, r/m16	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
0F 46 /r	CMOVBE r32, r/m32	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
REX.W + 0F 46 /r	CMOVBE r64, r/m64	RM	Valid	N.E.	Move if below or equal (CF=1 or ZF=1).
0F 42 /r	CMOVC r16, r/m16	RM	Valid	Valid	Move if carry (CF=1).
0F 42 /r	CMOVC r32, r/m32	RM	Valid	Valid	Move if carry (CF=1).
REX.W + 0F 42 /r	CMOVC r64, r/m64	RM	Valid	N.E.	Move if carry (CF=1).
0F 44 /r	CMOVE r16, r/m16	RM	Valid	Valid	Move if equal (ZF=1).
0F 44 /r	CMOVE r32, r/m32	RM	Valid	Valid	Move if equal (ZF=1).
REX.W + 0F 44 /r	CMOVE r64, r/m64	RM	Valid	N.E.	Move if equal (ZF=1).
0F 4F /r	CMOVG r16, r/m16	RM	Valid	Valid	Move if greater (ZF=0 and SF=OF).
0F 4F /r	CMOVG r32, r/m32	RM	Valid	Valid	Move if greater (ZF=0 and SF=OF).
REX.W + 0F 4F /r	CMOVG r64, r/m64	RM	V/N.E.	NA	Move if greater (ZF=0 and SF=OF).
0F 4D /r	CMOVGE r16, r/m16	RM	Valid	Valid	Move if greater or equal (SF=OF).
0F 4D /r	CMOVGE r32, r/m32	RM	Valid	Valid	Move if greater or equal (SF=OF).
REX.W + 0F 4D /r	CMOVGE r64, r/m64	RM	Valid	N.E.	Move if greater or equal (SF=OF).
0F 4C /r	CMOVL r16, r/m16	RM	Valid	Valid	Move if less (SF≠ OF).
0F 4C /r	CMOVL r32, r/m32	RM	Valid	Valid	Move if less (SF≠ OF).
REX.W + 0F 4C /r	CMOVL r64, r/m64	RM	Valid	N.E.	Move if less (SF≠ OF).
0F 4E /r	CMOVLE r16, r/m16	RM	Valid	Valid	Move if less or equal (ZF=1 or SF≠ OF).
0F 4E /r	CMOVLE r32, r/m32	RM	Valid	Valid	Move if less or equal (ZF=1 or SF≠ OF).
REX.W + 0F 4E /r	CMOVLE r64, r/m64	RM	Valid	N.E.	Move if less or equal (ZF=1 or SF≠ OF).
0F 46 /r	CMOVNA r16, r/m16	RM	Valid	Valid	Move if not above (CF=1 or ZF=1).
0F 46 /r	CMOVNA r32, r/m32	RM	Valid	Valid	Move if not above (CF=1 or ZF=1).
REX.W + 0F 46 /r	CMOVNA r64, r/m64	RM	Valid	N.E.	Move if not above (CF=1 or ZF=1).
0F 42 /r	CMOVNAE r16, r/m16	RM	Valid	Valid	Move if not above or equal (CF=1).
0F 42 /r	CMOVNAE r32, r/m32	RM	Valid	Valid	Move if not above or equal (CF=1).
REX.W + 0F 42 /r	CMOVNAE r64, r/m64	RM	Valid	N.E.	Move if not above or equal (CF=1).
0F 43 /r	CMOVNB r16, r/m16	RM	Valid	Valid	Move if not below (CF=0).
0F 43 /r	CMOVNB r32, r/m32	RM	Valid	Valid	Move if not below (CF=0).
REX.W + 0F 43 /r	CMOVNB r64, r/m64	RM	Valid	N.E.	Move if not below (CF=0).
0F 47 /r	CMOVNBE r16, r/m16	RM	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 47 /r	CMOVNBE r32, r/m32	RM	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
REX.W + OF 47 /r	CMOVNBE r64, r/m64	RM	Valid	N.E.	Move if not below or equal (CF=0 and ZF=0).
OF 43 /r	CMOVNC r16, r/m16	RM	Valid	Valid	Move if not carry (CF=0).
OF 43 /r	CMOVNC r32, r/m32	RM	Valid	Valid	Move if not carry (CF=0).
REX.W + OF 43 /r	CMOVNC r64, r/m64	RM	Valid	N.E.	Move if not carry (CF=0).
OF 45 /r	CMOVNE r16, r/m16	RM	Valid	Valid	Move if not equal (ZF=0).
OF 45 /r	CMOVNE r32, r/m32	RM	Valid	Valid	Move if not equal (ZF=0).
REX.W + OF 45 /r	CMOVNE r64, r/m64	RM	Valid	N.E.	Move if not equal (ZF=0).
OF 4E /r	CMOVNG r16, r/m16	RM	Valid	Valid	Move if not greater (ZF=1 or SF≠ OF).
OF 4E /r	CMOVNG r32, r/m32	RM	Valid	Valid	Move if not greater (ZF=1 or SF≠ OF).
REX.W + OF 4E /r	CMOVNG r64, r/m64	RM	Valid	N.E.	Move if not greater (ZF=1 or SF≠ OF).
OF 4C /r	CMOVNGE r16, r/m16	RM	Valid	Valid	Move if not greater or equal (SF≠ OF).
OF 4C /r	CMOVNGE r32, r/m32	RM	Valid	Valid	Move if not greater or equal (SF≠ OF).
REX.W + OF 4C /r	CMOVNGE r64, r/m64	RM	Valid	N.E.	Move if not greater or equal (SF≠ OF).
OF 4D /r	CMOVNL r16, r/m16	RM	Valid	Valid	Move if not less (SF=OF).
OF 4D /r	CMOVNL r32, r/m32	RM	Valid	Valid	Move if not less (SF=OF).
REX.W + OF 4D /r	CMOVNL r64, r/m64	RM	Valid	N.E.	Move if not less (SF=OF).
OF 4F /r	CMOVNLE r16, r/m16	RM	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
OF 4F /r	CMOVNLE r32, r/m32	RM	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
REX.W + OF 4F /r	CMOVNLE r64, r/m64	RM	Valid	N.E.	Move if not less or equal (ZF=0 and SF=OF).
OF 41 /r	CMOVNO r16, r/m16	RM	Valid	Valid	Move if not overflow (OF=0).
OF 41 /r	CMOVNO r32, r/m32	RM	Valid	Valid	Move if not overflow (OF=0).
REX.W + OF 41 /r	CMOVNO r64, r/m64	RM	Valid	N.E.	Move if not overflow (OF=0).
OF 4B /r	CMOVNP r16, r/m16	RM	Valid	Valid	Move if not parity (PF=0).
OF 4B /r	CMOVNP r32, r/m32	RM	Valid	Valid	Move if not parity (PF=0).
REX.W + OF 4B /r	CMOVNP r64, r/m64	RM	Valid	N.E.	Move if not parity (PF=0).
OF 49 /r	CMOVNS r16, r/m16	RM	Valid	Valid	Move if not sign (SF=0).
OF 49 /r	CMOVNS r32, r/m32	RM	Valid	Valid	Move if not sign (SF=0).
REX.W + OF 49 /r	CMOVNS r64, r/m64	RM	Valid	N.E.	Move if not sign (SF=0).
OF 45 /r	CMOVNZ r16, r/m16	RM	Valid	Valid	Move if not zero (ZF=0).
OF 45 /r	CMOVNZ r32, r/m32	RM	Valid	Valid	Move if not zero (ZF=0).
REX.W + OF 45 /r	CMOVNZ r64, r/m64	RM	Valid	N.E.	Move if not zero (ZF=0).
OF 40 /r	CMOVO r16, r/m16	RM	Valid	Valid	Move if overflow (OF=1).
OF 40 /r	CMOVO r32, r/m32	RM	Valid	Valid	Move if overflow (OF=1).
REX.W + OF 40 /r	CMOVO r64, r/m64	RM	Valid	N.E.	Move if overflow (OF=1).
OF 4A /r	CMOVP r16, r/m16	RM	Valid	Valid	Move if parity (PF=1).
OF 4A /r	CMOVP r32, r/m32	RM	Valid	Valid	Move if parity (PF=1).
REX.W + OF 4A /r	CMOVP r64, r/m64	RM	Valid	N.E.	Move if parity (PF=1).
OF 4A /r	CMOVPE r16, r/m16	RM	Valid	Valid	Move if parity even (PF=1).
OF 4A /r	CMOVPE r32, r/m32	RM	Valid	Valid	Move if parity even (PF=1).
REX.W + OF 4A /r	CMOVPE r64, r/m64	RM	Valid	N.E.	Move if parity even (PF=1).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 4B /r	CMOVPO r16, r/m16	RM	Valid	Valid	Move if parity odd (PF=0).
OF 4B /r	CMOVPO r32, r/m32	RM	Valid	Valid	Move if parity odd (PF=0).
REX.W + OF 4B /r	CMOVPO r64, r/m64	RM	Valid	N.E.	Move if parity odd (PF=0).
OF 48 /r	CMOVS r16, r/m16	RM	Valid	Valid	Move if sign (SF=1).
OF 48 /r	CMOVS r32, r/m32	RM	Valid	Valid	Move if sign (SF=1).
REX.W + OF 48 /r	CMOVS r64, r/m64	RM	Valid	N.E.	Move if sign (SF=1).
OF 44 /r	CMOVZ r16, r/m16	RM	Valid	Valid	Move if zero (ZF=1).
OF 44 /r	CMOVZ r32, r/m32	RM	Valid	Valid	Move if zero (ZF=1).
REX.W + OF 44 /r	CMOVZ r64, r/m64	RM	Valid	N.E.	Move if zero (ZF=1).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

The CMOVcc instructions check the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and perform a move operation if the flags are in a specified state (or condition). A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOVcc instruction.

These instructions can move 16-bit, 32-bit or 64-bit values from memory to a general-purpose register or from one general-purpose register to another. Conditional moves of 8-bit register operands are not supported.

The condition for each CMOVcc mnemonic is given in the description column of the above table. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the CMOVA (conditional move if above) instruction and the CMOVNBE (conditional move if not below or equal) instruction are alternate mnemonics for the opcode OF 47H.

The CMOVcc instructions were introduced in P6 family processors; however, these instructions may not be supported by all IA-32 processors. Software can determine if the CMOVcc instructions are supported by checking the processor’s feature information with the CPUID instruction (see “CPUID—CPU Identification” in this chapter).

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

Operation

```

temp ← SRC
IF condition TRUE
  THEN
    DEST ← temp;
  FI;
ELSE
  IF (OperandSize = 32 and IA-32e mode active)
    THEN
      DEST[63:32] ← 0;
    FI;

```

Fl;

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.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

CMP—Compare Two Operands

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
3C <i>ib</i>	CMP AL, <i>imm8</i>	I	Valid	Valid	Compare <i>imm8</i> with AL.
3D <i>iw</i>	CMP AX, <i>imm16</i>	I	Valid	Valid	Compare <i>imm16</i> with AX.
3D <i>id</i>	CMP EAX, <i>imm32</i>	I	Valid	Valid	Compare <i>imm32</i> with EAX.
REX.W + 3D <i>id</i>	CMP RAX, <i>imm32</i>	I	Valid	N.E.	Compare <i>imm32 sign-extended to 64-bits</i> with RAX.
80 /7 <i>ib</i>	CMP <i>r/m8, imm8</i>	MI	Valid	Valid	Compare <i>imm8</i> with <i>r/m8</i> .
REX + 80 /7 <i>ib</i>	CMP <i>r/m8*, imm8</i>	MI	Valid	N.E.	Compare <i>imm8</i> with <i>r/m8</i> .
81 /7 <i>iw</i>	CMP <i>r/m16, imm16</i>	MI	Valid	Valid	Compare <i>imm16</i> with <i>r/m16</i> .
81 /7 <i>id</i>	CMP <i>r/m32, imm32</i>	MI	Valid	Valid	Compare <i>imm32</i> with <i>r/m32</i> .
REX.W + 81 /7 <i>id</i>	CMP <i>r/m64, imm32</i>	MI	Valid	N.E.	Compare <i>imm32 sign-extended to 64-bits</i> with <i>r/m64</i> .
83 /7 <i>ib</i>	CMP <i>r/m16, imm8</i>	MI	Valid	Valid	Compare <i>imm8</i> with <i>r/m16</i> .
83 /7 <i>ib</i>	CMP <i>r/m32, imm8</i>	MI	Valid	Valid	Compare <i>imm8</i> with <i>r/m32</i> .
REX.W + 83 /7 <i>ib</i>	CMP <i>r/m64, imm8</i>	MI	Valid	N.E.	Compare <i>imm8</i> with <i>r/m64</i> .
38 / <i>r</i>	CMP <i>r/m8, r8</i>	MR	Valid	Valid	Compare <i>r8</i> with <i>r/m8</i> .
REX + 38 / <i>r</i>	CMP <i>r/m8*, r8*</i>	MR	Valid	N.E.	Compare <i>r8</i> with <i>r/m8</i> .
39 / <i>r</i>	CMP <i>r/m16, r16</i>	MR	Valid	Valid	Compare <i>r16</i> with <i>r/m16</i> .
39 / <i>r</i>	CMP <i>r/m32, r32</i>	MR	Valid	Valid	Compare <i>r32</i> with <i>r/m32</i> .
REX.W + 39 / <i>r</i>	CMP <i>r/m64, r64</i>	MR	Valid	N.E.	Compare <i>r64</i> with <i>r/m64</i> .
3A / <i>r</i>	CMP <i>r8, r/m8</i>	RM	Valid	Valid	Compare <i>r/m8</i> with <i>r8</i> .
REX + 3A / <i>r</i>	CMP <i>r8*, r/m8*</i>	RM	Valid	N.E.	Compare <i>r/m8</i> with <i>r8</i> .
3B / <i>r</i>	CMP <i>r16, r/m16</i>	RM	Valid	Valid	Compare <i>r/m16</i> with <i>r16</i> .
3B / <i>r</i>	CMP <i>r32, r/m32</i>	RM	Valid	Valid	Compare <i>r/m32</i> with <i>r32</i> .
REX.W + 3B / <i>r</i>	CMP <i>r64, r/m64</i>	RM	Valid	N.E.	Compare <i>r/m64</i> with <i>r64</i> .

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (w)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AE/EAX/RAX	imm8	NA	NA

Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The condition codes used by the Jcc, CMOVcc, and SETcc instructions are based on the results of a CMP instruction. Appendix B, "EFLAGS Condition Codes," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, shows the relationship of the status flags and the condition codes.

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

Operation

```
temp ← SRC1 – SignExtend(SRC2);
ModifyStatusFlags; (* Modify status flags in the same manner as the SUB instruction*)
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the result.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

CMPPD—Compare Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
66 0F C2 /r ib CMPPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE2	Compare packed double-precision floating-point values in <i>xmm2/m128</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.
VEX.NDS.128.66.0F.WIG C2 /r ib VCMPPD <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Compare packed double-precision floating-point values in <i>xmm3/m128</i> and <i>xmm2</i> using bits 4:0 of <i>imm8</i> as a comparison predicate.
VEX.NDS.256.66.0F.WIG C2 /r ib VCMPPD <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Compare packed double-precision floating-point values in <i>ymm3/m256</i> and <i>ymm2</i> using bits 4:0 of <i>imm8</i> as a comparison predicate.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed double-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false). The sign of zero is ignored for comparisons, so that -0.0 is equal to +0.0.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand.

Table 3-7. Comparison Predicate for CMPPD and CMPPS Instructions

Predic- cate	imm8 Encoding	Description	Relation where: A Is 1st Operand B Is 2nd Operand	Emulation	Result if NaN Operand	QNaN Oper-and Signals Invalid
EQ	000B	Equal	A = B		False	No
LT	001B	Less-than	A < B		False	Yes
LE	010B	Less-than-or-equal	A ≤ B		False	Yes
		Greater than	A > B	Swap Operands, Use LT	False	Yes
		Greater-than-or-equal	A ≥ B	Swap Operands, Use LE	False	Yes
UNORD	011B	Unordered	A, B = Unordered		True	No
NEQ	100B	Not-equal	A ≠ B		True	No
NLT	101B	Not-less-than	NOT(A < B)		True	Yes

Table 3-7. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)

Predicate	imm8 Encoding	Description	Relation where: A Is 1st Operand B Is 2nd Operand	Emulation	Result if NaN Operand	QNaN Oper-and Signals Invalid
NLE	110B	Not-less-than-or-equal	NOT(A \leq B)		True	Yes
		Not-greater-than	NOT(A > B)	Swap Operands, Use NLT	True	Yes
		Not-greater-than-or-equal	NOT(A \geq B)	Swap Operands, Use NLE	True	Yes
ORD	111B	Ordered	A, B = Ordered		False	No

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that the processors with "CPUID.1H:ECX.AVX =0" do not implement the greater-than, greater-than-or-equal, not-greater-than, and not-greater-than-or-equal relations. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-8. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-8. Pseudo-Op and CMPPD Implementation

Pseudo-Op	CMPPD Implementation
CMPEQPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 0</i>
CMPLTPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 1</i>
CMPLEPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 2</i>
CMPUNORDPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 3</i>
CMPNEQPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 4</i>
CMPNLTPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 5</i>
CMPNLEPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 6</i>
CMPORDPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 7</i>

The greater-than relations that the processor does not implement, require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

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

Enhanced Comparison Predicate for VEX-Encoded VCMPPD

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Four comparisons are performed with results written to the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-9). Bits 5 through 7 of the immediate are reserved.

Table 3-9. Comparison Predicate for VCMPPD and VCMPPS Instructions

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNaN ¹
			A > B	A < B	A = B	Unordered ¹	
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	False	False	True	False	No
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False	Yes
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False	Yes
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True	No
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True	No
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True	Yes
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True	Yes
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False	No
EQ_UQ	8H	Equal (unordered, non-signaling)	False	False	True	True	No
NGE_US (NGE)	9H	Not-greater-than-or-equal (unordered, signaling)	False	True	False	True	Yes
NGT_US (NGT)	AH	Not-greater-than (unordered, signaling)	False	True	True	True	Yes
FALSE_OQ(FALSE)	BH	False (ordered, non-signaling)	False	False	False	False	No
NEQ_OQ	CH	Not-equal (ordered, non-signaling)	True	True	False	False	No
GE_OS (GE)	DH	Greater-than-or-equal (ordered, signaling)	True	False	True	False	Yes
GT_OS (GT)	EH	Greater-than (ordered, signaling)	True	False	False	False	Yes
TRUE_UQ(TRUE)	FH	True (unordered, non-signaling)	True	True	True	True	No
EQ_OS	10H	Equal (ordered, signaling)	False	False	True	False	Yes
LT_OQ	11H	Less-than (ordered, nonsignaling)	False	True	False	False	No
LE_OQ	12H	Less-than-or-equal (ordered, non-signaling)	False	True	True	False	No
UNORD_S	13H	Unordered (signaling)	False	False	False	True	Yes
NEQ_US	14H	Not-equal (unordered, signaling)	True	True	False	True	Yes
NLT_UQ	15H	Not-less-than (unordered, nonsignaling)	True	False	True	True	No
NLE_UQ	16H	Not-less-than-or-equal (unordered, nonsignaling)	True	False	False	True	No
ORD_S	17H	Ordered (signaling)	True	True	True	False	Yes
EQ_US	18H	Equal (unordered, signaling)	False	False	True	True	Yes

Table 3-9. Comparison Predicate for VCMPPD and VCMPPS Instructions (Contd.)

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A > B	A < B	A = B	Unordered ¹	
NGE_UQ	19H	Not-greater-than-or-equal (unordered, nonsignaling)	False	True	False	True	No
NGT_UQ	1AH	Not-greater-than (unordered, non-signaling)	False	True	True	True	No
FALSE_OS	1BH	False (ordered, signaling)	False	False	False	False	Yes
NEQ_OS	1CH	Not-equal (ordered, signaling)	True	True	False	False	Yes
GE_OQ	1DH	Greater-than-or-equal (ordered, nonsignaling)	True	False	True	False	No
GT_OQ	1EH	Greater-than (ordered, nonsignaling)	True	False	False	False	No
TRUE_US	1FH	True (unordered, signaling)	True	True	True	True	Yes

NOTES:

1. If either operand A or B is a NAN.

Processors with “CPUID.1H:ECX.AVX =1” implement the full complement of 32 predicates shown in Table 3-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 3-10, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 3-10. Pseudo-Op and VCMPPD Implementation

Pseudo-Op	CMPPD Implementation
VCMPEQPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 0
VCMPLTPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 1
VCMPLEPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 2
VCMPUNORDPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 3
VCMPNEQPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 4
VCMPNLTPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 5
VCMPNLEPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 6
VCMPORDPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 7
VCMPEQ_UQPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 8
VCMPNGEPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 9
VCMPNGTPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 0AH
VCMPFALSEPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 0BH
VCMPNEQ_OQPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 0CH
VCMPGEPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 0DH
VCMPGTPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 0EH
VCMPTRUEPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 0FH
VCMPEQ_OSPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 10H
VCMPLT_OQPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 11H
VCMPLE_OQPD reg1, reg2, reg3	VCMPPD reg1, reg2, reg3, 12H

Table 3-10. Pseudo-Op and VCMPPD Implementation

Pseudo-Op	CMPPD Implementation
VCMPUNORD_SPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 13H</i>
VCMPNEQ_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 14H</i>
VCMPNLT_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 15H</i>
VCMPNLE_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 16H</i>
VCMPORD_SPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 17H</i>
VCMPEQ_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 18H</i>
VCMPNGE_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 19H</i>
VCMPNGT_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1AH</i>
VCMPFALSE_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1BH</i>
VCMPNEQ_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1CH</i>
VCMPGE_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1DH</i>
VCMPGT_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1EH</i>
VCMPTURE_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1FH</i>

Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 \leftarrow EQ_OQ; OP5 \leftarrow EQ_OQ;
- 1: OP3 \leftarrow LT_OS; OP5 \leftarrow LT_OS;
- 2: OP3 \leftarrow LE_OS; OP5 \leftarrow LE_OS;
- 3: OP3 \leftarrow UNORD_Q; OP5 \leftarrow UNORD_Q;
- 4: OP3 \leftarrow NEQ_UQ; OP5 \leftarrow NEQ_UQ;
- 5: OP3 \leftarrow NLT_US; OP5 \leftarrow NLT_US;
- 6: OP3 \leftarrow NLE_US; OP5 \leftarrow NLE_US;
- 7: OP3 \leftarrow ORD_Q; OP5 \leftarrow ORD_Q;
- 8: OP5 \leftarrow EQ_UQ;
- 9: OP5 \leftarrow NGE_US;
- 10: OP5 \leftarrow NGT_US;
- 11: OP5 \leftarrow FALSE_OQ;
- 12: OP5 \leftarrow NEQ_OQ;
- 13: OP5 \leftarrow GE_OS;
- 14: OP5 \leftarrow GT_OS;
- 15: OP5 \leftarrow TRUE_UQ;
- 16: OP5 \leftarrow EQ_OS;
- 17: OP5 \leftarrow LT_OQ;
- 18: OP5 \leftarrow LE_OQ;
- 19: OP5 \leftarrow UNORD_S;
- 20: OP5 \leftarrow NEQ_US;
- 21: OP5 \leftarrow NLT_UQ;
- 22: OP5 \leftarrow NLE_UQ;
- 23: OP5 \leftarrow ORD_S;
- 24: OP5 \leftarrow EQ_US;
- 25: OP5 \leftarrow NGE_UQ;
- 26: OP5 \leftarrow NGT_UQ;
- 27: OP5 \leftarrow FALSE_OS;
- 28: OP5 \leftarrow NEQ_OS;
- 29: OP5 \leftarrow GE_OQ;

30: OP5 \leftarrow GT_OQ;
 31: OP5 \leftarrow TRUE_US;
 DEFAULT: Reserved;

CMPPD (128-bit Legacy SSE version)

CMP0 \leftarrow SRC1[63:0] OP3 SRC2[63:0];
 CMP1 \leftarrow SRC1[127:64] OP3 SRC2[127:64];
 IF CMP0 = TRUE
 THEN DEST[63:0] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[63:0] \leftarrow 0000000000000000H; FI;
 IF CMP1 = TRUE
 THEN DEST[127:64] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[127:64] \leftarrow 0000000000000000H; FI;
 DEST[VLMAX-1:128] (Unmodified)

VCMPPD (VEX.128 encoded version)

CMP0 \leftarrow SRC1[63:0] OP5 SRC2[63:0];
 CMP1 \leftarrow SRC1[127:64] OP5 SRC2[127:64];
 IF CMP0 = TRUE
 THEN DEST[63:0] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[63:0] \leftarrow 0000000000000000H; FI;
 IF CMP1 = TRUE
 THEN DEST[127:64] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[127:64] \leftarrow 0000000000000000H; FI;
 DEST[VLMAX-1:128] \leftarrow 0

VCMPPD (VEX.256 encoded version)

CMP0 \leftarrow SRC1[63:0] OP5 SRC2[63:0];
 CMP1 \leftarrow SRC1[127:64] OP5 SRC2[127:64];
 CMP2 \leftarrow SRC1[191:128] OP5 SRC2[191:128];
 CMP3 \leftarrow SRC1[255:192] OP5 SRC2[255:192];
 IF CMP0 = TRUE
 THEN DEST[63:0] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[63:0] \leftarrow 0000000000000000H; FI;
 IF CMP1 = TRUE
 THEN DEST[127:64] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[127:64] \leftarrow 0000000000000000H; FI;
 IF CMP2 = TRUE
 THEN DEST[191:128] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[191:128] \leftarrow 0000000000000000H; FI;
 IF CMP3 = TRUE
 THEN DEST[255:192] \leftarrow FFFFFFFFFFFFFFFFH;
 ELSE DEST[255:192] \leftarrow 0000000000000000H; FI;

Intel C/C++ Compiler Intrinsic Equivalents

CMPPD for equality: $\text{__m128d}_\text{mm_cmpeq_pd}(\text{__m128d } a, \text{__m128d } b)$
 CMPPD for less-than: $\text{__m128d}_\text{mm_cmplt_pd}(\text{__m128d } a, \text{__m128d } b)$
 CMPPD for less-than-or-equal: $\text{__m128d}_\text{mm_cmple_pd}(\text{__m128d } a, \text{__m128d } b)$
 CMPPD for greater-than: $\text{__m128d}_\text{mm_cmpgt_pd}(\text{__m128d } a, \text{__m128d } b)$
 CMPPD for greater-than-or-equal: $\text{__m128d}_\text{mm_cmpge_pd}(\text{__m128d } a, \text{__m128d } b)$
 CMPPD for inequality: $\text{__m128d}_\text{mm_cmpneq_pd}(\text{__m128d } a, \text{__m128d } b)$
 CMPPD for not-less-than: $\text{__m128d}_\text{mm_cmpnlt_pd}(\text{__m128d } a, \text{__m128d } b)$

CMPPD for not-greater-than: `__m128d _mm_cmplt_pd(__m128d a, __m128d b)`
CMPPD for not-greater-than-or-equal: `__m128d _mm_cmpte_pd(__m128d a, __m128d b)`
CMPPD for ordered: `__m128d _mm_cmprord_pd(__m128d a, __m128d b)`
CMPPD for unordered: `__m128d _mm_cmprunord_pd(__m128d a, __m128d b)`
CMPPD for not-less-than-or-equal: `__m128d _mm_cmple_pd(__m128d a, __m128d b)`
VCMPPD: `__m256 _mm256_cmp_pd(__m256 a, __m256 b, const int imm)`
VCMPPD: `__m128 _mm_cmp_pd(__m128 a, __m128 b, const int imm)`

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in above table, Denormal.

Other Exceptions

See Exceptions Type 2.

CMPPS—Compare Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
OF C2 /r ib CMPPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE	Compare packed single-precision floating-point values in <i>xmm2/mem</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.
VEX.NDS.128.0F.WIG C2 /r ib VCMPPS <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Compare packed single-precision floating-point values in <i>xmm3/m128</i> and <i>xmm2</i> using bits 4:0 of <i>imm8</i> as a comparison predicate.
VEX.NDS.256.0F.WIG C2 /r ib VCMPPS <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Compare packed single-precision floating-point values in <i>ymm3/m256</i> and <i>ymm2</i> using bits 4:0 of <i>imm8</i> as a comparison predicate.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a SIMD compare of the packed single-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false). The sign of zero is ignored for comparisons, so that -0.0 is equal to +0.0.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate a fault, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-11. Compiler should treat reserved Imm8 values as illegal syntax.

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

Table 3-11. Pseudo-Ops and CMPPS

Pseudo-Op	Implementation
CMPEQPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 0</i>
CMPLETPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 1</i>
CMPLEPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 2</i>
CMPUNORDPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 3</i>
CMPNEQPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 4</i>
CMPNLTPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 5</i>
CMPNLEPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 6</i>
CMPORDPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 7</i>

The greater-than relations not implemented by processor require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Enhanced Comparison Predicate for VEX-Encoded VCMPPS

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-9). Bits 5 through 7 of the immediate are reserved.

Processors with “CPUID.1H:ECX.AVX =1” implement the full complement of 32 predicates shown in Table 3-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 3-12, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 3-12. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
VCMPEQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0</i>
VCMPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1</i>
VCMPLEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 2</i>
VCMPPUNORDPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 3</i>
VCMPPNEQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 4</i>
VCMPPNLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 5</i>
VCMPPNLEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 6</i>
VCMPPORDPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 8</i>
VCMPPNGEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 9</i>
VCMPPNGTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, OAH</i>
VCMPPFALSEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, OBH</i>

Table 3-12. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
VCMPNEQ_OQPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 0CH
VCMPGEPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 0DH
VCMPGTPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 0EH
VCMPTRUEPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 0FH
VCMPEQ_OSPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 10H
VCMPLT_OQPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 11H
VCMPLE_OQPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 12H
VCMPUNORD_SPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 13H
VCMPNEQ_USPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 14H
VCMPNLT_UQPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 15H
VCMPNLE_UQPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 16H
VCMPORD_SPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 17H
VCMPEQ_USPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 18H
VCMPNGE_UQPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 19H
VCMPNGT_UQPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 1AH
VCMPFALSE_OSPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 1BH
VCMPNEQ_OSPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 1CH
VCMPGE_OQPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 1DH
VCMPGT_OQPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 1EH
VCMPTRUE_USPS reg1, reg2, reg3	VCMPPS reg1, reg2, reg3, 1FH

Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 \leftarrow EQ_OQ; OP5 \leftarrow EQ_OQ;
- 1: OP3 \leftarrow LT_OS; OP5 \leftarrow LT_OS;
- 2: OP3 \leftarrow LE_OS; OP5 \leftarrow LE_OS;
- 3: OP3 \leftarrow UNORD_Q; OP5 \leftarrow UNORD_Q;
- 4: OP3 \leftarrow NEQ_UQ; OP5 \leftarrow NEQ_UQ;
- 5: OP3 \leftarrow NLT_US; OP5 \leftarrow NLT_US;
- 6: OP3 \leftarrow NLE_US; OP5 \leftarrow NLE_US;
- 7: OP3 \leftarrow ORD_Q; OP5 \leftarrow ORD_Q;
- 8: OP5 \leftarrow EQ_UQ;
- 9: OP5 \leftarrow NGE_US;
- 10: OP5 \leftarrow NGT_US;
- 11: OP5 \leftarrow FALSE_OQ;
- 12: OP5 \leftarrow NEQ_OQ;
- 13: OP5 \leftarrow GE_OS;
- 14: OP5 \leftarrow GT_OS;
- 15: OP5 \leftarrow TRUE_UQ;
- 16: OP5 \leftarrow EQ_OS;
- 17: OP5 \leftarrow LT_OQ;
- 18: OP5 \leftarrow LE_OQ;
- 19: OP5 \leftarrow UNORD_S;
- 20: OP5 \leftarrow NEQ_US;

```

21: OP5 ← NLT_UQ;
22: OP5 ← NLE_UQ;
23: OP5 ← ORD_S;
24: OP5 ← EQ_US;
25: OP5 ← NGE_UQ;
26: OP5 ← NGT_UQ;
27: OP5 ← FALSE_OS;
28: OP5 ← NEQ_OS;
29: OP5 ← GE_OQ;
30: OP5 ← GT_OQ;
31: OP5 ← TRUE_US;
DEFAULT: Reserved
EASC;

```

CMPPS (128-bit Legacy SSE version)

```

CMP0 ← SRC1[31:0] OP3 SRC2[31:0];
CMP1 ← SRC1[63:32] OP3 SRC2[63:32];
CMP2 ← SRC1[95:64] OP3 SRC2[95:64];
CMP3 ← SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
  THEN DEST[31:0] ← FFFFFFFFH;
  ELSE DEST[31:0] ← 00000000H; Fi;
IF CMP1 = TRUE
  THEN DEST[63:32] ← FFFFFFFFH;
  ELSE DEST[63:32] ← 00000000H; Fi;
IF CMP2 = TRUE
  THEN DEST[95:64] ← FFFFFFFFH;
  ELSE DEST[95:64] ← 00000000H; Fi;
IF CMP3 = TRUE
  THEN DEST[127:96] ← FFFFFFFFH;
  ELSE DEST[127:96] ← 00000000H; Fi;
DEST[VLMAX-1:128] (Unmodified)

```

VCMPPS (VEX.128 encoded version)

```

CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
IF CMP0 = TRUE
  THEN DEST[31:0] ← FFFFFFFFH;
  ELSE DEST[31:0] ← 00000000H; Fi;
IF CMP1 = TRUE
  THEN DEST[63:32] ← FFFFFFFFH;
  ELSE DEST[63:32] ← 00000000H; Fi;
IF CMP2 = TRUE
  THEN DEST[95:64] ← FFFFFFFFH;
  ELSE DEST[95:64] ← 00000000H; Fi;
IF CMP3 = TRUE
  THEN DEST[127:96] ← FFFFFFFFH;
  ELSE DEST[127:96] ← 00000000H; Fi;
DEST[VLMAX-1:128] ← 0

```

VCMPPS (VEX.256 encoded version)

```

CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
CMP4 ← SRC1[159:128] OP5 SRC2[159:128];
CMP5 ← SRC1[191:160] OP5 SRC2[191:160];
CMP6 ← SRC1[223:192] OP5 SRC2[223:192];
CMP7 ← SRC1[255:224] OP5 SRC2[255:224];
IF CMP0 = TRUE
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 00000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 00000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 00000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 00000000H; FI;
IF CMP4 = TRUE
    THEN DEST[159:128] ← FFFFFFFFH;
    ELSE DEST[159:128] ← 00000000H; FI;
IF CMP5 = TRUE
    THEN DEST[191:160] ← FFFFFFFFH;
    ELSE DEST[191:160] ← 00000000H; FI;
IF CMP6 = TRUE
    THEN DEST[223:192] ← FFFFFFFFH;
    ELSE DEST[223:192] ← 00000000H; FI;
IF CMP7 = TRUE
    THEN DEST[255:224] ← FFFFFFFFH;
    ELSE DEST[255:224] ← 00000000H; FI;

```

Intel C/C++ Compiler Intrinsic Equivalents

CMPSS for equality: `__m128 _mm_cmpeq_ps(__m128 a, __m128 b)`
 CMPSS for less-than: `__m128 _mm_cmplt_ps(__m128 a, __m128 b)`
 CMPSS for less-than-or-equal: `__m128 _mm_cmple_ps(__m128 a, __m128 b)`
 CMPSS for greater-than: `__m128 _mm_cmpgt_ps(__m128 a, __m128 b)`
 CMPSS for greater-than-or-equal: `__m128 _mm_cmpge_ps(__m128 a, __m128 b)`
 CMPSS for inequality: `__m128 _mm_cmpneq_ps(__m128 a, __m128 b)`
 CMPSS for not-less-than: `__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)`
 CMPSS for not-greater-than: `__m128 _mm_cmpngt_ps(__m128 a, __m128 b)`
 CMPSS for not-greater-than-or-equal: `__m128 _mm_cmpnge_ps(__m128 a, __m128 b)`
 CMPSS for ordered: `__m128 _mm_cmpord_ps(__m128 a, __m128 b)`
 CMPSS for unordered: `__m128 _mm_cmpunord_ps(__m128 a, __m128 b)`
 CMPSS for not-less-than-or-equal: `__m128 _mm_cmplne_ps(__m128 a, __m128 b)`
 VCMPPS: `__m256 _mm256_cmp_ps(__m256 a, __m256 b, const int imm)`
 VCMPPS: `__m128 _mm_cmp_ps(__m128 a, __m128 b, const int imm)`

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in above table, Denormal.

Other Exceptions

See Exceptions Type 2.

CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A6	CMPS <i>m8, m8</i>	NP	Valid	Valid	For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R E)SI to byte at address (R E)DI. The status flags are set accordingly.
A7	CMPS <i>m16, m16</i>	NP	Valid	Valid	For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R E)SI with word at address (R E)DI. The status flags are set accordingly.
A7	CMPS <i>m32, m32</i>	NP	Valid	Valid	For legacy mode, compare dword at address DS:(E)SI at dword at address ES:(E)DI; For 64-bit mode compare dword at address (R E)SI at dword at address (R E)DI. The status flags are set accordingly.
REX.W + A7	CMPS <i>m64, m64</i>	NP	Valid	N.E.	Compares quadword at address (R E)SI with quadword at address (R E)DI and sets the status flags accordingly.
A6	CMPSB	NP	Valid	Valid	For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R E)SI with byte at address (R E)DI. The status flags are set accordingly.
A7	CMPSW	NP	Valid	Valid	For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R E)SI with word at address (R E)DI. The status flags are set accordingly.
A7	CMPSD	NP	Valid	Valid	For legacy mode, compare dword at address DS:(E)SI with dword at address ES:(E)DI; For 64-bit mode compare dword at address (R E)SI with dword at address (R E)DI. The status flags are set accordingly.
REX.W + A7	CMPSQ	NP	Valid	N.E.	Compares quadword at address (R E)SI with quadword at address (R E)DI and sets the status flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Compares the byte, word, doubleword, or quadword specified with the first source operand with the byte, word, doubleword, or quadword specified with the second source operand and sets the status flags in the EFLAGS register according to the results.

Both source operands are located in memory. The address of the first source operand is read from DS:SI, DS:ESI or RSI (depending on the address-size attribute of the instruction is 16, 32, or 64, respectively). The address of the second source operand is read from ES:DI, ES:EDI or RDI (again depending on the address-size attribute of the

instruction is 16, 32, or 64). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

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 CMPS mnemonic) allows the two source operands to be specified explicitly. Here, the source operands should be symbols that indicate the size and location of the source values. This explicit-operand form is provided to allow documentation. However, note that the documentation provided by this form can be misleading. That is, the source operand symbols must specify the correct type (size) of the operands (bytes, words, or doublewords, quadwords), but they do not have to specify the correct location. Locations of the source operands are always specified by the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the CMPS instructions. Here also the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers are assumed by the processor to specify the location of the source operands. The size of the source operands is selected with the mnemonic: CMPSB (byte comparison), CMPSW (word comparison), CMPSD (doubleword comparison), or CMPSQ (quadword comparison using REX.W).

After the comparison, the (E|R)SI and (E|R)DI registers increment or decrement automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E|R)SI and (E|R)DI register increment; if the DF flag is 1, the registers decrement.) The registers increment or decrement by 1 for byte operations, by 2 for word operations, 4 for doubleword operations. If operand size is 64, RSI and RDI registers increment by 8 for quadword operations.

The CMPS, CMPSB, CMPSW, CMPSD, and CMPSQ instructions can be preceded by the REP prefix for block comparisons. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, 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. Use of the REX.W prefix promotes doubleword operation to 64 bits (see CMPSQ). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```

temp ← SRC1 - SRC2;
SetStatusFlags(temp);

IF (64-Bit Mode)
  THEN
    IF (Byte comparison)
      THEN IF DF = 0
        THEN
          (R|E)SI ← (R|E)SI + 1;
          (R|E)DI ← (R|E)DI + 1;
        ELSE
          (R|E)SI ← (R|E)SI - 1;
          (R|E)DI ← (R|E)DI - 1;
        FI;
      ELSE IF (Word comparison)
        THEN IF DF = 0
          THEN
            (R|E)SI ← (R|E)SI + 2;
            (R|E)DI ← (R|E)DI + 2;
          ELSE
            (R|E)SI ← (R|E)SI - 2;
            (R|E)DI ← (R|E)DI - 2;
          FI;
        ELSE IF (Doubleword comparison)

```

```

THEN IF DF = 0
    THEN
        (R|E)SI ← (R|E)SI + 4;
        (R|E)DI ← (R|E)DI + 4;
    ELSE
        (R|E)SI ← (R|E)SI - 4;
        (R|E)DI ← (R|E)DI - 4;
    Fl;
ELSE (* Quadword comparison *)
    THEN IF DF = 0
        (R|E)SI ← (R|E)SI + 8;
        (R|E)DI ← (R|E)DI + 8;
    ELSE
        (R|E)SI ← (R|E)SI - 8;
        (R|E)DI ← (R|E)DI - 8;
    Fl;
Fl;
ELSE (* Non-64-bit Mode *)
    IF (byte comparison)
        THEN IF DF = 0
            THEN
                (E)SI ← (E)SI + 1;
                (E)DI ← (E)DI + 1;
            ELSE
                (E)SI ← (E)SI - 1;
                (E)DI ← (E)DI - 1;
            Fl;
        ELSE IF (Word comparison)
            THEN IF DF = 0
                (E)SI ← (E)SI + 2;
                (E)DI ← (E)DI + 2;
            ELSE
                (E)SI ← (E)SI - 2;
                (E)DI ← (E)DI - 2;
            Fl;
        ELSE (* Doubleword comparison *)
            THEN IF DF = 0
                (E)SI ← (E)SI + 4;
                (E)DI ← (E)DI + 4;
            ELSE
                (E)SI ← (E)SI - 4;
                (E)DI ← (E)DI - 4;
            Fl;
        Fl;
    Fl;

```

Flags Affected

The CF, OF, SF, ZF, AF, and PF 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 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

CMPSD—Compare Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF C2 /r ib CMPSD <i>xmm1, xmm2/m64, imm8</i>	RMI	V/V	SSE2	Compare low double-precision floating-point value in <i>xmm2/m64</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.
VEX.NDS.LIG.F2.0F.WIG C2 /r ib VCMPSD <i>xmm1, xmm2, xmm3/m64, imm8</i>	RVMI	V/V	AVX	Compare low double precision floating-point value in <i>xmm3/m64</i> and <i>xmm2</i> using bits 4:0 of <i>imm8</i> as comparison predicate.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compares the low double-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed. The comparison result is a quad-word mask of all 1s (comparison true) or all 0s (comparison false). The sign of zero is ignored for comparisons, so that -0.0 is equal to +0.0.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate a fault, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination operand), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-13. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-13. Pseudo-Ops and CMPSD

Pseudo-Op	Implementation
CMPEQSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 0</i>
CMPLTSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 1</i>
CMPLES SD <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 2</i>
CMPUNORD SD <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 3</i>
CMPNEQSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 4</i>
CMPNLTS D <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 5</i>
CMPNLES D <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 6</i>
CMPORD SD <i>xmm1, xmm2</i>	CMPSD <i>xmm1,xmm2, 7</i>

The greater-than relations not implemented in the processor require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

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

Enhanced Comparison Predicate for VEX-Encoded VCMPSD

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-9). Bits 5 through 7 of the immediate are reserved.

Processors with “CPUID.1H:ECX.AVX =1” implement the full complement of 32 predicates shown in Table 3-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-14, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 3-14. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPEQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0</i>
VCMPLTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1</i>
VCMPLESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 2</i>
VCMPUNORDSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 3</i>
VCMPNEQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 4</i>
VCMPNLTS D <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 5</i>
VCMPNLES D <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 6</i>
VCMPORD SD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 8</i>
VCMPNGESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 9</i>
VCMPNGTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0AH</i>
VCMPFALSESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0BH</i>
VCMPNEQ_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0CH</i>
VCMPGESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0DH</i>
VCMPGTS D <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0EH</i>

Table 3-14. Pseudo-Op and VCMPSD Implementation (Contd.)

Pseudo-Op	CMPSD Implementation
VCMPTRUESD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 0FH
VCMPEQ_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 10H
VCMPLT_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 11H
VCMPLE_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 12H
VCMPUNORD_SSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 13H
VCMPNEQ_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 14H
VCMPNLT_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 15H
VCMPNLE_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 16H
VCMPORD_SSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 17H
VCMPEQ_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 18H
VCMPNGE_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 19H
VCMPNGT_UQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1AH
VCMPFALSE_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1BH
VCMPNEQ_OSSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1CH
VCMPGE_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1DH
VCMPGT_OQSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1EH
VCMPTURE_USSD reg1, reg2, reg3	VCMPSD reg1, reg2, reg3, 1FH

Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 \leftarrow EQ_OQ; OP5 \leftarrow EQ_OQ;
- 1: OP3 \leftarrow LT_OS; OP5 \leftarrow LT_OS;
- 2: OP3 \leftarrow LE_OS; OP5 \leftarrow LE_OS;
- 3: OP3 \leftarrow UNORD_Q; OP5 \leftarrow UNORD_Q;
- 4: OP3 \leftarrow NEQ_UQ; OP5 \leftarrow NEQ_UQ;
- 5: OP3 \leftarrow NLT_US; OP5 \leftarrow NLT_US;
- 6: OP3 \leftarrow NLE_US; OP5 \leftarrow NLE_US;
- 7: OP3 \leftarrow ORD_Q; OP5 \leftarrow ORD_Q;
- 8: OP5 \leftarrow EQ_UQ;
- 9: OP5 \leftarrow NGE_US;
- 10: OP5 \leftarrow NGT_US;
- 11: OP5 \leftarrow FALSE_OQ;
- 12: OP5 \leftarrow NEQ_OQ;
- 13: OP5 \leftarrow GE_OS;
- 14: OP5 \leftarrow GT_OS;
- 15: OP5 \leftarrow TRUE_UQ;
- 16: OP5 \leftarrow EQ_OS;
- 17: OP5 \leftarrow LT_OQ;
- 18: OP5 \leftarrow LE_OQ;
- 19: OP5 \leftarrow UNORD_S;
- 20: OP5 \leftarrow NEQ_US;
- 21: OP5 \leftarrow NLT_UQ;
- 22: OP5 \leftarrow NLE_UQ;
- 23: OP5 \leftarrow ORD_S;
- 24: OP5 \leftarrow EQ_US;

```

25: OP5 ← NGE_UQ;
26: OP5 ← NGT_UQ;
27: OP5 ← FALSE_OS;
28: OP5 ← NEQ_OS;
29: OP5 ← GE_OQ;
30: OP5 ← GT_OQ;
31: OP5 ← TRUE_US;
DEFAULT: Reserved
ESAC;

```

CMPSD (128-bit Legacy SSE version)

```

CMPO ← DEST[63:0] OP3 SRC[63:0];
IF CMPO = TRUE
THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ← 0000000000000000H; FI;
DEST[VLMAX-1:64] (Unmodified)

```

VCMPSD (VEX.128 encoded version)

```

CMPO ← SRC1[63:0] OP5 SRC2[63:0];
IF CMPO = TRUE
THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ← 0000000000000000H; FI;
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

```

Intel C/C++ Compiler Intrinsic Equivalents

CMPSD for equality: `__m128d _mm_cmpeq_sd(__m128d a, __m128d b)`
 CMPSD for less-than: `__m128d _mm_cmplt_sd(__m128d a, __m128d b)`
 CMPSD for less-than-or-equal: `__m128d _mm_cmple_sd(__m128d a, __m128d b)`
 CMPSD for greater-than: `__m128d _mm_cmpgt_sd(__m128d a, __m128d b)`
 CMPSD for greater-than-or-equal: `__m128d _mm_cmpte_sd(__m128d a, __m128d b)`
 CMPSD for inequality: `__m128d _mm_cmpneq_sd(__m128d a, __m128d b)`
 CMPSD for not-less-than: `__m128d _mm_cmplt_sd(__m128d a, __m128d b)`
 CMPSD for not-greater-than: `__m128d _mm_cmptgt_sd(__m128d a, __m128d b)`
 CMPSD for not-greater-than-or-equal: `__m128d _mm_cmpte_sd(__m128d a, __m128d b)`
 CMPSD for ordered: `__m128d _mm_cmppord_sd(__m128d a, __m128d b)`
 CMPSD for unordered: `__m128d _mm_cmppunord_sd(__m128d a, __m128d b)`
 CMPSD for not-less-than-or-equal: `__m128d _mm_cmppnle_sd(__m128d a, __m128d b)`
 VCMPSD: `__m128 _mm_cmpp_sd(__m128 a, __m128 b, const int imm)`

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in above table, Denormal.

Other Exceptions

See Exceptions Type 3.

CMPSS—Compare Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF C2 /r ib CMPSS <i>xmm1, xmm2/m32, imm8</i>	RMI	V/V	SSE	Compare low single-precision floating-point value in <i>xmm2/m32</i> and <i>xmm1</i> using <i>imm8</i> as comparison predicate.
VEX.NDS.LIG.F3.0F.WIG C2 /r ib VCMPSS <i>xmm1, xmm2, xmm3/m32, imm8</i>	RVMI	V/V	AVX	Compare low single precision floating-point value in <i>xmm3/m32</i> and <i>xmm2</i> using bits 4:0 of <i>imm8</i> as comparison predicate.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Compares the low single-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed. The comparison result is a double-word mask of all 1s (comparison true) or all 0s (comparison false). The sign of zero is ignored for comparisons, so that -0.0 is equal to +0.0.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate a fault, since a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination operand), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-15. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-15. Pseudo-Ops and CMPSS

Pseudo-Op	CMPSS Implementation
CMPEQSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 0</i>
CMLTSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 1</i>
CMPLESS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 2</i>
CMPUNORDSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 3</i>
CMPNEQSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 4</i>
CMPNLTSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 5</i>
CMPNLESS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 6</i>
CMPORDSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 7</i>

The greater-than relations not implemented in the processor require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

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

Enhanced Comparison Predicate for VEX-Encoded VCMPSD

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. Bits (VLMAX-1:128) of the destination YMM register are zeroed. The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-9). Bits 5 through 7 of the immediate are reserved.

Processors with “CPUID.1H:ECX.AVX =1” implement the full complement of 32 predicates shown in Table 3-9, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-16, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface.

Table 3-16. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSS Implementation
VCMPEQSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0</i>
VCMPLTSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1</i>
VCMPLSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 2</i>
VCMPPUNORDSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 3</i>
VCMPPNEQSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 4</i>
VCMPPNLTSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 5</i>
VCMPPNLESS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 6</i>
VCMPPORDSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 8</i>
VCMPPNGESS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 9</i>
VCMPPNGTSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0AH</i>
VCMPPFALSESS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0BH</i>
VCMPPNEQ_OQSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0CH</i>
VCMPPGESS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0DH</i>
VCMPPGTSS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0EH</i>

Table 3-16. Pseudo-Op and VCMPSS Implementation (Contd.)

Pseudo-Op	CMPSS Implementation
VCMPTRUESS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 0FH
VCMPEQ_OSSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 10H
VCMPLT_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 11H
VCMPLE_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 12H
VCMPUNORD_SSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 13H
VCMPNEQ_USSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 14H
VCMPNLT_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 15H
VCMPNLE_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 16H
VCMPORD_SSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 17H
VCMPEQ_USSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 18H
VCMPNGE_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 19H
VCMPNGT_UQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1AH
VCMPFALSE_OSSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1BH
VCMPNEQ_OSSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1CH
VCMPGE_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1DH
VCMPGT_OQSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1EH
VCMPTRUE_USSS reg1, reg2, reg3	VCMPSS reg1, reg2, reg3, 1FH

Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 \leftarrow EQ_OQ; OP5 \leftarrow EQ_OQ;
- 1: OP3 \leftarrow LT_OS; OP5 \leftarrow LT_OS;
- 2: OP3 \leftarrow LE_OS; OP5 \leftarrow LE_OS;
- 3: OP3 \leftarrow UNORD_Q; OP5 \leftarrow UNORD_Q;
- 4: OP3 \leftarrow NEQ_UQ; OP5 \leftarrow NEQ_UQ;
- 5: OP3 \leftarrow NLT_US; OP5 \leftarrow NLT_US;
- 6: OP3 \leftarrow NLE_US; OP5 \leftarrow NLE_US;
- 7: OP3 \leftarrow ORD_Q; OP5 \leftarrow ORD_Q;
- 8: OP5 \leftarrow EQ_UQ;
- 9: OP5 \leftarrow NGE_US;
- 10: OP5 \leftarrow NGT_US;
- 11: OP5 \leftarrow FALSE_OQ;
- 12: OP5 \leftarrow NEQ_OQ;
- 13: OP5 \leftarrow GE_OS;
- 14: OP5 \leftarrow GT_OS;
- 15: OP5 \leftarrow TRUE_UQ;
- 16: OP5 \leftarrow EQ_OS;
- 17: OP5 \leftarrow LT_OQ;
- 18: OP5 \leftarrow LE_OQ;
- 19: OP5 \leftarrow UNORD_S;
- 20: OP5 \leftarrow NEQ_US;
- 21: OP5 \leftarrow NLT_UQ;
- 22: OP5 \leftarrow NLE_UQ;
- 23: OP5 \leftarrow ORD_S;
- 24: OP5 \leftarrow EQ_US;

```

25: OP5 ← NGE_UQ;
26: OP5 ← NGT_UQ;
27: OP5 ← FALSE_OS;
28: OP5 ← NEQ_OS;
29: OP5 ← GE_OQ;
30: OP5 ← GT_OQ;
31: OP5 ← TRUE_US;
DEFAULT: Reserved
ESAC;

```

CMPSS (128-bit Legacy SSE version)

```

CMPO ← DEST[31:0] OP3 SRC[31:0];
IF CMPO = TRUE
THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 00000000H; FI;
DEST[VLMAX-1:32] (Unmodified)

```

VCMPPS (VEX.128 encoded version)

```

CMPO ← SRC1[31:0] OP5 SRC2[31:0];
IF CMPO = TRUE
THEN DEST[31:0] ← FFFFFFFFH;
ELSE DEST[31:0] ← 00000000H; FI;
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0

```

Intel C/C++ Compiler Intrinsic Equivalents

CMPSS for equality: `__m128_mm_cmpeq_ss(__m128 a, __m128 b)`
 CMPSS for less-than: `__m128_mm_cmplt_ss(__m128 a, __m128 b)`
 CMPSS for less-than-or-equal: `__m128_mm_cmple_ss(__m128 a, __m128 b)`
 CMPSS for greater-than: `__m128_mm_cmpgt_ss(__m128 a, __m128 b)`
 CMPSS for greater-than-or-equal: `__m128_mm_cmpge_ss(__m128 a, __m128 b)`
 CMPSS for inequality: `__m128_mm_cmpeq_ss(__m128 a, __m128 b)`
 CMPSS for not-less-than: `__m128_mm_cmpnlt_ss(__m128 a, __m128 b)`
 CMPSS for not-greater-than: `__m128_mm_cmppgt_ss(__m128 a, __m128 b)`
 CMPSS for not-greater-than-or-equal: `__m128_mm_cmppnge_ss(__m128 a, __m128 b)`
 CMPSS for ordered: `__m128_mm_cmpord_ss(__m128 a, __m128 b)`
 CMPSS for unordered: `__m128_mm_cmpunord_ss(__m128 a, __m128 b)`
 CMPSS for not-less-than-or-equal: `__m128_mm_cmpple_ss(__m128 a, __m128 b)`
 VCMPPS: `__m128_mm_cmp_ss(__m128 a, __m128 b, const int imm)`

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in above table, Denormal.

Other Exceptions

See Exceptions Type 3.

CMPXCHG—Compare and Exchange

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF B0/r CMPXCHG <i>r/m8, r8</i>	MR	Valid	Valid*	Compare AL with <i>r/m8</i> . If equal, ZF is set and <i>r8</i> is loaded into <i>r/m8</i> . Else, clear ZF and load <i>r/m8</i> into AL.
REX + OF B0/r CMPXCHG <i>r/m8**,r8</i>	MR	Valid	N.E.	Compare AL with <i>r/m8</i> . If equal, ZF is set and <i>r8</i> is loaded into <i>r/m8</i> . Else, clear ZF and load <i>r/m8</i> into AL.
OF B1/r CMPXCHG <i>r/m16, r16</i>	MR	Valid	Valid*	Compare AX with <i>r/m16</i> . If equal, ZF is set and <i>r16</i> is loaded into <i>r/m16</i> . Else, clear ZF and load <i>r/m16</i> into AX.
OF B1/r CMPXCHG <i>r/m32, r32</i>	MR	Valid	Valid*	Compare EAX with <i>r/m32</i> . If equal, ZF is set and <i>r32</i> is loaded into <i>r/m32</i> . Else, clear ZF and load <i>r/m32</i> into EAX.
REX.W + OF B1/r CMPXCHG <i>r/m64, r64</i>	MR	Valid	N.E.	Compare RAX with <i>r/m64</i> . If equal, ZF is set and <i>r64</i> is loaded into <i>r/m64</i> . Else, clear ZF and load <i>r/m64</i> into RAX.

NOTES:

* See the IA-32 Architecture Compatibility section below.

** 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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA

Description

Compares the value in the AL, AX, EAX, or RAX register with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, EAX or RAX register. RAX register is available only in 64-bit mode.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

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

IA-32 Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Intel486 processors.

Operation

```
(* Accumulator = AL, AX, EAX, or RAX depending on whether a byte, word, doubleword, or quadword comparison is being performed *)
TEMP ← DEST
IF accumulator = TEMP
  THEN
    ZF ← 1;
    DEST ← SRC;
  ELSE
```

```

ZF ← 0;
accumulator ← TEMP;
DEST ← TEMP;
Fl;

```

Flags Affected

The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF C7 /1 m64	M	Valid	Valid*	Compare EDX:EAX with m64. If equal, set ZF and load ECX:EBX into m64. Else, clear ZF and load m64 into EDX:EAX.
CMPXCHG8B m64				
REX.W + OF C7 /1 m128	M	Valid	N.E.	Compare RDX:RAX with m128. If equal, set ZF and load RCX:RBX into m128. Else, clear ZF and load m128 into RDX:RAX.
CMPXCHG16B m128				

NOTES:

*See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA

Description

Compares the 64-bit value in EDX:EAX (or 128-bit value in RDX:RAX if operand size is 128 bits) with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX (or 128-bit value in RCX:RBX) is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX (or RDX:RAX). The destination operand is an 8-byte memory location (or 16-byte memory location if operand size is 128 bits). For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value. For the RDX:RAX and RCX:RBX register pairs, RDX and RCX contain the high-order 64 bits and RAX and RBX contain the low-order 64bits of a 128-bit value.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

In 64-bit mode, default operation size is 64 bits. Use of the REX.W prefix promotes operation to 128 bits. Note that CMPXCHG16B requires that the destination (memory) operand be 16-byte aligned. See the summary chart at the beginning of this section for encoding data and limits. For information on the CPUID flag that indicates CMPXCHG16B, see page 3-170.

IA-32 Architecture Compatibility

This instruction encoding is not supported on Intel processors earlier than the Pentium processors.

Operation

IF (64-Bit Mode and OperandSize = 64)

THEN

 TEMP128 ← DEST

 IF (RDX:RAX = TEMP128)

 THEN

 ZF ← 1;

 DEST ← RCX:RBX;

 ELSE

 ZF ← 0;

 RDX:RAX ← TEMP128;

 DEST ← TEMP128;

 FI;

 FI

```

ELSE
    TEMP64 ← DEST;
    IF (EDX:EAX = TEMP64)
        THEN
            ZF ← 1;
            DEST ← ECX:EBX;
    ELSE
        ZF ← 0;
        EDX:EAX ← TEMP64;
        DEST ← TEMP64;
    FI;
FI;

```

Flags Affected

The ZF flag is set if the destination operand and EDX:EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are unaffected.

Protected Mode Exceptions

#UD	If the destination is not a memory operand.
#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

#UD	If the destination operand is not a memory location.
#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 not a memory location.
#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.
	If memory operand for CMPXCHG16B is not aligned on a 16-byte boundary.
	If CPUID.01H:ECX.CMPXCHG16B[bit 13] = 0.
#UD	If the destination operand is not a memory location.

INSTRUCTION SET REFERENCE, A-M

- #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.

COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 2F /r COMISD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Compare low double-precision floating-point values in <i>xmm1</i> and <i>xmm2/mem64</i> and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2F /r VCOMISD <i>xmm1, xmm2/m64</i>		RM V/V	AVX	Compare low double precision floating-point values in <i>xmm1</i> and <i>xmm2/mem64</i> and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Compares the double-precision floating-point values in the low quadwords of operand 1 (first operand) and 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). The sign of zero is ignored for comparisons, so that -0.0 is equal to +0.0.

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location.

The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

```
RESULT ← OrderedCompare(DEST[63:0] <> SRC[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 Equivalents

```
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)
```

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 2F /r COMISS <i>xmm1</i> , <i>xmm2/m32</i>		RM V/V	SSE	Compare low single-precision floating-point values in <i>xmm1</i> and <i>xmm2/mem32</i> and set the EFLAGS flags accordingly.
VEX.LIG.OF.WIG 2F /r VCOMISS <i>xmm1</i> , <i>xmm2/m32</i>		RM V/V	AVX	Compare low single precision floating-point values in <i>xmm1</i> and <i>xmm2/mem32</i> and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Compares the single-precision floating-point values in the low doublewords of operand 1 (first operand) and 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). The sign of zero is ignored for comparisons, so that -0.0 is equal to +0.0.

Operand 1 is an XMM register; Operand 2 can be an XMM register or a 32 bit memory location.

The COMISS instruction differs from the UCOMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISS instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

```
RESULT ← OrderedCompare(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 Equivalents

```
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)
```

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

CPUID—CPU Identification

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF A2	CPUID	NP	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.¹ The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 3-17 shows information returned, depending on the initial value loaded into the EAX register. Table 3-18 shows the maximum CPUID input value recognized for each family of IA-32 processors on which CPUID is implemented.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using the Intel Core i7 processor, the following is true:

```
CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
CPUID.EAX = 0BH (* Returns Extended Topology Enumeration leaf. *)
CPUID.EAX = 0CH (* INVALID: Returns the same information as CPUID.EAX = 0BH. *)
CPUID.EAX = 80000008H (* Returns linear/physical address size data. *)
CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0BH. *)
```

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers. For example, using the Intel Core i7 processor, the following is true:

```
CPUID.EAX = 07H (*Returns EAX=EBX=ECX=EDX=0. *)
```

When CPUID returns the highest basic leaf information as a result of an invalid input EAX value, any dependence on input ECX value in the basic leaf is honored.

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

See also:

"Serializing Instructions" in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.

“Caching Translation Information” in Chapter 4, “Paging,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Table 3-17. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor					
<i>Basic CPUID Information</i>						
0H	EAX	Maximum Input Value for Basic CPUID Information (see Table 3-18)				
	EBX	“Genu”				
	ECX	“ntel”				
	EDX	“inel”				
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-5)				
	EBX	Bits 07-00: Brand Index Bits 15-08: CLFLUSH line size (Value * 8 = cache line size in bytes) Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID				
	ECX	Feature Information (see Figure 3-6 and Table 3-20)				
	EDX	Feature Information (see Figure 3-7 and Table 3-21)				
	NOTES:					
	* The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1.					
02H	EAX	Cache and TLB Information (see Table 3-22)				
	EBX	Cache and TLB Information				
	ECX	Cache and TLB Information				
	EDX	Cache and TLB Information				
03H	EAX	Reserved.				
	EBX	Reserved.				
	ECX	Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)				
	EDX	Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)				
	NOTES:					
	Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.					
	See AP-485, <i>Intel Processor Identification and the CPUID Instruction</i> (Order Number 241618) for more information on PSN.					
CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLE.BOOT_NT4[bit 22] = 0 (default).						
<i>Deterministic Cache Parameters Leaf</i>						
04H	NOTES:					
	Leaf 04H output depends on the initial value in ECX.*					
	See also: “INPUT EAX = 4: Returns Deterministic Cache Parameters for each level on page 3-178.					
	EAX	Bits 04-00: Cache Type Field 0 = Null - No more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved				

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	<p>Bits 07-05: Cache Level (starts at 1) Bit 08: Self Initializing cache level (does not need SW initialization) Bit 09: Fully Associative cache Bits 13-10: Reserved Bits 25-14: Maximum number of addressable IDs for logical processors sharing this cache**, *** Bits 31-26: Maximum number of addressable IDs for processor cores in the physical package**, ****, *****</p> <p>EBX Bits 11-00: L = System Coherency Line Size** Bits 21-12: P = Physical Line partitions** Bits 31-22: W = Ways of associativity**</p> <p>ECX Bits 31-00: S = Number of Sets**</p> <p>EDX Bit 0: Write-Back Invalidate/Invalidate 0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache. 1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache. Bit 1: Cache Inclusiveness 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bit 2: Complex Cache Indexing 0 = Direct mapped cache. 1 = A complex function is used to index the cache, potentially using all address bits. Bits 31-03: Reserved = 0</p> <p>NOTES:</p> <ul style="list-style-type: none"> * If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Invalid sub-leaves of EAX = 04H: ECX = n, n > 3. ** Add one to the return value to get the result. ***The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache **** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID. ***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0. 	
<i>MONITOR/MwAIT Leaf</i>		
05H	EAX	Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	EBX	Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	ECX	Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported Bit 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled Bits 31 - 02: Reserved

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 03 - 00: Number of C0* sub C-states supported using MWAIT Bits 07 - 04: Number of C1* sub C-states supported using MWAIT Bits 11 - 08: Number of C2* sub C-states supported using MWAIT Bits 15 - 12: Number of C3* sub C-states supported using MWAIT Bits 19 - 16: Number of C4* sub C-states supported using MWAIT Bits 23 - 20: Number of C5* sub C-states supported using MWAIT Bits 27 - 24: Number of C6* sub C-states supported using MWAIT Bits 31 - 28: Number of C7* sub C-states supported using MWAIT NOTE: * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.
<i>Thermal and Power Management Leaf</i>		
06H	EAX	Bit 00: Digital temperature sensor is supported if set Bit 01: Intel Turbo Boost Technology Available (see description of IA32_MISC_ENABLE[38]). Bit 02: ARAT. APIC-Timer-always-running feature is supported if set. Bit 03: Reserved Bit 04: PLN. Power limit notification controls are supported if set. Bit 05: ECMD. Clock modulation duty cycle extension is supported if set. Bit 06: PTM. Package thermal management is supported if set. Bit 07: HWP. HWP base registers (IA32_PM_ENALBE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set. Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set. Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set. Bit 10: HWP_Energy_Performance_Preference. IA32_HWP_REQUEST[bits 31:24] is supported if set. Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set. Bit 12: Reserved. Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREAD_STALL MSRs are supported if set. Bits 31 - 15: Reserved
EBX		
	ECX	Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor Bits 31 - 04: Reserved
	EDX	Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MP自称 and IA32_AP自称). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of expected processor performance at frequency specified in CPUID Brand String Bits 02 - 01: Reserved = 0 Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H) Bits 31 - 04: Reserved = 0
<i>Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)</i>		
07H		Sub-leaf 0 (Input ECX = 0). *
	EAX	Bits 31-00: Reports the maximum input value for supported leaf 7 sub-leaves.

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EBX	<p>Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1.</p> <p>Bit 01: IA32_TSC_ADJUST MSR is supported if 1.</p> <p>Bit 02: Reserved</p> <p>Bit 03: BMI1</p> <p>Bit 04: HLE</p> <p>Bit 05: AVX2</p> <p>Bit 06: Reserved</p> <p>Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1.</p> <p>Bit 08: BMI2</p> <p>Bit 09: Supports Enhanced REP MOVSB/STOSB if 1.</p> <p>Bit 10: INVPCID. If 1, supports INVPCID instruction for system software that manages process-context identifiers.</p> <p>Bit 11: RTM</p> <p>Bit 12: Supports Platform Quality of Service Monitoring (PQM) capability if 1.</p> <p>Bit 13: Deprecates FPU CS and FPU DS values if 1.</p> <p>Bit 14: Reserved.</p> <p>Bit 15: Supports Platform Quality of Service Enforcement (PQE) capability if 1.</p> <p>Bits 31:16: Reserved</p>
<i>Direct Cache Access Information Leaf</i>		
09H	EAX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H)
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
<i>Architectural Performance Monitoring Leaf</i>		
0AH	EAX	<p>Bits 07 - 00: Version ID of architectural performance monitoring</p> <p>Bits 15 - 08: Number of general-purpose performance monitoring counter per logical processor</p> <p>Bits 23 - 16: Bit width of general-purpose, performance monitoring counter</p> <p>Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events</p>
	EBX	<p>Bit 00: Core cycle event not available if 1</p> <p>Bit 01: Instruction retired event not available if 1</p> <p>Bit 02: Reference cycles event not available if 1</p> <p>Bit 03: Last-level cache reference event not available if 1</p> <p>Bit 04: Last-level cache misses event not available if 1</p> <p>Bit 05: Branch instruction retired event not available if 1</p> <p>Bit 06: Branch mispredict retired event not available if 1</p> <p>Bits 31 - 07: Reserved = 0</p>
	ECX	Reserved = 0
	EDX	<p>Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1)</p> <p>Bits 12 - 05: Bit width of fixed-function performance counters (if Version ID > 1)</p> <p>Reserved = 0</p>
<i>Extended Topology Enumeration Leaf</i>		

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
OBH	<p>NOTES:</p> <p>Most of Leaf OBH output depends on the initial value in ECX.</p> <p>The EDX output of leaf OBH is always valid and does not vary with input value in ECX.</p> <p>Output value in ECX[7:0] always equals input value in ECX[7:0].</p> <p>For sub-leaves that return an invalid level-type of 0 in ECX[15:8], EAX and EBX will return 0.</p> <p>If an input value n in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX > n also return 0 in ECX[15:8].</p>
<i>Processor Extended State Enumeration Main Leaf (EAX = ODH, ECX = 0)</i>	
ODH	<p>NOTES:</p> <p>Leaf ODH main leaf (ECX = 0).</p>
EAX	<p>Bits 31-00: Reports the valid bit fields of the lower 32 bits of XCRO. If a bit is 0, the corresponding bit field in XCRO is reserved.</p> <p>Bit 00: legacy x87</p> <p>Bit 01: 128-bit SSE</p> <p>Bit 02: 256-bit AVX</p> <p>Bits 31 - 03: Reserved</p>
EBX	<p>Bits 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCRO. May be different than ECX if some features at the end of the XSAVE save area are not enabled.</p>
ECX	<p>Bit 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e all the valid bit fields in XCRO.</p>
EDX	<p>Bit 31-00: Reports the valid bit fields of the upper 32 bits of XCRO. If a bit is 0, the corresponding bit field in XCRO is reserved.</p>

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
<i>Processor Extended State Enumeration Sub-leaf (EAX = ODH, ECX = 1)</i>		
ODH	EAX	<p>Bits 31-04: Reserved</p> <p>Bit 00: XSAVEOPT is available</p> <p>Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set</p> <p>Bit 02: Supports XGETBV with ECX = 1 if set</p> <p>Bit 03: Supports XSAVES/XRSTORS and IA32_XSS if set</p>
	EBX	Bits 31-00: The size in bytes of the XSAVE area containing all states enabled by XCRO IA32_XSS.
	ECX	<p>Bits 31-00: Reports the valid bit fields of the lower 32 bits of IA32_XSS. If a bit is 0, the corresponding bit field in IA32_XSS is reserved.</p> <p>Bits 07-00: Reserved</p> <p>Bit 08: IA32_XSS[bit 8] is supported if 1</p> <p>Bits 31-09: Reserved</p>
	EDX	<p>Bits 31-00: Reports the valid bit fields of the upper 32 bits of IA32_XSS. If a bit is 0, the corresponding bit field in IA32_XSS is reserved.</p> <p>Bits 31-00: Reserved</p>
<i>Processor Extended State Enumeration Sub-leaves (EAX = ODH, ECX = n, n > 1)</i>		
ODH	NOTES:	
	<p>Leaf ODH output depends on the initial value in ECX.</p> <p>Each valid sub-leaf index maps to a valid bit in either the XCRO register or the IA32_XSS MSR starting at bit position 2.</p> <p>* If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Invalid sub-leaves of EAX = ODH: ECX = n, n > 2.</p>	
	EAX	<p>Bits 31-0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, n. This field reports 0 if the sub-leaf index, n, does not map to a valid bit in the XCRO register*.</p>
	EBX	<p>Bits 31-0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area.</p> <p>This field reports 0 if the sub-leaf index, n, is invalid*.</p>
	ECX	This field reports 0 if the sub-leaf index, n, is invalid*; otherwise, bit 0 is set if the sub-leaf index, n, maps to a valid bit in the IA32_XSS MSR, and bits 31-1 are reserved.
	EDX	This field reports 0 if the sub-leaf index, n, is invalid*; otherwise it is reserved.
<i>Platform QoS Monitoring Enumeration Sub-leaf (EAX = OFH, ECX = 0)</i>		
OFH	NOTES:	
	<p>Leaf OFH output depends on the initial value in ECX.</p> <p>Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX</p>	
	EAX	Reserved.
	EBX	Bits 31-0: Maximum range (zero-based) of RMID within this physical processor of all types.
	ECX	Reserved.
	EDX	<p>Bit 00: Reserved.</p> <p>Bit 01: Supports L3 Cache QoS Monitoring if 1.</p> <p>Bits 31:02: Reserved</p>
<i>L3 Cache QoS Monitoring Capability Enumeration Sub-leaf (EAX = OFH, ECX = 1)</i>		

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor			
OFH	NOTES: Leaf OFH output depends on the initial value in ECX.			
	EAX	Reserved.		
	EBX	Bits 31:0: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes).		
	ECX	Maximum range (zero-based) of RMID of this resource type.		
	EDX	Bit 00: Supports L3 occupancy monitoring if 1. Bits 31:01: Reserved		
<i>Platform QoS Enforcement Enumeration Sub-leaf (EAX = 10H, ECX = 0)</i>				
10H	NOTES: Leaf 10H output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EDX			
	EAX	Reserved.		
	EBX	Bit 00: Reserved. Bit 01: Supports L3 Cache QoS Enforcement if 1. Bits 31:02: Reserved		
	ECX	Reserved.		
	EDX	Reserved.		
<i>L3 Cache QoS Enforcement Enumeration Sub-leaf (EAX = 10H, ECX = ResID = 1)</i>				
10H	NOTES: Leaf 10H output depends on the initial value in ECX.			
	EAX	Bits 4:0: Length of the capacity bit mask for the corresponding ResID. Bits 31:05: Reserved		
	EBX	Bits 31:0: Bit-granular map of isolation/contention of allocation units.		
	ECX	Bit 00: Reserved. Bit 01: Updates of COS should be infrequent if 1. Bits 31:02: Reserved		
	EDX	Bits 15:0: Highest COS number supported for this ResID. Bits 31:16: Reserved		
<i>Unimplemented CPUID Leaf Functions</i>				
40000000H - 4FFFFFFFH	Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH.			
<i>Extended Function CPUID Information</i>				
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information (see Table 3-18).		
	EBX	Reserved		
	ECX	Reserved		
	EDX	Reserved		

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor		
80000001H	EAX	Extended Processor Signature and Feature Bits.	
	EBX	Reserved	
	ECX	Bit 00: LAHF/SAHF available in 64-bit mode Bits 04-01 Reserved Bit 05: LZCNT Bits 07-06 Reserved Bit 08: PREFETCHW Bits 31-09 Reserved	
	EDX	Bits 10-00: Reserved Bit 11: SYSCALL/SYSRET available in 64-bit mode Bits 19-12: Reserved = 0 Bit 20: Execute Disable Bit available Bits 25-21: Reserved = 0 Bit 26: 1-GByte pages are available if 1 Bit 27: RDTSCP and IA32_TSC_AUX are available if 1 Bits 28: Reserved = 0 Bit 29: Intel® 64 Architecture available if 1 Bits 31-30: Reserved = 0	
80000002H	EAX	Processor Brand String	
	EBX	Processor Brand String Continued	
	ECX	Processor Brand String Continued	
	EDX	Processor Brand String Continued	
80000003H	EAX	Processor Brand String Continued	
	EBX	Processor Brand String Continued	
	ECX	Processor Brand String Continued	
	EDX	Processor Brand String Continued	
80000004H	EAX	Processor Brand String Continued	
	EBX	Processor Brand String Continued	
	ECX	Processor Brand String Continued	
	EDX	Processor Brand String Continued	
80000005H	EAX	Reserved = 0	
	EBX	Reserved = 0	
	ECX	Reserved = 0	
	EDX	Reserved = 0	
80000006H	EAX	Reserved = 0	
	EBX	Reserved = 0	
	ECX	Bits 07-00: Cache Line size in bytes Bits 11-08: Reserved Bits 15-12: L2 Associativity field * Bits 31-16: Cache size in 1K units Reserved = 0	
	EDX		

Table 3-17. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor		
	NOTES: * L2 associativity field encodings: 00H - Disabled 01H - Direct mapped 02H - 2-way 04H - 4-way 06H - 8-way 08H - 16-way 0FH - Fully associative		
80000007H	EAX	Reserved = 0	
	EBX	Reserved = 0	
	ECX	Reserved = 0	
	EDX	Bits 07-00: Reserved = 0 Bit 08: Invariant TSC available if 1 Bits 31-09: Reserved = 0	
80000008H	EAX	Linear/Physical Address size Bits 07-00: #Physical Address Bits* Bits 15-8: #Linear Address Bits Bits 31-16: Reserved = 0	
	EBX	Reserved = 0	
	ECX	Reserved = 0	
	EDX	Reserved = 0	
	NOTES: * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.		

INPUT EAX = 0: Returns CPUID's Highest Value for Basic Processor Information and the Vendor Identification String

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register (see Table 3-18) and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string is "GenuineIntel" and is expressed:

EBX ← 756e6547h (* "Genu", with G in the low eight bits of BL *)

EDX ← 49656e69h (* "inel", with i in the low eight bits of DL *)

ECX ← 6c65746eh (* "ntel", with n in the low eight bits of CL *)

INPUT EAX = 80000000H: Returns CPUID's Highest Value for Extended Processor Information

When CPUID executes with EAX set to 80000000H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register (see Table 3-18) and is processor specific.

Table 3-18. Highest CPUID Source Operand for Intel 64 and IA-32 Processors

Intel 64 or IA-32 Processors	Highest Value in EAX	
	Basic Information	Extended Function Information
Earlier Intel486 Processors	CPUID Not Implemented	CPUID Not Implemented
Later Intel486 Processors and Pentium Processors	01H	Not Implemented

Table 3-18. Highest CPUID Source Operand for Intel 64 and IA-32 Processors (Contd.)

Intel 64 or IA-32 Processors	Highest Value in EAX	
	Basic Information	Extended Function Information
Pentium Pro and Pentium II Processors, Intel® Celeron® Processors	02H	Not Implemented
Pentium III Processors	03H	Not Implemented
Pentium 4 Processors	02H	80000004H
Intel Xeon Processors	02H	80000004H
Pentium M Processor	02H	80000004H
Pentium 4 Processor supporting Hyper-Threading Technology	05H	80000008H
Pentium D Processor (8xx)	05H	80000008H
Pentium D Processor (9xx)	06H	80000008H
Intel Core Duo Processor	0AH	80000008H
Intel Core 2 Duo Processor	0AH	80000008H
Intel Xeon Processor 3000, 5100, 5200, 5300, 5400 Series	0AH	80000008H
Intel Core 2 Duo Processor 8000 Series	0DH	80000008H
Intel Xeon Processor 5200, 5400 Series	0AH	80000008H
Intel Atom Processor	0AH	80000008H
Intel Core i7 Processor	0BH	80000008H

IA32_BIOS_SIGN_ID Returns Microcode Update Signature

For processors that support the microcode update facility, the IA32_BIOS_SIGN_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 9 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

INPUT EAX = 1: Returns Model, Family, Stepping Information

When CPUID executes with EAX set to 1, version information is returned in EAX (see Figure 3-5). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

- Model — 1111B
- Family — 0101B
- Processor Type — 00B

See Table 3-19 for available processor type values. Stepping IDs are provided as needed.

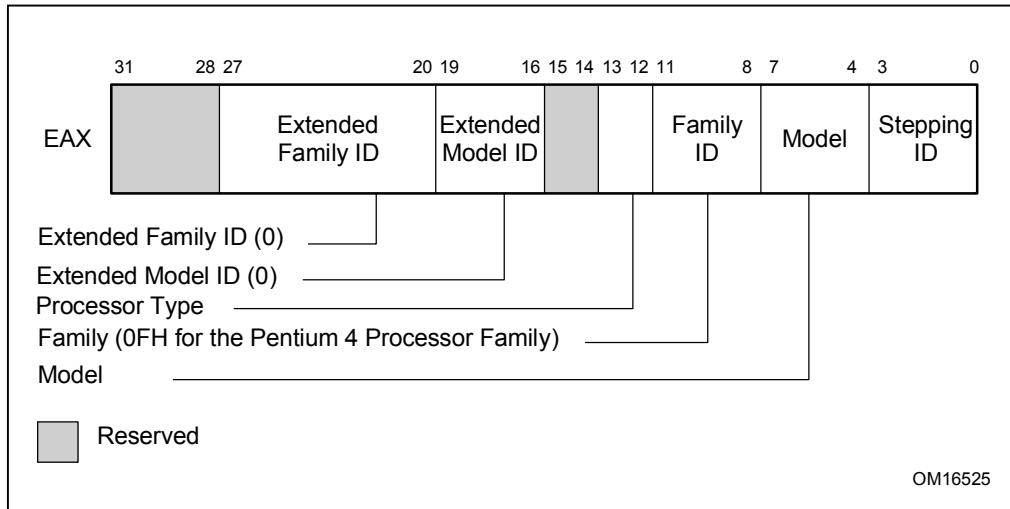


Figure 3-5. Version Information Returned by CPUID in EAX

Table 3-19. Processor Type Field

Type	Encoding
Original OEM Processor	00B
Intel OverDrive® Processor	01B
Dual processor (not applicable to Intel486 processors)	10B
Intel reserved	11B

NOTE

See Chapter 17 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for information on identifying earlier IA-32 processors.

The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

```
IF Family_ID ≠ 0FH
    THEN DisplayFamily = Family_ID;
    ELSE DisplayFamily = Extended_Family_ID + Family_ID;
        (* Right justify and zero-extend 4-bit field. *)
FI;
(* Show DisplayFamily as HEX field. *)
```

The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

```
IF (Family_ID = 06H or Family_ID = 0FH)
    THEN DisplayModel = (Extended_Model_ID << 4) + Model_ID;
        (* Right justify and zero-extend 4-bit field; display Model_ID as HEX field.*)
    ELSE DisplayModel = Model_ID;
FI;
(* Show DisplayModel as HEX field. *)
```

INPUT EAX = 1: Returns Additional Information in EBX

When CPUID executes with EAX set to 1, additional information is returned to the EBX register:

- Brand index (low byte of EBX) — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- CLFLUSH instruction cache line size (second byte of EBX) — this number indicates the size of the cache line flushed with CLFLUSH instruction in 8-byte increments. This field was introduced in the Pentium 4 processor.
- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

INPUT EAX = 1: Returns Feature Information in ECX and EDX

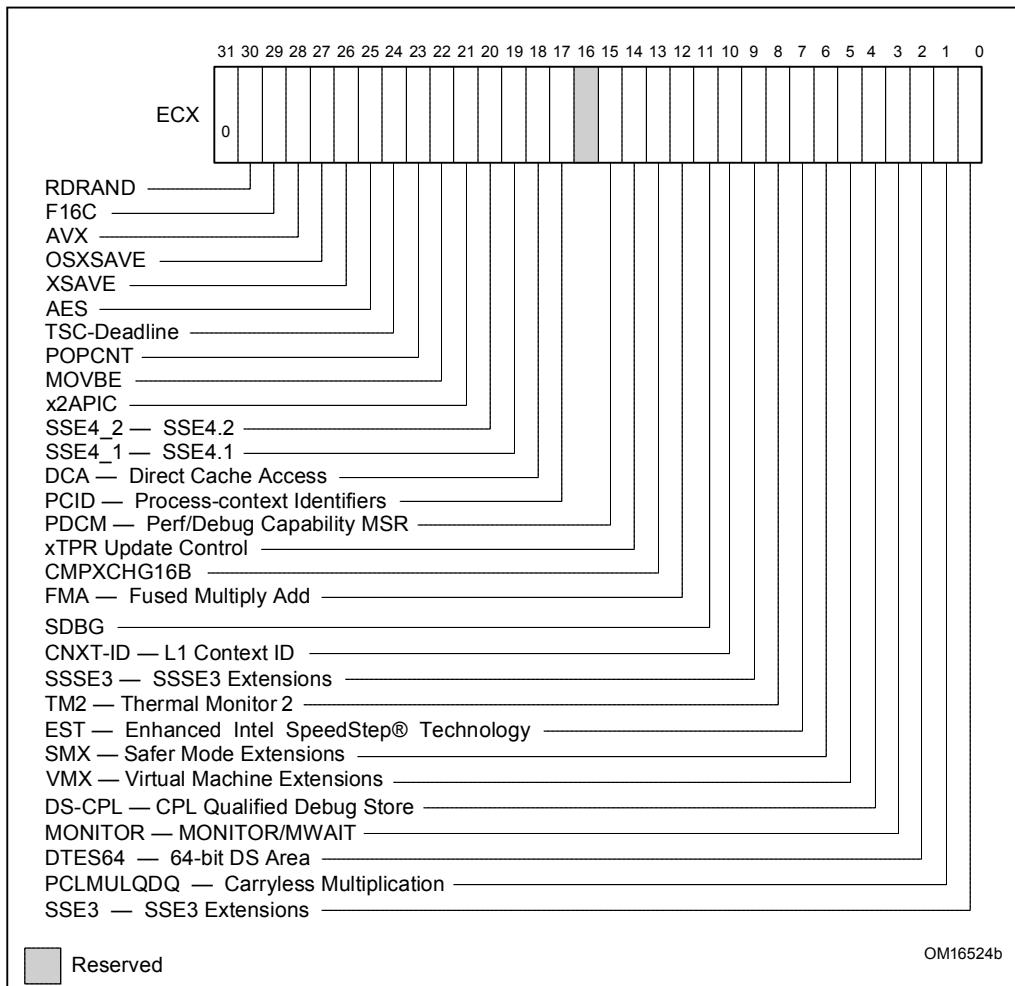
When CPUID executes with EAX set to 1, feature information is returned in ECX and EDX.

- Figure 3-6 and Table 3-20 show encodings for ECX.
- Figure 3-7 and Table 3-21 show encodings for EDX.

For all feature flags, a 1 indicates that the feature is supported. Use Intel to properly interpret feature flags.

NOTE

Software must confirm that a processor feature is present using feature flags returned by CPUID prior to using the feature. Software should not depend on future offerings retaining all features.

**Figure 3-6. Feature Information Returned in the ECX Register****Table 3-20. Feature Information Returned in the ECX Register**

Bit #	Mnemonic	Description
0	SSE3	Streaming SIMD Extensions 3 (SSE3). A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	PCLMULQDQ. A value of 1 indicates the processor supports the PCLMULQDQ instruction
2	DTES64	64-bit DS Area. A value of 1 indicates the processor supports DS area using 64-bit layout
3	MONITOR	MONITOR/MWAIT. A value of 1 indicates the processor supports this feature.
4	DS-CPL	CPL Qualified Debug Store. A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	Virtual Machine Extensions. A value of 1 indicates that the processor supports this technology
6	SMX	Safer Mode Extensions. A value of 1 indicates that the processor supports this technology. See Chapter 5, "Safer Mode Extensions Reference".
7	EIST	Enhanced Intel SpeedStep® technology. A value of 1 indicates that the processor supports this technology.
8	TM2	Thermal Monitor 2. A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor

Table 3-20. Feature Information Returned in the ECX Register (Contd.)

Bit #	Mnemonic	Description
10	CNXT-ID	L1 Context ID. A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	SDBG	A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	CMPXCHG16B Available. A value of 1 indicates that the feature is available. See the “CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes” section in this chapter for a description.
14	xTPR Update Control	xTPR Update Control. A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].
15	PDCM	Perfmon and Debug Capability: A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	Process-context identifiers. A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor’s local APIC timer supports one-shot operation using a TSC deadline value.
25	AESNI	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCRO.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable the XSAVE feature set.
28	AVX	A value of 1 indicates the processor supports the AVX instruction extensions.
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always returns 0.

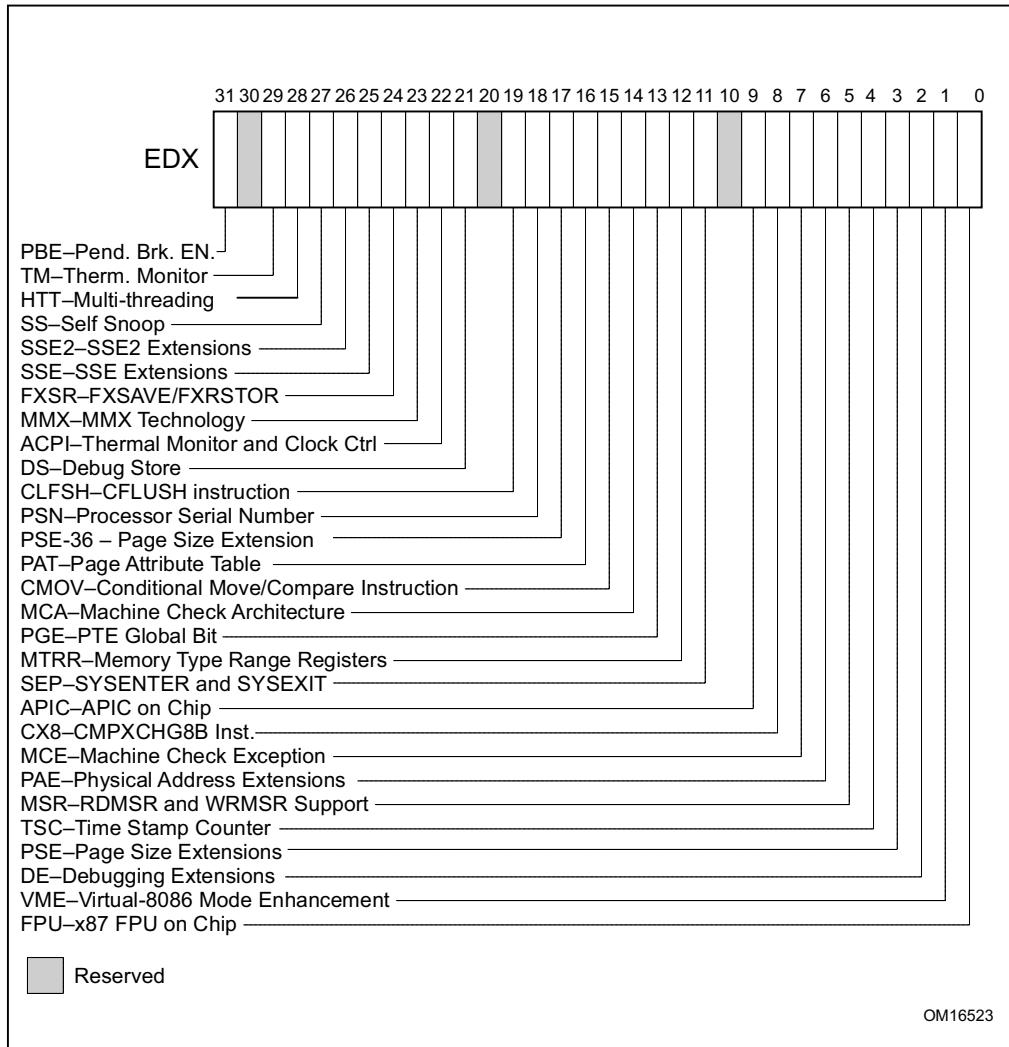


Figure 3-7. Feature Information Returned in the EDX Register

Table 3-21. More on Feature Information Returned in the EDX Register

Bit #	Mnemonic	Description
0	FPU	Floating Point Unit On-Chip. The processor contains an x87 FPU.
1	VME	Virtual 8086 Mode Enhancements. Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	Debugging Extensions. Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	Page Size Extension. Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	Time Stamp Counter. The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	Model Specific Registers RDMSR and WRMSR Instructions. The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.
6	PAE	Physical Address Extension. Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1.
7	MCE	Machine Check Exception. Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	CMPXCHG8B Instruction. The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	APIC On-Chip. The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	SYSENTER and SYSEXIT Instructions. The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	Memory Type Range Registers. MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	Page Global Bit. The global bit is supported in paging-structure entries that map a page, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	Machine Check Architecture. The Machine Check Architecture, which provides a compatible mechanism for error reporting in P6 family, Pentium 4, Intel Xeon processors, and future processors, is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	Conditional Move Instructions. The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	Page Attribute Table. Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory accessed through a linear address on a 4KB granularity.
17	PSE-36	36-Bit Page Size Extension. 4-MByte pages addressing physical memory beyond 4 GBytes are supported with 32-bit paging. This feature indicates that upper bits of the physical address of a 4-MByte page are encoded in bits 20:13 of the page-directory entry. Such physical addresses are limited by MAXPHYADDR and may be up to 40 bits in size.
18	PSN	Processor Serial Number. The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	CLFLUSH Instruction. CLFLUSH Instruction is supported.
20	Reserved	Reserved

Table 3-21. More on Feature Information Returned in the EDX Register (Contd.)

Bit #	Mnemonic	Description
21	DS	Debug Store. The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 23, “Introduction to Virtual-Machine Extensions,” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C</i>).
22	ACPI	Thermal Monitor and Software Controlled Clock Facilities. The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	Intel MMX Technology. The processor supports the Intel MMX technology.
24	FXSR	FXSAVE and FXRSTOR Instructions. The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.
25	SSE	SSE. The processor supports the SSE extensions.
26	SSE2	SSE2. The processor supports the SSE2 extensions.
27	SS	Self Snoop. The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	Max APIC IDs reserved field is Valid. A value of 0 for HTT indicates there is only a single logical processor in the package and software should assume only a single APIC ID is reserved. A value of 1 for HTT indicates the value in CPUID.1.EBX[23:16] (the Maximum number of addressable IDs for logical processors in this package) is valid for the package.
29	TM	Thermal Monitor. The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	Pending Break Enable. The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

INPUT EAX = 2: TLB/Cache/Prefetch Information Returned in EAX, EBX, ECX, EDX

When CPUID executes with EAX set to 2, the processor returns information about the processor’s internal TLBs, cache and prefetch hardware in the EAX, EBX, ECX, and EDX registers. The information is reported in encoded form and fall into the following categories:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor’s TLB/Cache/Prefetch hardware. The Intel Xeon processor 7400 series will return a 1.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. There are four types of encoding values for the byte descriptor, the encoding type is noted in the second column of Table 3-22. Table 3-22 lists the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache, prefetch, or TLB types. The descriptors may appear in any order. Note also a processor may report a general descriptor type (FFH) and not report any byte descriptor of “cache type” via CPUID leaf 2.

Table 3-22. Encoding of CPUID Leaf 2 Descriptors

Value	Type	Description
00H	General	Null descriptor, this byte contains no information
01H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	TLB	Instruction TLB: 4 MByte pages, fully associative, 2 entries
03H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	TLB	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	TLB	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	Cache	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	Cache	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
09H	Cache	1st-level instruction cache: 32KBytes, 4-way set associative, 64 byte line size
0AH	Cache	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	TLB	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
0DH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size
0EH	Cache	1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size
1DH	Cache	2nd-level cache: 128 KBytes, 2-way set associative, 64 byte line size
21H	Cache	2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size
22H	Cache	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	Cache	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
24H	Cache	2nd-level cache: 1 MBytes, 16-way set associative, 64 byte line size
25H	Cache	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	Cache	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	Cache	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	Cache	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	Cache	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	Cache	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	Cache	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	Cache	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	Cache	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	Cache	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	Cache	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
48H	Cache	2nd-level cache: 3MByte, 12-way set associative, 64 byte line size
49H	Cache	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
4AH	Cache	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	Cache	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	Cache	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size
4DH	Cache	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size
4EH	Cache	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
4FH	TLB	Instruction TLB: 4 KByte pages, 32 entries

Table 3-22. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
50H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
55H	TLB	Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries
56H	TLB	Data TLBO: 4 MByte pages, 4-way set associative, 16 entries
57H	TLB	Data TLBO: 4 KByte pages, 4-way associative, 16 entries
59H	TLB	Data TLBO: 4 KByte pages, fully associative, 16 entries
5AH	TLB	Data TLBO: 2-MByte or 4 MByte pages, 4-way set associative, 32 entries
5BH	TLB	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	TLB	Data TLB: 4 KByte and 4 MByte pages, 128 entries
5DH	TLB	Data TLB: 4 KByte and 4 MByte pages, 256 entries
60H	Cache	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
61H	TLB	Instruction TLB: 4 KByte pages, fully associative, 48 entries
63H	TLB	Data TLB: 1 GByte pages, 4-way set associative, 4 entries
66H	Cache	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	Cache	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	Cache	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
70H	Cache	Trace cache: 12 K-μop, 8-way set associative
71H	Cache	Trace cache: 16 K-μop, 8-way set associative
72H	Cache	Trace cache: 32 K-μop, 8-way set associative
76H	TLB	Instruction TLB: 2M/4M pages, fully associative, 8 entries
78H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	Cache	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	Cache	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	Cache	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	Cache	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
80H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size
82H	Cache	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	Cache	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	Cache	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size
A0H	DTLB	DTLB: 4k pages, fully associative, 32 entries
B0H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B1H	TLB	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B2H	TLB	Instruction TLB: 4KByte pages, 4-way set associative, 64 entries
B3H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	TLB	Data TLB1: 4 KByte pages, 4-way associative, 256 entries

Table 3-22. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
B5H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 64 entries
B6H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 128 entries
BAH	TLB	Data TLB1: 4 KByte pages, 4-way associative, 64 entries
COH	TLB	Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries
C1H	STLB	Shared 2nd-Level TLB: 4 KByte/2MByte pages, 8-way associative, 1024 entries
C2H	DTLB	DTLB: 4 KByte/2 MByte pages, 4-way associative, 16 entries
CAH	STLB	Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries
D0H	Cache	3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size
D1H	Cache	3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size
D2H	Cache	3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size
D6H	Cache	3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size
D7H	Cache	3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size
D8H	Cache	3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size
DCH	Cache	3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size
DDH	Cache	3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size
DEH	Cache	3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size
E2H	Cache	3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size
E3H	Cache	3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size
E4H	Cache	3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size
EAH	Cache	3rd-level cache: 12MByte, 24-way set associative, 64 byte line size
EBH	Cache	3rd-level cache: 18MByte, 24-way set associative, 64 byte line size
ECH	Cache	3rd-level cache: 24MByte, 24-way set associative, 64 byte line size
F0H	Prefetch	64-Byte prefetching
F1H	Prefetch	128-Byte prefetching
FFH	General	CPUID leaf 2 does not report cache descriptor information, use CPUID leaf 4 to query cache parameters

Example 3-1. Example of Cache and TLB Interpretation

The first member of the family of Pentium 4 processors returns the following information about caches and TLBs when the CPUID executes with an input value of 2:

```

EAX  66 5B 50 01H
EBX  0H
ECX  0H
EDX  00 7A 70 00H

```

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This indicates that CPUID needs to be executed once with an input value of 2 to retrieve complete information about caches and TLBs.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
 - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
 - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
 - 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.

- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
 - 00H - NULL descriptor.
 - 70H - Trace cache: 12 K-μop, 8-way set associative.
 - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
 - 00H - NULL descriptor.

INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level

When CPUID executes with EAX set to 04H and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-17.

This Cache Size in Bytes

$$\begin{aligned} &= (\text{Ways} + 1) * (\text{Partitions} + 1) * (\text{Line_Size} + 1) * (\text{Sets} + 1) \\ &= (\text{EBX}[31:22] + 1) * (\text{EBX}[21:12] + 1) * (\text{EBX}[11:0] + 1) * (\text{ECX} + 1) \end{aligned}$$

The CPUID leaf 04H also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 8, “Multiple-Processor Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

INPUT EAX = 05H: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-17.

INPUT EAX = 06H: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 3-17.

INPUT EAX = 07H: Returns Structured Extended Feature Enumeration Information

When CPUID executes with EAX set to 07H and ECX = 0, the processor returns information about the maximum input value for sub-leaves that contain extended feature flags. See Table 3-17.

When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-17), the processor returns 0 in EAX/EBX/ECX/EDX. In subleaf 0, EAX returns the maximum input value of the highest leaf 7 sub-leaf, and EBX, ECX & EDX contain information of extended feature flags.

INPUT EAX = 09H: Returns Direct Cache Access Information

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 3-17.

INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-17) is greater than Pn 0. See Table 3-17.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 23, “Introduction to Virtual-Machine Extensions,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*.

INPUT EAX = 0BH: Returns Extended Topology Information

When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is \geq 0BH, and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 3-17.

INPUT EAX = 0DH: Returns Processor Extended States Enumeration Information

When CPUID executes with EAX set to 0DH and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 3-17.

When CPUID executes with EAX set to 0DH and ECX = n ($n > 1$, and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 3-17. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:

```
For i = 2 to 62 // sub-leaf 1 is reserved
  IF (CPUID.(EAX=0DH, ECX=0):VECTOR[i] = 1 ) // VECTOR is the 64-bit value of EDX:EAX
    Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;
  Fl;
```

INPUT EAX = 0FH: Returns Platform Quality of Service (PQoS) Monitoring Enumeration Information

When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor of any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 3-17.

When CPUID executes with EAX set to 0FH and ECX = n ($n \geq 1$, and is a valid ResID), the processor returns information software can use to program IA32_PQR_ASSOC, IA32_QM_EVTSEL MSRs before reading QoS data from the IA32_QM_CTR MSR.

INPUT EAX = 10H: Returns Platform Quality of Service (PQoS) Enforcement Enumeration Information

When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 3-17.

When CPUID executes with EAX set to 10H and ECX = n ($n \geq 1$, and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32_resourceType_Mask_n.

METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

1. Processor brand string method; this method also returns the processor’s maximum operating frequency
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: "Identification of Earlier IA-32 Processors" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The Processor Brand String Method

Figure 3-8 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the maximum operating frequency of the processor to the EAX, EBX, ECX, and EDX registers.

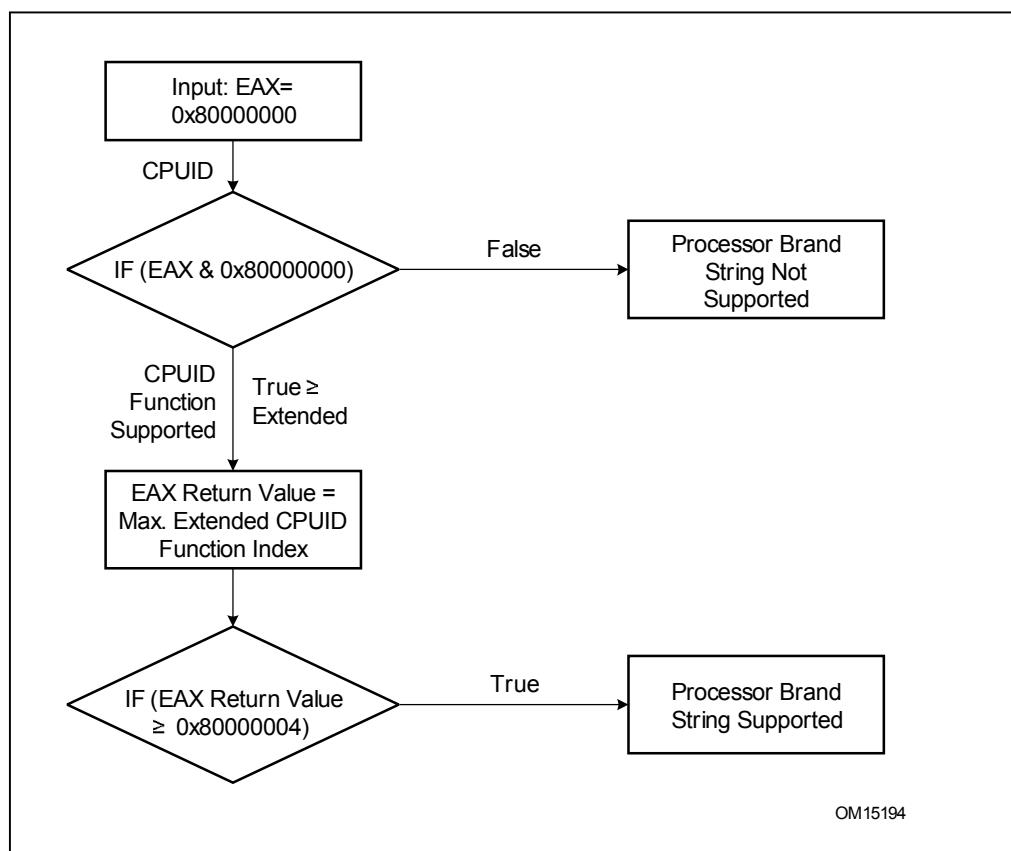


Figure 3-8. Determination of Support for the Processor Brand String

How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 8000002H through 8000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

Table 3-23 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 3-23. Processor Brand String Returned with Pentium 4 Processor

EAX Input Value	Return Values	ASCII Equivalent
-----------------	---------------	------------------

Table 3-23. Processor Brand String Returned with Pentium 4 Processor (Contd.)

80000002H	EAX = 20202020H EBX = 20202020H ECX = 20202020H EDX = 6E492020H	" " " " " " "nl "
80000003H	EAX = 286C6574H EBX = 50202952H ECX = 69746E65H EDX = 52286D75H	"(let" "P)R" "itne" "R(mu"
80000004H	EAX = 20342029H EBX = 20555043H ECX = 30303531H EDX = 007A484DH	" 4)" " UPC" "0051" "\0zHM"

Extracting the Maximum Processor Frequency from Brand Strings

Figure 3-9 provides an algorithm which software can use to extract the maximum processor operating frequency from the processor brand string.

NOTE

When a frequency is given in a brand string, it is the maximum qualified frequency of the processor, not the frequency at which the processor is currently running.

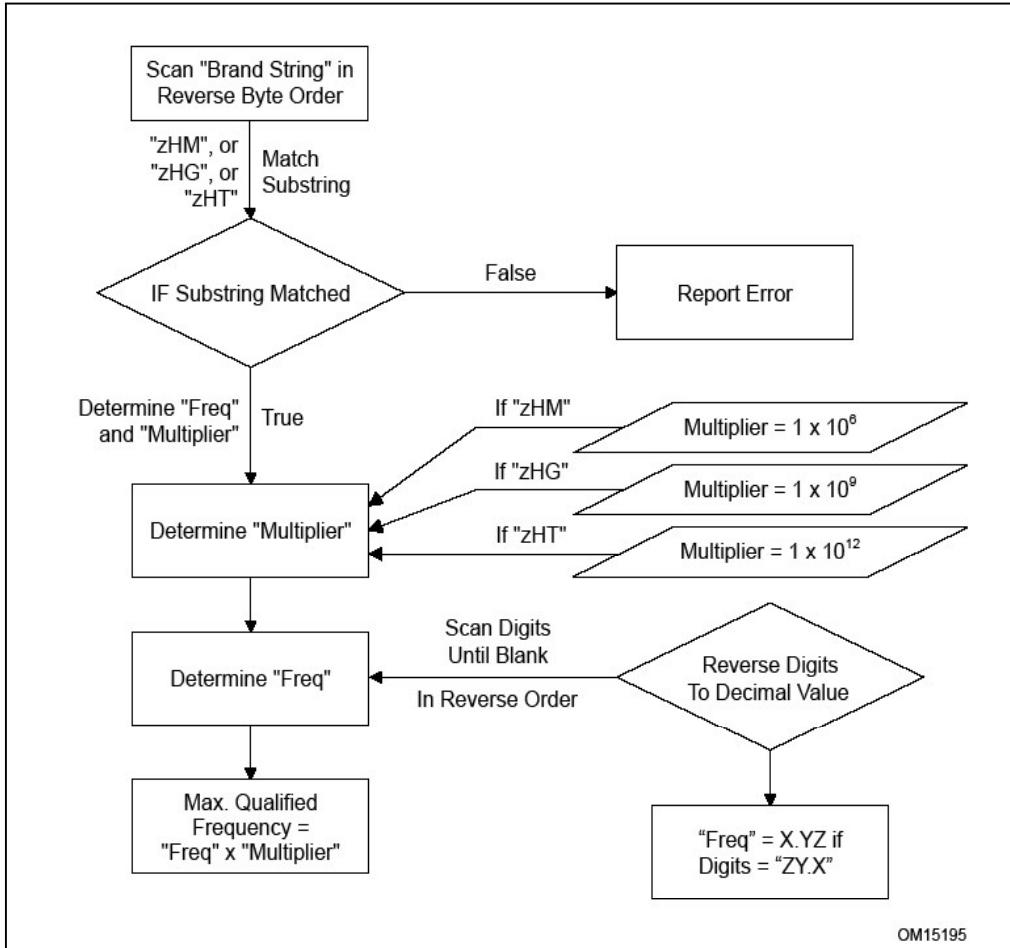


Figure 3-9. Algorithm for Extracting Maximum Processor Frequency

The Processor Brand Index Method

The brand index method (introduced with Pentium® III Xeon® processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associated with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 3-24 shows brand indices that have identification strings associated with them.

Table 3-24. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

Brand Index	Brand String
00H	This processor does not support the brand identification feature
01H	Intel(R) Celeron(R) processor ¹
02H	Intel(R) Pentium(R) III processor ¹

Table 3-24. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

03H	Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor
04H	Intel(R) Pentium(R) III processor
06H	Mobile Intel(R) Pentium(R) III processor-M
07H	Mobile Intel(R) Celeron(R) processor ¹
08H	Intel(R) Pentium(R) 4 processor
09H	Intel(R) Pentium(R) 4 processor
0AH	Intel(R) Celeron(R) processor ¹
0BH	Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP
0CH	Intel(R) Xeon(R) processor MP
0EH	Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor
0FH	Mobile Intel(R) Celeron(R) processor ¹
11H	Mobile Genuine Intel(R) processor
12H	Intel(R) Celeron(R) M processor
13H	Mobile Intel(R) Celeron(R) processor ¹
14H	Intel(R) Celeron(R) processor
15H	Mobile Genuine Intel(R) processor
16H	Intel(R) Pentium(R) M processor
17H	Mobile Intel(R) Celeron(R) processor ¹
18H - OFFH	RESERVED

NOTES:

1. Indicates versions of these processors that were introduced after the Pentium III

IA-32 Architecture Compatibility

CPUID is not supported in early models of the Intel486 processor or in any IA-32 processor earlier than the Intel486 processor.

Operation

IA32_BIOS_SIGN_ID MSR ← Update with installed microcode revision number;

CASE (EAX) OF

 EAX = 0:

 EAX ← Highest basic function input value understood by CPUID;
 EBX ← Vendor identification string;
 EDX ← Vendor identification string;
 ECX ← Vendor identification string;

 BREAK;

 EAX = 1H:

 EAX[3:0] ← Stepping ID;
 EAX[7:4] ← Model;
 EAX[11:8] ← Family;
 EAX[13:12] ← Processor type;
 EAX[15:14] ← Reserved;
 EAX[19:16] ← Extended Model;
 EAX[27:20] ← Extended Family;

```

EAX[31:28] ← Reserved;
EBX[7:0] ← Brand Index; (* Reserved if the value is zero. *)
EBX[15:8] ← CLFLUSH Line Size;
EBX[16:23] ← Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
EBX[24:31] ← Initial APIC ID;
ECX ← Feature flags; (* See Figure 3-6. *)
EDX ← Feature flags; (* See Figure 3-7. *)
BREAK;
EAX = 2H:
    EAX ← Cache and TLB information;
    EBX ← Cache and TLB information;
    ECX ← Cache and TLB information;
    EDX ← Cache and TLB information;
BREAK;
EAX = 3H:
    EAX ← Reserved;
    EBX ← Reserved;
    ECX ← ProcessorSerialNumber[31:0];
    (* Pentium III processors only, otherwise reserved. *)
    EDX ← ProcessorSerialNumber[63:32];
    (* Pentium III processors only, otherwise reserved. *)
BREAK
EAX = 4H:
    EAX ← Deterministic Cache Parameters Leaf; (* See Table 3-17. *)
    EBX ← Deterministic Cache Parameters Leaf;
    ECX ← Deterministic Cache Parameters Leaf;
    EDX ← Deterministic Cache Parameters Leaf;
BREAK;
EAX = 5H:
    EAX ← MONITOR/MWAIT Leaf; (* See Table 3-17. *)
    EBX ← MONITOR/MWAIT Leaf;
    ECX ← MONITOR/MWAIT Leaf;
    EDX ← MONITOR/MWAIT Leaf;
BREAK;
EAX = 6H:
    EAX ← Thermal and Power Management Leaf; (* See Table 3-17. *)
    EBX ← Thermal and Power Management Leaf;
    ECX ← Thermal and Power Management Leaf;
    EDX ← Thermal and Power Management Leaf;
BREAK;
EAX = 7H:
    EAX ← Structured Extended Feature Flags Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Structured Extended Feature Flags Enumeration Leaf;
    ECX ← Structured Extended Feature Flags Enumeration Leaf;
    EDX ← Structured Extended Feature Flags Enumeration Leaf;
BREAK;
EAX = 8H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = 9H:
    EAX ← Direct Cache Access Information Leaf; (* See Table 3-17. *)

```

```

EBX ← Direct Cache Access Information Leaf;
ECX ← Direct Cache Access Information Leaf;
EDX ← Direct Cache Access Information Leaf;
BREAK;
EAX = AH:
    EAX ← Architectural Performance Monitoring Leaf; (* See Table 3-17. *)
    EBX ← Architectural Performance Monitoring Leaf;
    ECX ← Architectural Performance Monitoring Leaf;
    EDX ← Architectural Performance Monitoring Leaf;
    BREAK
EAX = BH:
    EAX ← Extended Topology Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Extended Topology Enumeration Leaf;
    ECX ← Extended Topology Enumeration Leaf;
    EDX ← Extended Topology Enumeration Leaf;
BREAK;
EAX = CH:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = DH:
    EAX ← Processor Extended State Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Processor Extended State Enumeration Leaf;
    ECX ← Processor Extended State Enumeration Leaf;
    EDX ← Processor Extended State Enumeration Leaf;
BREAK;
EAX = EH:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = FH:
    EAX ← Platform Quality of Service Monitoring Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Platform Quality of Service Monitoring Enumeration Leaf;
    ECX ← Platform Quality of Service Monitoring Enumeration Leaf;
    EDX ← Platform Quality of Service Monitoring Enumeration Leaf;
BREAK;
EAX = 10H:
    EAX ← Platform Quality of Service Enforcement Enumeration Leaf; (* See Table 3-17. *)
    EBX ← Platform Quality of Service Enforcement Enumeration Leaf;
    ECX ← Platform Quality of Service Enforcement Enumeration Leaf;
    EDX ← Platform Quality of Service Enforcement Enumeration Leaf;
BREAK;
EAX = 80000000H:
    EAX ← Highest extended function input value understood by CPUID;
    EBX ← Reserved;
    ECX ← Reserved;
    EDX ← Reserved;
BREAK;
EAX = 80000001H:

```

```

EAX ← Reserved;
EBX ← Reserved;
ECX ← Extended Feature Bits (* See Table 3-17.*);
EDX ← Extended Feature Bits (* See Table 3-17. *);
BREAK;
EAX = 80000002H:
EAX ← Processor Brand String;
EBX ← Processor Brand String, continued;
ECX ← Processor Brand String, continued;
EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000003H:
EAX ← Processor Brand String, continued;
EBX ← Processor Brand String, continued;
ECX ← Processor Brand String, continued;
EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000004H:
EAX ← Processor Brand String, continued;
EBX ← Processor Brand String, continued;
ECX ← Processor Brand String, continued;
EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000005H:
EAX ← Reserved = 0;
EBX ← Reserved = 0;
ECX ← Reserved = 0;
EDX ← Reserved = 0;
BREAK;
EAX = 80000006H:
EAX ← Reserved = 0;
EBX ← Reserved = 0;
ECX ← Cache information;
EDX ← Reserved = 0;
BREAK;
EAX = 80000007H:
EAX ← Reserved = 0;
EBX ← Reserved = 0;
ECX ← Reserved = 0;
EDX ← Reserved = Misc Feature Flags;
BREAK;
EAX = 80000008H:
EAX ← Reserved = Physical Address Size Information;
EBX ← Reserved = Virtual Address Size Information;
ECX ← Reserved = 0;
EDX ← Reserved = 0;
BREAK;
EAX >= 40000000H and EAX <= 4FFFFFFFH:
DEFAULT: (* EAX = Value outside of recognized range for CPUID. *)
(* If the highest basic information leaf data depend on ECX input value, ECX is honored.*)
EAX ← Reserved; (* Information returned for highest basic information leaf. *)
EBX ← Reserved; (* Information returned for highest basic information leaf. *)
ECX ← Reserved; (* Information returned for highest basic information leaf. *)
EDX ← Reserved; (* Information returned for highest basic information leaf. *)

```

BREAK;
ESAC;

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated.

CRC32 – Accumulate CRC32 Value

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 OF 38 F0 /r CRC32 r32, r/m8	RM	Valid	Valid	Accumulate CRC32 on r/m8.
F2 REX OF 38 F0 /r CRC32 r32, r/m8*	RM	Valid	N.E.	Accumulate CRC32 on r/m8.
F2 OF 38 F1 /r CRC32 r32, r/m16	RM	Valid	Valid	Accumulate CRC32 on r/m16.
F2 OF 38 F1 /r CRC32 r32, r/m32	RM	Valid	Valid	Accumulate CRC32 on r/m32.
F2 REX.W OF 38 F0 /r CRC32 r64, r/m8	RM	Valid	N.E.	Accumulate CRC32 on r/m8.
F2 REX.W OF 38 F1 /r CRC32 r64, r/m64	RM	Valid	N.E.	Accumulate CRC32 on 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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Starting with an initial value in the first operand (destination operand), accumulates a CRC32 (polynomial 0x11EDC6F41) value for the second operand (source operand) and stores the result in the destination operand. The source operand can be a register or a memory location. The destination operand must be an r32 or r64 register. If the destination is an r64 register, then the 32-bit result is stored in the least significant double word and 00000000H is stored in the most significant double word of the r64 register.

The initial value supplied in the destination operand is a double word integer stored in the r32 register or the least significant double word of the r64 register. To incrementally accumulate a CRC32 value, software retains the result of the previous CRC32 operation in the destination operand, then executes the CRC32 instruction again with new input data in the source operand. Data contained in the source operand is processed in reflected bit order. This means that the most significant bit of the source operand is treated as the least significant bit of the quotient, and so on, for all the bits of the source operand. Likewise, the result of the CRC operation is stored in the destination operand in reflected bit order. This means that the most significant bit of the resulting CRC (bit 31) is stored in the least significant bit of the destination operand (bit 0), and so on, for all the bits of the CRC.

Operation**Notes:**

BIT_REFLECT64: DST[63-0] = SRC[0-63]
 BIT_REFLECT32: DST[31-0] = SRC[0-31]
 BIT_REFLECT16: DST[15-0] = SRC[0-15]
 BIT_REFLECT8: DST[7-0] = SRC[0-7]
 MOD2: Remainder from Polynomial division modulus 2

CRC32 instruction for 64-bit source operand and 64-bit destination operand:

```

TEMP1[63-0] ← BIT_REFLECT64 (SRC[63-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[95-0] ← TEMP1[63-0] << 32
TEMP4[95-0] ← TEMP2[31-0] << 64
TEMP5[95-0] ← TEMP3[95-0] XOR TEMP4[95-0]
TEMP6[31-0] ← TEMP5[95-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])
DEST[63-32] ← 00000000H

```

CRC32 instruction for 32-bit source operand and 32-bit destination operand:

```

TEMP1[31-0] ← BIT_REFLECT32 (SRC[31-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[63-0] ← TEMP1[31-0] << 32
TEMP4[63-0] ← TEMP2[31-0] << 32
TEMP5[63-0] ← TEMP3[63-0] XOR TEMP4[63-0]
TEMP6[31-0] ← TEMP5[63-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])

```

CRC32 instruction for 16-bit source operand and 32-bit destination operand:

```

TEMP1[15-0] ← BIT_REFLECT16 (SRC[15-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[47-0] ← TEMP1[15-0] << 32
TEMP4[47-0] ← TEMP2[31-0] << 16
TEMP5[47-0] ← TEMP3[47-0] XOR TEMP4[47-0]
TEMP6[31-0] ← TEMP5[47-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])

```

CRC32 instruction for 8-bit source operand and 64-bit destination operand:

```

TEMP1[7-0] ← BIT_REFLECT8(SRC[7-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] ← TEMP1[7-0] << 32
TEMP4[39-0] ← TEMP2[31-0] << 8
TEMP5[39-0] ← TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] ← TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])
DEST[63-32] ← 00000000H

```

CRC32 instruction for 8-bit source operand and 32-bit destination operand:

```

TEMP1[7-0] ← BIT_REFLECT8(SRC[7-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] ← TEMP1[7-0] << 32
TEMP4[39-0] ← TEMP2[31-0] << 8
TEMP5[39-0] ← TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] ← TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])

```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

```
unsigned int _mm_crc32_u8( unsigned int crc, unsigned char data )
unsigned int _mm_crc32_u16( unsigned int crc, unsigned short data )
unsigned int _mm_crc32_u32( unsigned int crc, unsigned int data )
unsinged __int64 _mm_crc32_u64( unsinged __int64 crc, unsigned __int64 data )
```

SIMD Floating Point Exceptions

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#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.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0. If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0. If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0. If LOCK prefix is used.

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(0)	If a memory address referencing the SS segment 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.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0. If LOCK prefix is used.

CVTDQ2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF E6 CVTDQ2PD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Convert two packed signed doubleword integers from <i>xmm2/m128</i> to two packed double-precision floating-point values in <i>xmm1</i> .
VEX.128.F3.0F.WIG E6 /r VCVTQ2PD <i>xmm1, xmm2/m64</i>		RM V/V	AVX	Convert two packed signed doubleword integers from <i>xmm2/mem</i> to two packed double-precision floating-point values in <i>xmm1</i> .
VEX.256.F3.0F.WIG E6 /r VCVTQ2PD <i>ymm1, xmm2/m128</i>		RM V/V	AVX	Convert four packed signed doubleword integers from <i>xmm2/mem</i> to four packed double-precision floating-point values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed double-precision floating-point values in the destination operand (first operand).

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

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding XMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 128-bit memory location. The destination operation is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

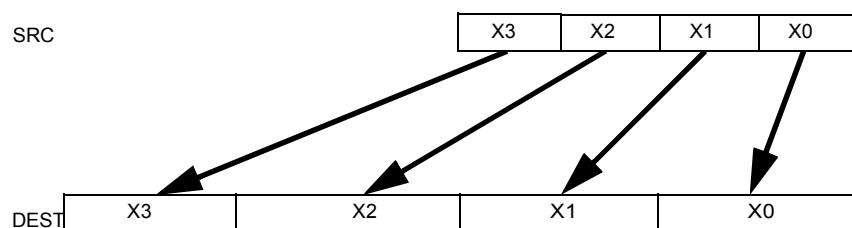


Figure 3-10. CVTDQ2PD (VEX.256 encoded version)

Operation

CVTDQ2PD (128-bit Legacy SSE version)

DEST[63:0] \leftarrow Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
 DEST[127:64] \leftarrow Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
 DEST[VLMAX-1:128] (unmodified)

VCVTDQ2PD (VEX.128 encoded version)

DEST[63:0] \leftarrow Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
 DEST[127:64] \leftarrow Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
 DEST[VLMAX-1:128] \leftarrow 0

VCVTDQ2PD (VEX.256 encoded version)

DEST[63:0] \leftarrow Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
 DEST[127:64] \leftarrow Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
 DEST[191:128] \leftarrow Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
 DEST[255:192] \leftarrow Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96])

Intel C/C++ Compiler Intrinsic Equivalent

CVTDQ2PD: `__m128d _mm_cvtepi32_pd(__m128i a)`
 VCVTDQ2PD: `__m256d _mm256_cvtepi32_pd (__m128i src)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

CVTDQ2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5B /r CVTDQ2PS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Convert four packed signed doubleword integers from <i>xmm2/m128</i> to four packed single-precision floating-point values in <i>xmm1</i> .
VEX.128.0F.WIG 5B /r VCVTDQ2PS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Convert four packed signed doubleword integers from <i>xmm2/mem</i> to four packed single-precision floating-point values in <i>xmm1</i> .
VEX.256.0F.WIG 5B /r VCVTDQ2PS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Convert eight packed signed doubleword integers from <i>ymm2/mem</i> to eight packed single-precision floating-point values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts four packed signed doubleword integers in the source operand (second operand) to four packed single-precision floating-point values in the destination operand (first operand).

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding XMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation**CVTDQ2PS (128-bit Legacy SSE version)**

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
DEST[VLMAX-1:128] (unmodified)
```

VCVTDQ2PS (VEX.128 encoded version)

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
DEST[VLMAX-1:128] ← 0
```

VCVTDQ2PS (VEX.256 encoded version)

```

DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96])
DEST[159:128] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[159:128])
DEST[191:160] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[191:160])
DEST[223:192] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[223:192])
DEST[255:224] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[255:224])

```

Intel C/C++ Compiler Intrinsic Equivalent

CVTDQ2PS: `__m128 _mm_cvtepi32_ps(__m128i a)`
 VCVTDQ2PS: `__m256 _mm256_cvtepi32_ps (__m256i src)`

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

CVTPD2DQ—Convert Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF E6 /r CVTPD2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Convert two packed double-precision floating-point values from <i>xmm2/m128</i> to two packed signed doubleword integers in <i>xmm1</i> .
VEX.128.F2.0F.WIG E6 /r VCVTPD2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Convert two packed double-precision floating-point values in <i>xmm2/mem</i> to two signed doubleword integers in <i>xmm1</i> .
VEX.256.F2.0F.WIG E6 /r VCVTPD2DQ <i>xmm1, ymm2/m256</i>	RM	V/V	AVX	Convert four packed double-precision floating-point values in <i>ymm2/mem</i> to four signed doubleword integers in <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand and the high quadword is cleared to all 0s.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:64) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

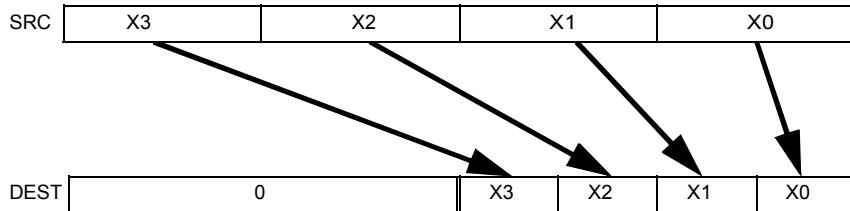


Figure 3-11. VCVTPD2DQ (VEX.256 encoded version)

Operation

CVTPD2DQ (128-bit Legacy SSE version)

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[127:64] ← 0
DEST[VLMAX-1:128] (unmodified)
```

VCVTPD2DQ (VEX.128 encoded version)

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[VLMAX-1:64] ← 0
```

VCVTPD2DQ (VEX.256 encoded version)

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[255:192])
DEST[255:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2DQ: __m128i _mm_cvtpd_epi32 (__m128d src)
 CVTPD2DQ: __m128i _mm256_cvtpd_epi32 (__m256d src)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

CVTPD2PI—Convert Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2D /r CVTPD2PI mm, xmm/m128		RM	Valid	Convert two packed double-precision floating-point values from <i>xmm/m128</i> to two packed signed doubleword integers in <i>mm</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPD2PI instruction is executed.

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

Operation

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32(SRC[63:0]);
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer32(SRC[127:64]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD1PI: `__m64 _mm_cvtpd_pi32(__m128d a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 22-4, “Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

CVTPD2PS—Convert Packed Double-Precision FP Values to Packed Single-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5A /r CVTPD2PS <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Convert two packed double-precision floating-point values in <i>xmm2/m128</i> to two packed single-precision floating-point values in <i>xmm1</i> .
VEX.128.66.0F.WIG 5A /r VCVTPD2PS <i>xmm1, xmm2/m128</i>		RM V/V	AVX	Convert two packed double-precision floating-point values in <i>xmm2/mem</i> to two single-precision floating-point values in <i>xmm1</i> .
VEX.256.66.0F.WIG 5A /r VCVTPD2PS <i>xmm1, ymm2/m256</i>		RM V/V	AVX	Convert four packed double-precision floating-point values in <i>ymm2/mem</i> to four single-precision floating-point values in <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed single-precision floating-point values in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:64) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

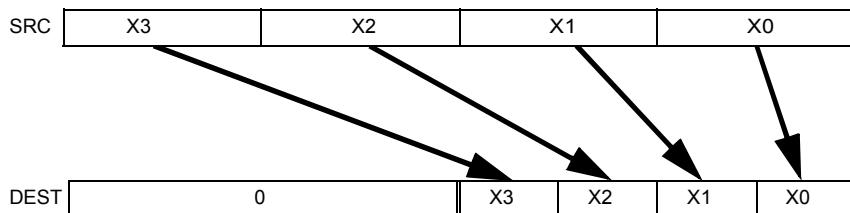


Figure 3-12. VCVTPD2PS (VEX.256 encoded version)

Operation

CVTPD2PS (128-bit Legacy SSE version)

DEST[31:0] \leftarrow Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
 DEST[63:32] \leftarrow Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
 DEST[127:64] \leftarrow 0
 DEST[VLMAX-1:128] (unmodified)

VCVTPD2PS (VEX.128 encoded version)

DEST[31:0] \leftarrow Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
 DEST[63:32] \leftarrow Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
 DEST[VLMAX-1:64] \leftarrow 0

VCVTPD2PS (VEX.256 encoded version)

DEST[31:0] \leftarrow Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
 DEST[63:32] \leftarrow Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
 DEST[95:64] \leftarrow Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
 DEST[127:96] \leftarrow Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192])
 DEST[255:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2PS: `__m128 _mm_cvtpd_ps(__m128d a)`
 CVTPD2PS: `__m256 _mm256_cvtpd_ps (__m256d a)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

CVTPI2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2A /r CVTPI2PD <i>xmm, mm/m64</i> *	RM	Valid	Valid	Convert two packed signed doubleword integers from <i>mm/mem64</i> to two packed double-precision floating-point values in <i>xmm</i> .

NOTES:

*Operation is different for different operand sets; see the Description section.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed double-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. In addition, depending on the operand configuration:

- **For operands *xmm, mm*:** the instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PD instruction is executed.
- **For operands *xmm, m64*:** the instruction does not cause a transition to MMX technology and does not take x87 FPU exceptions.

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

Operation

```
DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PD: `__m128d _mm_cvtpi32_pd(__m64 a)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 22-6, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

CVTPI2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 2A /r CVTPI2PS <i>xmm, mm/m64</i>	RM	Valid	Valid	Convert two signed doubleword integers from <i>mm/m64</i> to two single-precision floating-point values in <i>xmm</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed single-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. The results are stored in the low quadword of the destination operand, and the high quadword remains unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PS instruction is executed.

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

Operation

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32]);
(* High quadword of destination unchanged *)
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PS: `_m128 _mm_cvtpi32_ps(_m128 a, _m64 b)`

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

CVTPS2DQ—Convert Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5B /r CVTPS2DQ <i>xmm1</i> , <i>xmm2/m128</i>		RM	V/V	SSE2 Convert four packed single-precision floating-point values from <i>xmm2/m128</i> to four packed signed doubleword integers in <i>xmm1</i> .
VEX.128.66.0F.WIG 5B /r VCVTPS2DQ <i>xmm1</i> , <i>xmm2/m128</i>		RM	V/V	AVX Convert four packed single precision floating-point values from <i>xmm2/mem</i> to four packed signed doubleword values in <i>xmm1</i> .
VEX.256.66.0F.WIG 5B /r VCVTPS2DQ <i>ymm1</i> , <i>ymm2/m256</i>		RM	V/V	AVX Convert eight packed single precision floating-point values from <i>ymm2/mem</i> to eight packed signed doubleword values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts four or eight packed single-precision floating-point values in the source operand to four or eight signed doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation**CVTPS2DQ (128-bit Legacy SSE version)**

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[VLMAX-1:128] (unmodified)
```

VCVTPS2DQ (VEX.128 encoded version)

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[VLMAX-1:128] ← 0
```

VCVTPS2DQ (VEX.256 encoded version)

DEST[31:0] \leftarrow Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
 DEST[63:32] \leftarrow Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
 DEST[95:64] \leftarrow Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
 DEST[127:96] \leftarrow Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
 DEST[159:128] \leftarrow Convert_Single_Precision_Floating_Point_To_Integer(SRC[159:128])
 DEST[191:160] \leftarrow Convert_Single_Precision_Floating_Point_To_Integer(SRC[191:160])
 DEST[223:192] \leftarrow Convert_Single_Precision_Floating_Point_To_Integer(SRC[223:192])
 DEST[255:224] \leftarrow Convert_Single_Precision_Floating_Point_To_Integer(SRC[255:224])

Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2DQ: `__m128i _mm_cvtps_epi32(__m128 a)`
 VCVTPS2DQ: `__m256i _mm256_cvtps_epi32 (__m256 a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

CVTPS2PD—Convert Packed Single-Precision FP Values to Packed Double-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5A /r CVTPS2PD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Convert two packed single-precision floating-point values in <i>xmm2/m64</i> to two packed double-precision floating-point values in <i>xmm1</i> .
VEX.128.0F.WIG 5A /r VCVTPS2PD <i>xmm1, xmm2/m64</i>		RM V/V	AVX	Convert two packed single-precision floating-point values in <i>xmm2/mem</i> to two packed double-precision floating-point values in <i>xmm1</i> .
VEX.256.0F.WIG 5A /r VCVTPS2PD <i>ymm1, xmm2/m128</i>		RM V/V	AVX	Convert four packed single-precision floating-point values in <i>xmm2/mem</i> to four packed double-precision floating-point values in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two or four packed single-precision floating-point values in the source operand (second operand) to two or four packed double-precision floating-point values in the destination operand (first operand).

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

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

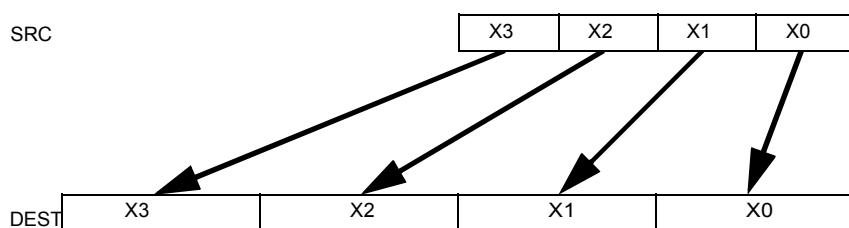


Figure 3-13. CVTPS2PD (VEX.256 encoded version)

Operation

CVTPS2PD (128-bit Legacy SSE version)

DEST[63:0] \leftarrow Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
 DEST[127:64] \leftarrow Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
 DEST[VLMAX-1:128] (unmodified)

VCVTPS2PD (VEX.128 encoded version)

DEST[63:0] \leftarrow Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
 DEST[127:64] \leftarrow Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
 DEST[VLMAX-1:128] \leftarrow 0

VCVTPS2PD (VEX.256 encoded version)

DEST[63:0] \leftarrow Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
 DEST[127:64] \leftarrow Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
 DEST[191:128] \leftarrow Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
 DEST[255:192] \leftarrow Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96])

Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2PD: __m128d _mm_cvtps_pd(__m128 a)
 VCVTPS2PD: __m256d _mm256_cvtps_pd (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UDIf VEX.vvvv != 1111B.

CVTPS2PI—Convert Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 2D /r CVTPS2PI <i>mm, xmm/m64</i>		RM	Valid	Convert two packed single-precision floating-point values from <i>xmm/m64</i> to two packed signed doubleword integers in <i>mm</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

CVTPS2PI causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPS2PI instruction is executed.

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

Operation

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2PI: `__m64 _mm_cvtps_pi32(__m128 a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 22-5, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

CVTSD2SI—Convert Scalar Double-Precision FP Value to Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 2D /r CVTSD2SI r32, xmm/m64	RM	V/V	SSE2	Convert one double-precision floating-point value from <i>xmm/m64</i> to one signed doubleword integer <i>r32</i> .
F2 REX.W OF 2D /r CVTSD2SI r64, xmm/m64	RM	V/N.E.	SSE2	Convert one double-precision floating-point value from <i>xmm/m64</i> to one signed quadword integer sign-extended into <i>r64</i> .
VEX.LIG.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64	RM	V/V	AVX	Convert one double precision floating-point value from <i>xmm1/m64</i> to one signed doubleword integer <i>r32</i> .
VEX.LIG.F2.0F.W1 2D /r VCVTSD2SI r64, xmm1/m64	RM	V/N.E. ¹	AVX	Convert one double precision floating-point value from <i>xmm1/m64</i> to one signed quadword integer sign-extended into <i>r64</i> .

NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a double-precision floating-point value in the source operand (second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

Legacy SSE instructions: Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

IF 64-Bit Mode and OperandSize = 64

THEN

 DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_Integer64(SRC[63:0]);

ELSE

 DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32(SRC[63:0]);

FI;

Intel C/C++ Compiler Intrinsic Equivalent

```
int _mm_cvtsd_si32(__m128d a)  
__int64 _mm_cvtsd_si64(__m128d a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

CVTSD2SS—Convert Scalar Double-Precision FP Value to Scalar Single-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 5A /r CVTSD2SS <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Convert one double-precision floating-point value in <i>xmm2/m64</i> to one single-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 5A /r VCVTSD2SS <i>xmm1,xmm2, xmm3/m64</i>	RVM	V/V	AVX	Convert one double-precision floating-point value in <i>xmm3/m64</i> to one single-precision floating-point value and merge with high bits in <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts a double-precision floating-point value in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand).

The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand, and the upper 3 doublewords are left unchanged. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

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

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation**CVTSD2SS (128-bit Legacy SSE version)**

DEST[31:0] \leftarrow Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[VLMAX-1:32] Unmodified *)

VCVTSD2SS (VEX.128 encoded version)

DEST[31:0] \leftarrow Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] \leftarrow SRC1[127:32]
DEST[VLMAX-1:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

CVTSD2SS: `__m128_mm_cvtsd_ss(__m128 a, __m128d b)`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

CVTSI2SD—Convert Dword Integer to Scalar Double-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 2A /r CVTSI2SD <i>xmm</i> , <i>r/m32</i>	RM	V/V	SSE2	Convert one signed doubleword integer from <i>r/m32</i> to one double-precision floating-point value in <i>xmm</i> .
F2 REX.W OF 2A /r CVTSI2SD <i>xmm</i> , <i>r/m64</i>	RM	V/N.E.	SSE2	Convert one signed quadword integer from <i>r/m64</i> to one double-precision floating-point value in <i>xmm</i> .
VEX.NDS.LIG.F2.OF.W0 2A /r VCVTSI2SD <i>xmm1</i> , <i>xmm2</i> , <i>r/m32</i>	RVM	V/V	AVX	Convert one signed doubleword integer from <i>r/m32</i> to one double-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F2.OF.W1 2A /r VCVTSI2SD <i>xmm1</i> , <i>xmm2</i> , <i>r/m64</i>	RVM	V/N.E. ¹	AVX	Convert one signed quadword integer from <i>r/m64</i> to one double-precision floating-point value in <i>xmm1</i> .

NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the second source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Legacy SSE instructions: Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

CVTSI2SD

IF 64-Bit Mode And OperandSize = 64

THEN

 DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:0]);

ELSE

 DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);

FI;

DEST[VLMAX-1:64] (Unmodified)

VCVTSI2SD

IF 64-Bit Mode And OperandSize = 64
 THEN
 DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
 ELSE
 DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
 F;
 DEST[127:64] ← SRC1[127:64]
 DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

CVTSI2SD: __m128d _mm_cvtsi32_sd(__m128d a, int b)
 CVTSI2SD: __m128d _mm_cvtsi64_sd(__m128d a, __int64 b)

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Exceptions Type 3.

CVTSI2SS—Convert Dword Integer to Scalar Single-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 2A /r CVTSI2SS <i>xmm, r/m32</i>		RM V/V	SSE	Convert one signed doubleword integer from <i>r/m32</i> to one single-precision floating-point value in <i>xmm</i> .
F3 REX.W OF 2A /r CVTSI2SS <i>xmm, r/m64</i>		RM V/N.E.	SSE	Convert one signed quadword integer from <i>r/m64</i> to one single-precision floating-point value in <i>xmm</i> .
VEX.NDS.LIG.F3.OF.W0 2A /r VCVTSI2SS <i>xmm1, xmm2, r/m32</i>	RVM	V/V	AVX	Convert one signed doubleword integer from <i>r/m32</i> to one single-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F3.OF.W1 2A /r VCVTSI2SS <i>xmm1, xmm2, r/m64</i>	RVM	V/N.E. ¹	AVX	Convert one signed quadword integer from <i>r/m64</i> to one single-precision floating-point value in <i>xmm1</i> .

NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Legacy SSE instructions: In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation**CVTSI2SS (128-bit Legacy SSE version)**

IF 64-Bit Mode And OperandSize = 64

THEN

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
```

ELSE

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
```

Fl;

DEST[VLMAX-1:32] (Unmodified)

VCVTSI2SS (VEX.128 encoded version)

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);

ELSE

DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);

FI;

DEST[127:32] ← SRC1[127:32]

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

CVTSI2SS: __m128 _mm_cvtsi32_ss(__m128 a, int b)

CVTSI2SS: __m128 _mm_cvtsi64_ss(__m128 a, __int64 b)

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Exceptions Type 3.

CVTSS2SD—Convert Scalar Single-Precision FP Value to Scalar Double-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 5A /r CVTSS2SD <i>xmm1, xmm2/m32</i>		RM V/V	SSE2	Convert one single-precision floating-point value in <i>xmm2/m32</i> to one double-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F3.OF.WIG 5A /r VCVTSS2SD <i>xmm1, xmm2, xmm3/m32</i>	RVM	V/V	AVX	Convert one single-precision floating-point value in <i>xmm3/m32</i> to one double-precision floating-point value and merge with high bits of <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts a single-precision floating-point value in the source operand (second operand) to a double-precision floating-point value in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand, and the high quadword is left unchanged.

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

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

CVTSS2SD (128-bit Legacy SSE version)

```
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[VLMAX-1:64] (Unmodified)
```

VCVTSS2SD (VEX.128 encoded version)

```
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0])
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTSS2SD: `__m128d _mm_cvtss_sd(__m128d a, __m128 b)`

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Exceptions Type 3.

CVTSS2SI—Convert Scalar Single-Precision FP Value to Dword Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 2D /r CVTSS2SI r32, xmm/m32	RM	V/V	SSE	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed doubleword integer in <i>r32</i> .
F3 REX.W OF 2D /r CVTSS2SI r64, xmm/m32	RM	V/N.E.	SSE	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed quadword integer in <i>r64</i> .
VEX.LIG.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32	RM	V/V	AVX	Convert one single-precision floating-point value from <i>xmm1/m32</i> to one signed doubleword integer in <i>r32</i> .
VEX.LIG.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32	RM	V/N.E. ¹	AVX	Convert one single-precision floating-point value from <i>xmm1/m32</i> to one signed quadword integer in <i>r64</i> .

NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

IF 64-bit Mode and OperandSize = 64

THEN

 DEST[64:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);

ELSE

 DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);

FI;

Intel C/C++ Compiler Intrinsic Equivalent

```
int _mm_cvtss_si32(__m128d a)
__int64 _mm_cvtss_si64(__m128d a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

CVTTPD2DQ—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F E6 /r CVTTPD2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Convert two packed double-precision floating-point values from <i>xmm2/m128</i> to two packed signed doubleword integers in <i>xmm1</i> using truncation.
VEX.128.66.0F.WIG E6 /r VCVTTPD2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Convert two packed double-precision floating-point values in <i>xmm2/mem</i> to two signed doubleword integers in <i>xmm1</i> using truncation.
VEX.256.66.0F.WIG E6 /r VCVTTPD2DQ <i>xmm1, ymm2/m256</i>	RM	V/V	AVX	Convert four packed double-precision floating-point values in <i>ymm2/mem</i> to four signed doubleword integers in <i>xmm1</i> using truncation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two or four packed double-precision floating-point values in the source operand (second operand) to two or four packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is an XMM register. The upper bits (255:128) of the corresponding YMM register destination are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

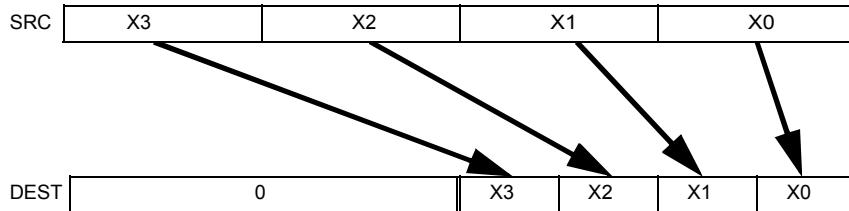


Figure 3-14. VCVTPD2DQ (VEX.256 encoded version)

Operation

CVTTPD2DQ (128-bit Legacy SSE version)

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[127:64] ← 0
DEST[VLMAX-1:128] (unmodified)
```

VCVTPD2DQ (VEX.128 encoded version)

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[VLMAX-1:64] ← 0
```

VCVTPD2DQ (VEX.256 encoded version)

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192])
DEST[255:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
CVTTPD2DQ: __m128i _mm_cvtpd_epi32(__m128d a)
VCVTPD2DQ: __m128i _mm256_cvtpd_epi32 (__m256d src)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

CVTPD2PI—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2C /r CVTPD2PI <i>mm, xmm/m128</i>		RM	Valid	Convert two packed double-precision floating-point values from <i>xmm/m128</i> to two packed signed doubleword integers in <i>mm</i> using truncation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPD2PI instruction is executed.

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

Operation

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32_Truncate(SRC[63:0]);
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer32_
Truncate(SRC[127:64]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD1PI: __m64 _mm_cvtpd_pi32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Mode Exceptions

See Table 22-4, "Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

CVTTPS2DQ—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 5B /r CVTTPS2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Convert four single-precision floating-point values from <i>xmm2/m128</i> to four signed doubleword integers in <i>xmm1</i> using truncation.
VEX.128.F3.0F.WIG 5B /r VCVTTPS2DQ <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Convert four packed single precision floating-point values from <i>xmm2/mem</i> to four packed signed doubleword values in <i>xmm1</i> using truncation.
VEX.256.F3.0F.WIG 5B /r VCVTTPS2DQ <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Convert eight packed single precision floating-point values from <i>ymm2/mem</i> to eight packed signed doubleword values in <i>ymm1</i> using truncation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts four or eight packed single-precision floating-point values in the source operand to four or eight signed doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operation is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

CVTTPS2DQ (128-bit Legacy SSE version)

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[VLMAX-1:128] (unmodified)
```

VCVTPPS2DQ (VEX.128 encoded version)

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[VLMAX-1:128] ← 0
```

VCVTPPS2DQ (VEX.256 encoded version)

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[159:128] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[159:128])
DEST[191:160] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[191:160])
DEST[223:192] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[223:192])
DEST[255:224] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[255:224])
```

Intel C/C++ Compiler Intrinsic Equivalent

`CVTPPS2DQ: __m128i _mm_cvttps_epi32(__m128 a)`
`VCVTPPS2DQ: __m256i _mm256_cvttps_epi32 (__m256 a)`

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.vvvv != 1111B.

CVTTPS2PI—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 2C /r CVTTPS2PI <i>mm, xmm/m64</i>		RM	Valid	Convert two single-precision floating-point values from <i>xmm/m64</i> to two signed doubleword signed integers in <i>mm</i> using truncation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPS2PI instruction is executed.

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

Operation

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2PI: __m64 _mm_cvttps_pi32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

CVTTSD2SI—Convert with Truncation Scalar Double-Precision FP Value to Signed Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 2C /r CVTTSD2SI r32, xmm/m64	RM	V/V	SSE2	Convert one double-precision floating-point value from <i>xmm/m64</i> to one signed doubleword integer in <i>r32</i> using truncation.
F2 REX.W OF 2C /r CVTTSD2SI r64, xmm/m64	RM	V/N.E.	SSE2	Convert one double precision floating-point value from <i>xmm/m64</i> to one signedquadword integer in <i>r64</i> using truncation.
VEX.LIG.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64	RM	V/V	AVX	Convert one double-precision floating-point value from <i>xmm1/m64</i> to one signed doubleword integer in <i>r32</i> using truncation.
VEX.LIG.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64	RM	V/N.E. ¹	AVX	Convert one double precision floating-point value from <i>xmm1/m64</i> to one signed quadword integer in <i>r64</i> using truncation.

NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a double-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

IF 64-Bit Mode and OperandSize = 64

THEN

```
DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_
    Integer64_Truncate(SRC[63:0]);
```

ELSE

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_
    Integer32_Truncate(SRC[63:0]);
```

FI;

Intel C/C++ Compiler Intrinsic Equivalent

```
int _mm_cvtsd_si32(__m128d a)
__int64 _mm_cvtsd_si64(__m128d a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

CVTTSS2SI—Convert with Truncation Scalar Single-Precision FP Value to Dword Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 2C /r CVTTSS2SI r32, xmm/m32	RM	V/V	SSE	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed doubleword integer in <i>r32</i> using truncation.
F3 REX.W OF 2C /r CVTTSS2SI r64, xmm/m32	RM	V/N.E.	SSE	Convert one single-precision floating-point value from <i>xmm/m32</i> to one signed quadword integer in <i>r64</i> using truncation.
VEX.LIG.F3.OF.W0 2C /r VCVTTSS2SI r32, xmm1/m32	RM	V/V	AVX	Convert one single-precision floating-point value from <i>xmm1/m32</i> to one signed doubleword integer in <i>r32</i> using truncation.
VEX.LIG.F3.OF.W1 2C /r VCVTTSS2SI r64, xmm1/m32	RM	V/N.E. ¹	AVX	Convert one single-precision floating-point value from <i>xmm1/m32</i> to one signed quadword integer in <i>r64</i> using truncation.

NOTES:

1. Encoding the VEX prefix with VEX.W=1 in non-64-bit mode is ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H) is returned.

Legacy SSE instructions: In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

IF 64-Bit Mode and OperandSize = 64

THEN

```
DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_
Integer_Truncate(SRC[31:0]);
```

ELSE

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_
Integer_Truncate(SRC[31:0]);
```

FI;

Intel C/C++ Compiler Intrinsic Equivalent

```
int _mm_cvttss_si32(_m128d a)
__int64 _mm_cvttss_si64(_m128d a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
99	CWD		NP	Valid	DX:AX ← sign-extend of AX.
99	CDQ		NP	Valid	EDX:EAX ← sign-extend of EAX.
REX.W + 99	CQO		NP	Valid	N.E. RDX:RAX ← sign-extend of RAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Doubles the size of the operand in register AX, EAX, or RAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX, EDX:EAX, or RDX:RAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register. The CQO instruction (available in 64-bit mode only) copies the sign (bit 63) of the value in the RAX register into every bit position in the RDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before word division. The CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division. The CQO instruction can be used to produce a double quadword dividend from a quadword before a quadword division.

The CWD and CDQ mnemonics reference the same opcode. The CWD instruction is intended for use when the operand-size attribute is 16 and the CDQ instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CWD is used and to 32 when CDQ is used. Others may treat these mnemonics as synonyms (CWD/CDQ) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. The CQO mnemonics reference the same opcode as CWD/CDQ. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```

IF OperandSize = 16 (* CWD instruction *)
  THEN
    DX ← SignExtend(AX);
  ELSE IF OperandSize = 32 (* CDQ instruction *)
    EDX ← SignExtend(EAX); Fl;
  ELSE IF 64-Bit Mode and OperandSize = 64 (* CQO instruction*)
    RDX ← SignExtend(RAX); Fl;
Fl;

```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

DAA—Decimal Adjust AL after Addition

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
27	DAA	NP	Invalid	Valid	Decimal adjust AL after addition.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Adjusts the sum of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two 2-digit, packed BCD values and stores a byte result in the AL register. The DAA instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal carry is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

IF 64-Bit Mode

```

THEN
#UD;
ELSE
old_AL ← AL;
old_CF ← CF;
CF ← 0;
IF (((AL AND 0FH) > 9) or AF = 1)
THEN
    AL ← AL + 6;
    CF ← old_CF or (Carry from AL ← AL + 6);
    AF ← 1;
ELSE
    AF ← 0;
FI;
IF ((old_AL > 99H) or (old_CF = 1))
THEN
    AL ← AL + 60H;
    CF ← 1;
ELSE
    CF ← 0;
FI;
FI;
```

Example

ADD	AL, BL	Before: AL=79H BL=35H EFLAGS(OSZAPC)=XXXXXX After: AL=AEH BL=35H EFLAGS(OSZAPC)=110000
DAA		Before: AL=AEH BL=35H EFLAGS(OSZAPC)=110000 After: AL=14H BL=35H EFLAGS(OSZAPC)=X00111
DAA		Before: AL=2EH BL=35H EFLAGS(OSZAPC)=110000 After: AL=34H BL=35H EFLAGS(OSZAPC)=X00101

Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal carry in either digit of the result (see the “Operation” section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

DAS—Decimal Adjust AL after Subtraction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
2F	DAS	NP	Invalid	Valid	Decimal adjust AL after subtraction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Adjusts the result of the subtraction of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one 2-digit, packed BCD value from another and stores a byte result in the AL register. The DAS instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal borrow is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

IF 64-Bit Mode

```

THEN
#UD;
ELSE
old_AL ← AL;
old_CF ← CF;
CF ← 0;
IF (((AL AND 0FH) > 9) or AF = 1)
THEN
    AL ← AL - 6;
    CF ← old_CF or (Borrow from AL ← AL - 6);
    AF ← 1;
ELSE
    AF ← 0;
FI;
IF ((old_AL > 99H) or (old_CF = 1))
THEN
    AL ← AL - 60H;
    CF ← 1;
FI;
FI;
```

Example

SUB	AL, BL	Before: AL = 35H, BL = 47H, EFLAGS(OSZAPC) = XXXXXX
		After: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
DAA		Before: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
		After: AL = 88H, BL = 47H, EFLAGS(OSZAPC) = X10111

Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal borrow in either digit of the result (see the “Operation” section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

Protected Mode Exceptions

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD If in 64-bit mode.

DEC—Decrement by 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FE /1	DEC r/m8	M	Valid	Valid	Decrement r/m8 by 1.
REX + FE /1	DEC r/m8*	M	Valid	N.E.	Decrement r/m8 by 1.
FF /1	DEC r/m16	M	Valid	Valid	Decrement r/m16 by 1.
FF /1	DEC r/m32	M	Valid	Valid	Decrement r/m32 by 1.
REX.W + FF /1	DEC r/m64	M	Valid	N.E.	Decrement r/m64 by 1.
48+rw	DEC r16	O	N.E.	Valid	Decrement r16 by 1.
48+rd	DEC r32	O	N.E.	Valid	Decrement r32 by 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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
O	opcode + rd (r, w)	NA	NA	NA

Description

Subtracts 1 from the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.)

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

In 64-bit mode, DEC r16 and DEC r32 are not encodable (because opcodes 48H through 4FH are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← DEST - 1;

Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination 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.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

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.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

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.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

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.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

DIV—Unsigned Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /6	DIV r/m8	M	Valid	Valid	Unsigned divide AX by r/m8, with result stored in AL ← Quotient, AH ← Remainder.
REX + F6 /6	DIV r/m8*	M	Valid	N.E.	Unsigned divide AX by r/m8, with result stored in AL ← Quotient, AH ← Remainder.
F7 /6	DIV r/m16	M	Valid	Valid	Unsigned divide DX:AX by r/m16, with result stored in AX ← Quotient, DX ← Remainder.
F7 /6	DIV r/m32	M	Valid	Valid	Unsigned divide EDX:EAX by r/m32, with result stored in EAX ← Quotient, EDX ← Remainder.
REX.W + F7 /6	DIV r/m64	M	Valid	N.E.	Unsigned divide RDX:RAX by r/m64, with result stored in RAX ← Quotient, RDX ← Remainder.

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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). Division using 64-bit operand is available only in 64-bit mode.

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the unsigned value in RDX:RAX by the source operand and stores the quotient in RAX, the remainder in RDX.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-25.

Table 3-25. DIV Action

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	r/m8	AL	AH	255
Doubleword/word	DX:AX	r/m16	AX	DX	65,535
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	$2^{64} - 1$

Operation

```

IF SRC = 0
    THEN #DE; Fl; (* Divide Error *)
IF OperandSize = 8 (* Word/Byte Operation *)
    THEN
        temp ← AX / SRC;
        IF temp > FFH
            THEN #DE; (* Divide error *)
        ELSE
            AL ← temp;
            AH ← AX MOD SRC;
        Fl;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp ← DX:AX / SRC;
            IF temp > FFFFH
                THEN #DE; (* Divide error *)
            ELSE
                AX ← temp;
                DX ← DX:AX MOD SRC;
            Fl;
        Fl;
    ELSE IF Operandsize = 32 (* Quadword/doubleword operation *)
        THEN
            temp ← EDX:EAX / SRC;
            IF temp > FFFFFFFFH
                THEN #DE; (* Divide error *)
            ELSE
                EAX ← temp;
                EDX ← EDX:EAX MOD SRC;
            Fl;
        Fl;
    ELSE IF 64-Bit Mode and Operandsize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp ← RDX:RAX / SRC;
            IF temp > FFFFFFFFFFFFFF
                THEN #DE; (* Divide error *)
            ELSE
                RAX ← temp;
                RDX ← RDX:RAX MOD SRC;
            Fl;
        Fl;
    Fl;

```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#DE	If the source operand (divisor) is 0 If the quotient is too large for the designated register.
#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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#DE	If the source operand (divisor) is 0. If the quotient is too large for the designated register.
#GP	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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#DE	If the source operand (divisor) is 0. If the quotient is too large for the designated register.
#GP(0)	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.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#DE	If the source operand (divisor) is 0 If the quotient is too large for the designated register.
#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.
#UD	If the LOCK prefix is used.

DIVPD—Divide Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5E /r DIVPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Divide packed double-precision floating-point values in <i>xmm1</i> by packed double-precision floating-point values <i>xmm2/m128</i> .
VEX.NDS.128.66.0F.WIG 5E /r VDIVPD <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Divide packed double-precision floating-point values in <i>xmm2</i> by packed double-precision floating-point values in <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 5E /r VDIVPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Divide packed double-precision floating-point values in <i>ymm2</i> by packed double-precision floating-point values in <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD divide of the two or four packed double-precision floating-point values in the first source operand by the two or four packed double-precision floating-point values in the second source operand. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of a SIMD double-precision floating-point operation.

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation**DIVPD (128-bit Legacy SSE version)**

DEST[63:0] \leftarrow SRC1[63:0] / SRC2[63:0]
 DEST[127:64] \leftarrow SRC1[127:64] / SRC2[127:64]
 DEST[VLMAX-1:128] (Unmodified)

VDIVPD (VEX.128 encoded version)

DEST[63:0] \leftarrow SRC1[63:0] / SRC2[63:0]
 DEST[127:64] \leftarrow SRC1[127:64] / SRC2[127:64]
 DEST[VLMAX-1:128] \leftarrow 0

VDIVPD (VEX.256 encoded version)

DEST[63:0] \leftarrow SRC1[63:0] / SRC2[63:0]
DEST[127:64] \leftarrow SRC1[127:64] / SRC2[127:64]
DEST[191:128] \leftarrow SRC1[191:128] / SRC2[191:128]
DEST[255:192] \leftarrow SRC1[255:192] / SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

DIVPD: `__m128d _mm_div_pd(__m128d a, __m128d b)`
VDIVPD: `__m256d _mm256_div_pd (__m256d a, __m256d b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

DIVPS—Divide Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5E /r DIVPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Divide packed single-precision floating-point values in <i>xmm1</i> by packed single-precision floating-point values <i>xmm2/m128</i> .
VEX.NDS.128.0F.WIG 5E /r VDIVPS <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Divide packed single-precision floating-point values in <i>xmm2</i> by packed double-precision floating-point values in <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG 5E /r VDIVPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Divide packed single-precision floating-point values in <i>ymm2</i> by packed double-precision floating-point values in <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD divide of the four or eight packed single-precision floating-point values in the first source operand by the four or eight packed single-precision floating-point values in the second source operand. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of a SIMD single-precision floating-point operation.

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation**DIVPS (128-bit Legacy SSE version)**

```
DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[63:32] ← SRC1[63:32] / SRC2[63:32]
DEST[95:64] ← SRC1[95:64] / SRC2[95:64]
DEST[127:96] ← SRC1[127:96] / SRC2[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

VDIVPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[63:32] ← SRC1[63:32] / SRC2[63:32]
DEST[95:64] ← SRC1[95:64] / SRC2[95:64]
DEST[127:96] ← SRC1[127:96] / SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

VDIVPS (VEX.256 encoded version)

DEST[31:0] \leftarrow SRC1[31:0] / SRC2[31:0]
 DEST[63:32] \leftarrow SRC1[63:32] / SRC2[63:32]
 DEST[95:64] \leftarrow SRC1[95:64] / SRC2[95:64]
 DEST[127:96] \leftarrow SRC1[127:96] / SRC2[127:96]
 DEST[159:128] \leftarrow SRC1[159:128] / SRC2[159:128]
 DEST[191:160] \leftarrow SRC1[191:160] / SRC2[191:160]
 DEST[223:192] \leftarrow SRC1[223:192] / SRC2[223:192]
 DEST[255:224] \leftarrow SRC1[255:224] / SRC2[255:224].

Intel C/C++ Compiler Intrinsic Equivalent

DIVPS: `__m128 _mm_div_ps(__m128 a, __m128 b)`
 VDIVPS: `__m256 _mm256_div_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

DIVSD—Divide Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 5E /r DIVSD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Divide low double-precision floating-point value in <i>xmm1</i> by low double-precision floating-point value in <i>xmm2/mem64</i> .
VEX.NDS.LIG.F2.OF.WIG 5E /r VDIVSD <i>xmm1, xmm2, xmm3/m64</i>	RVM	V/V	AVX	Divide low double-precision floating point values in <i>xmm2</i> by low double precision floating-point value in <i>xmm3/mem64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Divides the low double-precision floating-point value in the first source operand by the low double-precision floating-point value in the second source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination hyperons are XMM registers. The high quadword of the destination operand is copied from the high quadword of the first source operand. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of a scalar double-precision floating-point operation.

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

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation**DIVSD (128-bit Legacy SSE version)**

DEST[63:0] \leftarrow *DEST*[63:0] / *SRC*[63:0]
DEST[VLMAX-1:64] (Unmodified)

VDIVSD (VEX.128 encoded version)

DEST[63:0] \leftarrow *SRC1*[63:0] / *SRC2*[63:0]
DEST[127:64] \leftarrow *SRC1*[127:64]
DEST[VLMAX-1:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

DIVSD: __m128d _mm_div_sd (m128d a, m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

DIVSS—Divide Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 5E /r DIVSS <i>xmm1, xmm2/m32</i>		RM V/V	SSE	Divide low single-precision floating-point value in <i>xmm1</i> by low single-precision floating-point value in <i>xmm2/m32</i> .
VEX.NDS.LIG.F3.0F.WIG 5E /r VDIVSS <i>xmm1, xmm2, xmm3/m32</i>		RVM V/V	AVX	Divide low single-precision floating point value in <i>xmm2</i> by low single precision floating-point value in <i>xmm3/m32</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Divides the low single-precision floating-point value in the first source operand by the low single-precision floating-point value in the second source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. The three high-order doublewords of the destination are copied from the same dwords of the first source operand. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an overview of a scalar single-precision floating-point operation.

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

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation**DIVSS (128-bit Legacy SSE version)**

DEST[31:0] \leftarrow DEST[31:0] / SRC[31:0]
DEST[VLMAX-1:32] (Unmodified)

VDIVSS (VEX.128 encoded version)

DEST[31:0] \leftarrow SRC1[31:0] / SRC2[31:0]
DEST[127:32] \leftarrow SRC1[127:32]
DEST[VLMAX-1:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

DIVSS: __m128 _mm_div_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

See Exceptions Type 3.

DPPD – Dot Product of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 41 /r ib DPPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Selectively multiply packed DP floating-point values from <i>xmm1</i> with packed DP floating-point values from <i>xmm2</i> , add and selectively store the packed DP floating-point values to <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 41 /r ib VDPPD <i>xmm1,xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Selectively multiply packed DP floating-point values from <i>xmm2</i> with packed DP floating-point values from <i>xmm3</i> , add and selectively store the packed DP floating-point values to <i>xmm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Conditionally multiplies the packed double-precision floating-point values in the destination operand (first operand) with the packed double-precision floating-point values in the source (second operand) depending on a mask extracted from bits [5:4] of the immediate operand (third operand). If a condition mask bit is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

The two resulting double-precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [1:0] of the immediate byte.

If a broadcast mask bit is "1", the intermediate result is copied to the corresponding qword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

DPPD follows the NaN forwarding rules stated in the Software Developer’s Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

If VDPPD is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

DP_primitive (SRC1, SRC2)

```

IF (imm8[4] = 1)
    THEN Temp1[63:0] ← DEST[63:0] * SRC[63:0]; // update SIMD exception flags
    ELSE Temp1[63:0] ← +0.0; Fl;
IF (imm8[5] = 1)
    THEN Temp1[127:64] ← DEST[127:64] * SRC[127:64]; // update SIMD exception flags
    ELSE Temp1[127:64] ← +0.0; Fl;
/* if unmasked exception reported, execute exception handler*/

```

```

Temp2[63:0] ← Temp1[63:0] + Temp1[127:64]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/

```

```

IF (imm8[0] = 1)
    THEN DEST[63:0] ← Temp2[63:0];
    ELSE DEST[63:0] ← +0.0; Fl;
IF (imm8[1] = 1)
    THEN DEST[127:64] ← Temp2[63:0];
    ELSE DEST[127:64] ← +0.0; Fl;

```

DPPD (128-bit Legacy SSE version)

```

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-1:128] (Unmodified)

```

VDPPD (VEX.128 encoded version)

```

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-1:128] ← 0

```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

DPPD: `__m128d _mm_dp_pd (__m128d a, __m128d b, const int mask);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Exceptions are determined separately for each add and multiply operation. Unmasked exceptions will leave the destination untouched.

Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.L= 1.

DPPS – Dot Product of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 40 /r ib DPPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Selectively multiply packed SP floating-point values from <i>xmm1</i> with packed SP floating-point values from <i>xmm2</i> , add and selectively store the packed SP floating-point values or zero values to <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 40 /r ib VDPPS <i>xmm1,xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Multiply packed SP floating point values from <i>xmm1</i> with packed SP floating point values from <i>xmm2/mem</i> selectively add and store to <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG 40 /r ib VDPPS <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Multiply packed single-precision floating-point values from <i>ymm2</i> with packed SP floating point values from <i>ymm3/mem</i> , selectively add pairs of elements and store to <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Conditionally multiplies the packed single precision floating-point values in the destination operand (first operand) with the packed single-precision floats in the source (second operand) depending on a mask extracted from the high 4 bits of the immediate byte (third operand). If a condition mask bit in Imm8[7:4] is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

The four resulting single-precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [3:0] of the immediate byte.

If a broadcast mask bit is "1", the intermediate result is copied to the corresponding dword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

DPPS follows the NaN forwarding rules stated in the Software Developer’s Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

```

DP_primitive (SRC1, SRC2)
IF (imm8[4] = 1)
    THEN Temp1[31:0] ← DEST[31:0] * SRC[31:0]; // update SIMD exception flags
    ELSE Temp1[31:0] ← +0.0; Fl;
IF (imm8[5] = 1)
    THEN Temp1[63:32] ← DEST[63:32] * SRC[63:32]; // update SIMD exception flags
    ELSE Temp1[63:32] ← +0.0; Fl;
IF (imm8[6] = 1)
    THEN Temp1[95:64] ← DEST[95:64] * SRC[95:64]; // update SIMD exception flags
    ELSE Temp1[95:64] ← +0.0; Fl;
IF (imm8[7] = 1)
    THEN Temp1[127:96] ← DEST[127:96] * SRC[127:96]; // update SIMD exception flags
    ELSE Temp1[127:96] ← +0.0; Fl;

Temp2[31:0] ← Temp1[31:0] + Temp1[63:32]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/
Temp3[31:0] ← Temp1[95:64] + Temp1[127:96]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/
Temp4[31:0] ← Temp2[31:0] + Temp3[31:0]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/

IF (imm8[0] = 1)
    THEN DEST[31:0] ← Temp4[31:0];
    ELSE DEST[31:0] ← +0.0; Fl;
IF (imm8[1] = 1)
    THEN DEST[63:32] ← Temp4[31:0];
    ELSE DEST[63:32] ← +0.0; Fl;
IF (imm8[2] = 1)
    THEN DEST[95:64] ← Temp4[31:0];
    ELSE DEST[95:64] ← +0.0; Fl;
IF (imm8[3] = 1)
    THEN DEST[127:96] ← Temp4[31:0];
    ELSE DEST[127:96] ← +0.0; Fl;

```

DPPS (128-bit Legacy SSE version)

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
 DEST[VLMAX-1:128] (Unmodified)

VDPPS (VEX.128 encoded version)

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
 DEST[VLMAX-1:128] ← 0

VDPPS (VEX.256 encoded version)

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
 DEST[255:128] ← DP_Primitive(SRC1[255:128], SRC2[255:128]);

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

(V)DPPS: __m128 _mm_dp_ps (__m128 a, __m128 b, const int mask);

VDPPS: __m256 _mm256_dp_ps (__m256 a, __m256 b, const int mask);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Exceptions are determined separately for each add and multiply operation, in the order of their execution.
Unmasked exceptions will leave the destination operands unchanged.

Other Exceptions

See Exceptions Type 2.

EMMS—Empty MMX Technology State

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 77	EMMS	NP	Valid	Valid	Set the x87 FPU tag word to empty.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Sets the values of all the tags in the x87 FPU tag word to empty (all 1s). This operation marks the x87 FPU data registers (which are aliased to the MMX technology registers) as available for use by x87 floating-point instructions. (See Figure 8-7 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for the format of the x87 FPU tag word.) All other MMX instructions (other than the EMMS instruction) set all the tags in x87 FPU tag word to valid (all 0s).

The EMMS instruction must be used to clear the MMX technology state at the end of all MMX technology procedures or subroutines and before calling other procedures or subroutines that may execute x87 floating-point instructions. If a floating-point instruction loads one of the registers in the x87 FPU data register stack before the x87 FPU tag word has been reset by the EMMS instruction, an x87 floating-point register stack overflow can occur that will result in an x87 floating-point exception or incorrect result.

EMMS operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
x87FPUTagWord ← FFFFH;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_empty()
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If CR0.EM[bit 2] = 1.
#NM	If CR0.TS[bit 3] = 1.
#MF	If there is a pending FPU exception.
#UD	If the LOCK prefix is used.

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.

ENTER—Make Stack Frame for Procedure Parameters

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C8 <i>iw</i> 00	ENTER <i>imm16</i> , 0	II	Valid	Valid	Create a stack frame for a procedure.
C8 <i>iw</i> 01	ENTER <i>imm16</i> , 1	II	Valid	Valid	Create a nested stack frame for a procedure.
C8 <i>iw</i> ib	ENTER <i>imm16</i> , <i>imm8</i>	II	Valid	Valid	Create a nested stack frame for a procedure.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
II	<i>iw</i>	<i>imm8</i>	NA	NA

Description

Creates a stack frame for a procedure. The first operand (size operand) specifies the size of the stack frame (that is, the number of bytes of dynamic storage allocated on the stack for the procedure). The second operand (nesting level operand) gives the lexical nesting level (0 to 31) of the procedure. The nesting level determines the number of stack frame pointers that are copied into the “display area” of the new stack frame from the preceding frame. Both of these operands are immediate values.

The stack-size attribute determines whether the BP (16 bits), EBP (32 bits), or RBP (64 bits) register specifies the current frame pointer and whether SP (16 bits), ESP (32 bits), or RSP (64 bits) specifies the stack pointer. In 64-bit mode, stack-size attribute is always 64-bits.

The ENTER and companion LEAVE instructions are provided to support block structured languages. The ENTER instruction (when used) is typically the first instruction in a procedure and is used to set up a new stack frame for a procedure. The LEAVE instruction is then used at the end of the procedure (just before the RET instruction) to release the stack frame.

If the nesting level is 0, the processor pushes the frame pointer from the BP/EBP/RBP register onto the stack, copies the current stack pointer from the SP/ESP/RSP register into the BP/EBP/RBP register, and loads the SP/ESP/RSP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack. See “Procedure Calls for Block-Structured Languages” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about the actions of the ENTER instruction.

The ENTER instruction causes a page fault whenever a write using the final value of the stack pointer (within the current stack segment) would do so.

In 64-bit mode, default operation size is 64 bits; 32-bit operation size cannot be encoded.

Operation

NestingLevel \leftarrow NestingLevel MOD 32

IF 64-Bit Mode (StackSize = 64)

THEN

Push(RBP);

FrameTemp \leftarrow RSP;

ELSE IF StackSize = 32

THEN

Push(EBP);

FrameTemp \leftarrow ESP; Fl;

ELSE (* StackSize = 16 *)

Push(BP);

FrameTemp \leftarrow SP;

Fl;

IF NestingLevel = 0

INSTRUCTION SET REFERENCE, A-M

```
THEN GOTO CONTINUE;
Fl;

IF (NestingLevel > 1)
    THEN FOR i ← 1 to (NestingLevel - 1)
        DO
            IF 64-Bit Mode (StackSize = 64)
                THEN
                    RBP ← RBP - 8;
                    Push([RBP]); (* Quadword push *)
                ELSE IF OperandSize = 32
                    THEN
                        IF StackSize = 32
                            EBP ← EBP - 4;
                            Push([EBP]); (* Doubleword push *)
                        ELSE (* StackSize = 16 *)
                            BP ← BP - 4;
                            Push([BP]); (* Doubleword push *)
                        Fl;
                    Fl;
                ELSE (* OperandSize = 16 *)
                    IF StackSize = 32
                        THEN
                            EBP ← EBP - 2;
                            Push([EBP]); (* Word push *)
                        ELSE (* StackSize = 16 *)
                            BP ← BP - 2;
                            Push([BP]); (* Word push *)
                        Fl;
                    Fl;
                OD;
            Fl;

IF 64-Bit Mode (StackSize = 64)
    THEN
        Push(FrameTemp); (* Quadword push *)
    ELSE IF OperandSize = 32
        THEN
            Push(FrameTemp); Fl; (* Doubleword push *)
    ELSE (* OperandSize = 16 *)
        Push(FrameTemp); (* Word push *)
Fl;

CONTINUE:
IF 64-Bit Mode (StackSize = 64)
    THEN
        RBP ← FrameTemp;
        RSP ← RSP - Size;
    ELSE IF StackSize = 32
        THEN
            EBP ← FrameTemp;
            ESP ← ESP - Size; Fl;
    ELSE (* StackSize = 16 *)
        BP ← FrameTemp;
        SP ← SP - Size;
```

```
Fl;  
END;
```

Flags Affected

None.

Protected Mode Exceptions

- #SS(0) If the new value of the SP or ESP register is outside the stack segment limit.
- #PF(fault-code) If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #SS If the new value of the SP or ESP register is outside the stack segment limit.
- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #SS(0) If the new value of the SP or ESP register is outside the stack segment limit.
- #PF(fault-code) If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
- #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #SS(0) If the stack address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
- #UD If the LOCK prefix is used.

EXTRACTPS – Extract Packed Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 17 /r ib EXTRACTPS reg/m32, xmm2, imm8	MRI	V/V	SSE4_1	Extract a single-precision floating-point value from <i>xmm2</i> at the source offset specified by <i>imm8</i> and store the result to <i>reg</i> or <i>m32</i> . The upper 32 bits of r64 is zeroed if reg is r64.
VEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS r/m32, xmm1, imm8	MRI	V/V	AVX	Extract one single-precision floating-point value from <i>xmm1</i> at the offset specified by <i>imm8</i> and store the result in <i>reg</i> or <i>m32</i> . Zero extend the results in 64-bit register if applicable.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

Description

Extracts a single-precision floating-point value from the source operand (second operand) at the 32-bit offset specified from imm8. Immediate bits higher than the most significant offset for the vector length are ignored.

The extracted single-precision floating-point value is stored in the low 32-bits of the destination operand

In 64-bit mode, destination register operand has default operand size of 64 bits. The upper 32-bits of the register are filled with zero. REX.W is ignored.

128-bit Legacy SSE version: When a REX.W prefix is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

VEX.128 encoded version: When VEX.128.66.0F3A.W1 17 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits. VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

The source register is an XMM register. Imm8[1:0] determine the starting DWORD offset from which to extract the 32-bit floating-point value.

If VEXTRACTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation**EXTRACTPS (128-bit Legacy SSE version)**

```

SRC_OFFSET ← IMM8[1:0]
IF ( 64-Bit Mode and DEST is register)
    DEST[31:0] ← (SRC[127:0] » (SRC_OFFSET*32)) AND OFFFFFFFh
    DEST[63:32] ← 0
ELSE
    DEST[31:0] ← (SRC[127:0] » (SRC_OFFSET*32)) AND OFFFFFFFh
FI

```

VEXTRACTPS (VEX.128 encoded version)

```

SRC_OFFSET ← IMM8[1:0]
IF ( 64-Bit Mode and DEST is register)
    DEST[31:0] ← (SRC[127:0] » (SRC_OFFSET*32)) AND 0xFFFFFFFFh
    DEST[63:32] ← 0
ELSE
    DEST[31:0] ← (SRC[127:0] » (SRC_OFFSET*32)) AND 0xFFFFFFFFh
FI

```

Intel C/C++ Compiler Intrinsic Equivalent

EXTRACTPS: `_mm_extractmem_ps (float *dest, __m128 a, const int nidx);`

EXTRACTPS: `__m128 _mm_extract_ps (__m128 a, const int nidx);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L= 1.

F2XM1—Compute $2^{ST(0)} - 1$

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F0	F2XM1	Valid	Valid	Replace ST(0) with $(2^{ST(0)} - 1)$.

Description

Computes the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range -1.0 to $+1.0$. If the source value is outside this range, the result is undefined.

The following table shows the results obtained when computing the exponential value of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-26. Results Obtained from F2XM1

ST(0) SRC	ST(0) DEST
-1.0 to -0	-0.5 to -0
-0	-0
$+0$	$+0$
$+0$ to $+1.0$	$+0$ to 1.0

Values other than 2 can be exponentiated using the following formula:

$$x^y \leftarrow 2^{(y * \log_2 x)}$$

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

Operation

$$ST(0) \leftarrow (2^{ST(0)} - 1);$$

FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.
- C0, C2, C3 Undefined.

Floating-Point Exceptions

- #IS Stack underflow occurred.
- #IA Source operand is an SNaN value or unsupported format.
- #D Source is a denormal value.
- #U Result is too small for destination format.
- #P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #UD If the LOCK prefix is used.

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.

FABS—Absolute Value

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 E1	FABS	Valid	Valid	Replace ST with its absolute value.

Description

Clears the sign bit of ST(0) to create the absolute value of the operand. The following table shows the results obtained when creating the absolute value of various classes of numbers.

Table 3-27. Results Obtained from FABS

ST(0) SRC	ST(0) DEST
- ∞	+ ∞
- F	+ F
- 0	+ 0
+ 0	+ 0
+ F	+ F
+ ∞	+ ∞
NaN	NaN

NOTES:

F Means finite floating-point value.

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

Operation

$ST(0) \leftarrow |ST(0)|;$

FPU Flags Affected

C1 Set to 0.
C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

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.

FADD/FADDP/FIADD—Add

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /0	FADD <i>m32fp</i>	Valid	Valid	Add <i>m32fp</i> to ST(0) and store result in ST(0).
DC /0	FADD <i>m64fp</i>	Valid	Valid	Add <i>m64fp</i> to ST(0) and store result in ST(0).
D8 C0+i	FADD ST(0), ST(i)	Valid	Valid	Add ST(0) to ST(i) and store result in ST(0).
DC C0+i	FADD ST(i), ST(0)	Valid	Valid	Add ST(i) to ST(0) and store result in ST(i).
DE C0+i	FADDP ST(i), ST(0)	Valid	Valid	Add ST(0) to ST(i), store result in ST(i), and pop the register stack.
DE C1	FADDP	Valid	Valid	Add ST(0) to ST(1), store result in ST(1), and pop the register stack.
DA /0	FIADD <i>m32int</i>	Valid	Valid	Add <i>m32int</i> to ST(0) and store result in ST(0).
DE /0	FIADD <i>m16int</i>	Valid	Valid	Add <i>m16int</i> to ST(0) and store result in ST(0).

Description

Adds the destination and source operands and stores the sum in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction adds the contents of the ST(0) register to the ST(1) register. The one-operand version adds the contents of a memory location (either a floating-point or an integer value) to the contents of the ST(0) register. The two-operand version, adds the contents of the ST(0) register to the ST(i) register or vice versa. The value in ST(0) can be doubled by coding:

FADD ST(0), ST(0);

The FADDP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. (The no-operand version of the floating-point add instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FADD rather than FADDP.)

The FIADD instructions convert an integer source operand to double extended-precision floating-point format before performing the addition.

The table on the following page shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

When the sum of two operands with opposite signs is 0, the result is +0, except for the round toward $-\infty$ mode, in which case the result is -0. When the source operand is an integer 0, it is treated as a +0.

When both operand are infinities of the same sign, the result is ∞ of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated. See Table 3-28.

Table 3-28. FADD/FADDP/FIADD Results

		DEST						
		-∞	-F	-0	+0	+F	+∞	NaN
SRC	-∞	-∞	-∞	-∞	-∞	-∞	*	NaN
	-F or -I	-∞	-F	SRC	SRC	±F or ±0	+∞	NaN
	-0	-∞	DEST	-0	±0	DEST	+∞	NaN
	+0	-∞	DEST	±0	+0	DEST	+∞	NaN
	+F or +I	-∞	±F or ±0	SRC	SRC	+F	+∞	NaN
	+∞	*	+∞	+∞	+∞	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

IF Instruction = FIADD

THEN

DEST ← DEST + Convert.ToDoubleExtendedPrecisionFP(SRC);

ELSE (* Source operand is floating-point value *)

DEST ← DEST + SRC;

FI;

IF Instruction = FADDP

THEN

PopRegisterStack;

FI;

FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.
 Set if result was rounded up; cleared otherwise.
C0, C2, C3 Undefined.

Floating-Point Exceptions

- #IS Stack underflow occurred.
#IA Operand is an SNaN value or unsupported format.
 Operands are infinities of unlike sign.
#D Source operand is a denormal value.
#U Result is too small for destination format.
#O Result is too large for destination format.
#P Value cannot be represented exactly in destination format.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FBLD—Load Binary Coded Decimal

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DF /4	FBLD <i>m80dec</i>	Valid	Valid	Convert BCD value to floating-point and push onto the FPU stack.

Description

Converts the BCD source operand into double extended-precision floating-point format and pushes the value onto the FPU stack. The source operand is loaded without rounding errors. The sign of the source operand is preserved, including that of –0.

The packed BCD digits are assumed to be in the range 0 through 9; the instruction does not check for invalid digits (AH through FH). Attempting to load an invalid encoding produces an undefined result.

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

Operation

```
TOP ← TOP – 1;  
ST(0) ← Convert.ToDoubleExtendedPrecisionFP(SRC);
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS Stack overflow occurred.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FBSTP—Store BCD Integer and Pop

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DF /6	FBSTP m80bcd	Valid	Valid	Store ST(0) in m80bcd and pop ST(0).

Description

Converts the value in the ST(0) register to an 18-digit packed BCD integer, stores the result in the destination operand, and pops the register stack. If the source value is a non-integral value, it is rounded to an integer value, according to rounding mode specified by the RC field of the FPU control word. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The destination operand specifies the address where the first byte destination value is to be stored. The BCD value (including its sign bit) requires 10 bytes of space in memory.

The following table shows the results obtained when storing various classes of numbers in packed BCD format.

Table 3-29. FBSTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	-D
$-1 < F < -0$	**
-0	-0
+0	+0
$+0 < F < +1$	**
$F \geq +1$	+D
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

D Means packed-BCD number.

* Indicates floating-point invalid-operation (#IA) exception.

**±0 or ±1, depending on the rounding mode.

If the converted value is too large for the destination format, or if the source operand is an ∞ , SNaN, QNaN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the packed BCD indefinite value is stored in memory.

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

Operation

DEST \leftarrow BCD(ST(0));
PopRegisterStack;

FPU Flags Affected

- | | |
|------------|---|
| C1 | Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Converted value that exceeds 18 BCD digits in length.
	Source operand is an SNaN, QNaN, $\pm\infty$, or in an unsupported format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a segment register is being loaded with a segment selector that 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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FCHS—Change Sign

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 E0	FCHS	Valid	Valid	Complements sign of ST(0).

Description

Complements the sign bit of ST(0). This operation changes a positive value into a negative value of equal magnitude or vice versa. The following table shows the results obtained when changing the sign of various classes of numbers.

Table 3-30. FCHS Results

ST(0) SRC	ST(0) DEST
- ∞	+ ∞
- F	+ F
- 0	+ 0
+ 0	- 0
+ F	- F
+ ∞	- ∞
NaN	NaN

NOTES:

* F means finite floating-point value.

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

Operation

$\text{SignBit}(\text{ST}(0)) \leftarrow \text{NOT}(\text{SignBit}(\text{ST}(0)))$;

FPU Flags Affected

C1 Set to 0.
 C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
 #UD If the LOCK prefix is used.

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.

FCLEX/FNCLEX—Clear Exceptions

Opcode*	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B DB E2	FCLEX	Valid	Valid	Clear floating-point exception flags after checking for pending unmasked floating-point exceptions.
DB E2	FNCLEX*	Valid	Valid	Clear floating-point exception flags without checking for pending unmasked floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Clears the floating-point exception flags (PE, UE, OE, ZE, DE, and IE), the exception summary status flag (ES), the stack fault flag (SF), and the busy flag (B) in the FPU status word. The FCLEX instruction checks for and handles any pending unmasked floating-point exceptions before clearing the exception flags; the FNCLEX instruction does not.

The assembler issues two instructions for the FCLEX instruction (an FWAIT instruction followed by an FNCLEX instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS* compatibility mode, it is possible (under unusual circumstances) for an FNCLEX instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNCLEX instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

This instruction affects only the x87 FPU floating-point exception flags. It does not affect the SIMD floating-point exception flags in the MXCRS register.

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

Operation

```
FPUStatusWord[0:7] ← 0;  
FPUStatusWord[15] ← 0;
```

FPU Flags Affected

The PE, UE, OE, ZE, DE, IE, ES, SF, and B flags in the FPU status word are cleared. The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.

FCMOVcc—Floating-Point Conditional Move

Opcode*	Instruction	64-Bit Mode	Compat/Leg Mode*	Description
DA C0+i	FCMOVB ST(0), ST(i)	Valid	Valid	Move if below (CF=1).
DA C8+i	FCMOVE ST(0), ST(i)	Valid	Valid	Move if equal (ZF=1).
DA D0+i	FCMOVBE ST(0), ST(i)	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
DA D8+i	FCMOVU ST(0), ST(i)	Valid	Valid	Move if unordered (PF=1).
DB C0+i	FCMOVNBT ST(0), ST(i)	Valid	Valid	Move if not below (CF=0).
DB C8+i	FCMOVNE ST(0), ST(i)	Valid	Valid	Move if not equal (ZF=0).
DB D0+i	FCMOVNBE ST(0), ST(i)	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
DB D8+i	FCMOVNU ST(0), ST(i)	Valid	Valid	Move if not unordered (PF=0).

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Tests the status flags in the EFLAGS register and moves the source operand (second operand) to the destination operand (first operand) if the given test condition is true. The condition for each mnemonic is given in the Description column above and in Chapter 8 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*. The source operand is always in the ST(i) register and the destination operand is always ST(0).

The FCMOVcc instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

A processor may not support the FCMOVcc instructions. Software can check if the FCMOVcc instructions are supported by checking the processor’s feature information with the CPUID instruction (see “COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS” in this chapter). If both the CMOV and FPU feature bits are set, the FCMOVcc instructions are supported.

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

IA-32 Architecture Compatibility

The FCMOVcc instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

Operation

```
IF condition TRUE
  THEN ST(0) ← ST(i);
  Fl;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
-----	---------------------------

Integer Flags Affected

None.

Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #UD If the LOCK prefix is used.

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.

FCOM/FCOMP/FCOMPP—Compare Floating Point Values

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /2	FCOM <i>m32fp</i>	Valid	Valid	Compare ST(0) with <i>m32fp</i> .
DC /2	FCOM <i>m64fp</i>	Valid	Valid	Compare ST(0) with <i>m64fp</i> .
D8 D0+i	FCOM ST(i)	Valid	Valid	Compare ST(0) with ST(i).
D8 D1	FCOM	Valid	Valid	Compare ST(0) with ST(1).
D8 /3	FCOMP <i>m32fp</i>	Valid	Valid	Compare ST(0) with <i>m32fp</i> and pop register stack.
DC /3	FCOMP <i>m64fp</i>	Valid	Valid	Compare ST(0) with <i>m64fp</i> and pop register stack.
D8 D8+i	FCOMP ST(i)	Valid	Valid	Compare ST(0) with ST(i) and pop register stack.
D8 D9	FCOMP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack.
DE D9	FCOMPP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack twice.

Description

Compares the contents of register ST(0) and source value and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). The source operand can be a data register or a memory location. If no source operand is given, the value in ST(0) is compared with the value in ST(1). The sign of zero is ignored, so that -0.0 is equal to +0.0.

Table 3-31. FCOM/FCOMP/FCOMPP Results

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered*	1	1	1

NOTES:

- * Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

This instruction checks the class of the numbers being compared (see “FXAM—Examine ModR/M” in this chapter). If either operand is a NaN or is in an unsupported format, an invalid-arithmetic-operand exception (#IA) is raised and, if the exception is masked, the condition flags are set to “unordered.” If the invalid-arithmetic-operand exception is unmasked, the condition code flags are not set.

The FCOMP instruction pops the register stack following the comparison operation and the FCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The FCOM instructions perform the same operation as the FUCOM instructions. The only difference is how they handle QNaN operands. The FCOM instructions raise an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value or is in an unsupported format. The FUCOM instructions perform the same operation as the FCOM instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs.

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

Operation

```
CASE (relation of operands) OF
    ST > SRC:      C3, C2, C0 ← 000;
    ST < SRC:      C3, C2, C0 ← 001;
    ST = SRC:     C3, C2, C0 ← 100;
ESAC;

IF ST(0) or SRC = NaN or unsupported format
THEN
    #IA
    IF FPUControlWord.IM = 1
        THEN
            C3, C2, C0 ← 111;
        FI;
    FI;

IF Instruction = FCOMP
THEN
    PopRegisterStack;
FI;

IF Instruction = FCOMPP
THEN
    PopRegisterStack;
    PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	See table on previous page.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are NaN values or have unsupported formats.
	Register is marked empty.
#D	One or both operands are denormal values.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DB F0+i	FCOMI ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i) and set status flags accordingly.
DF F0+i	FCOMIP ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), set status flags accordingly, and pop register stack.
DB E8+i	FUCOMI ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), check for ordered values, and set status flags accordingly.
DF E8+i	FUCOMIP ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), check for ordered values, set status flags accordingly, and pop register stack.

Description

Performs an unordered comparison of the contents of registers ST(0) and ST(i) and sets the status flags ZF, PF, and CF in the EFLAGS register according to the results (see the table below). The sign of zero is ignored for comparisons, so that -0.0 is equal to +0.0.

Table 3-32. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results

Comparison Results*	ZF	PF	CF
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered**	1	1	1

NOTES:

* See the IA-32 Architecture Compatibility section below.

** Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see “FXAM—Examine ModR/M” in this chapter). The FUCOMI/FUCOMIP instructions perform the same operations as the FCOMI/FCOMIP instructions. The only difference is that the FUCOMI/FUCOMIP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOMI/FCOMIP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

If the operation results in an invalid-arithmetic-operand exception being raised, the status flags in the EFLAGS register are set only if the exception is masked.

The FCOMI/FCOMIP and FUCOMI/FUCOMIP instructions set the OF, SF and AF flags to zero in the EFLAGS register (regardless of whether an invalid-operation exception is detected).

The FCOMIP and FUCOMIP instructions also pop the register stack following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

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

IA-32 Architecture Compatibility

The FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

Operation

CASE (relation of operands) OF

```
ST(0) > ST(i):      ZF, PF, CF ← 000;
ST(0) < ST(i):      ZF, PF, CF ← 001;
ST(0) = ST(i):      ZF, PF, CF ← 100;
```

ESAC;

IF Instruction is FCOMI or FCOMIP

THEN

 IF ST(0) or ST(i) = NaN or unsupported format

 THEN

 #IA

 IF FPUControlWord.IM = 1

 THEN

 ZF, PF, CF ← 111;

 FI;

 FI;

 FI;

IF Instruction is FUCOMI or FUCOMIP

THEN

 IF ST(0) or ST(i) = QNaN, but not SNaN or unsupported format

 THEN

 ZF, PF, CF ← 111;

 ELSE (* ST(0) or ST(i) is SNaN or unsupported format *)

 #IA;

 IF FPUControlWord.IM = 1

 THEN

 ZF, PF, CF ← 111;

 FI;

 FI;

 FI;

IF Instruction is FCOMIP or FUCOMIP

THEN

 PopRegisterStack;

FI;

FPU Flags Affected

C1 Set to 0.

C0, C2, C3 Not affected.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA (FCOMI or FCOMIP instruction) One or both operands are NaN values or have unsupported formats.

 (FUCOMI or FUCOMIP instruction) One or both operands are SNaN values (but not QNaNs) or have undefined formats. Detection of a QNaN value does not raise an invalid-operand exception.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

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.

FCOS—Cosine

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 FF	FCOS	Valid	Valid	Replace ST(0) with its cosine.

Description

Computes the cosine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the cosine of various classes of numbers.

Table 3-33. FCOS Results

ST(0) SRC	ST(0) DEST
$-\infty$	*
$-F$	-1 to $+1$
-0	$+1$
$+0$	$+1$
$+F$	-1 to $+1$
$+\infty$	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π or by using the FPREM instruction with a divisor of 2π . See the section titled “Pi” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for π in performing such reductions.

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

Operation

```

IF |ST(0)| <  $2^{63}$ 
THEN
    C2  $\leftarrow$  0;
    ST(0)  $\leftarrow$  cosine(ST(0));
ELSE (* Source operand is out-of-range *)
    C2  $\leftarrow$  1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise. Undefined if C2 is 1.
C2	Set to 1 if outside range ($-2^{63} < \text{source operand} < +2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞ , or unsupported format.
#D	Source is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FDECSTP—Decrement Stack-Top Pointer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F6	FDECSTP	Valid	Valid	Decrement TOP field in FPU status word.

Description

Subtracts one from the TOP field of the FPU status word (decrements the top-of-stack pointer). If the TOP field contains a 0, it is set to 7. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected.

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

Operation

```
IF TOP = 0
    THEN TOP ← 7;
    ELSE TOP ← TOP - 1;
FI;
```

FPU Flags Affected

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

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.

FDIV/FDIVP/FIDIV—Divide

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /6	FDIV <i>m32fp</i>	Valid	Valid	Divide ST(0) by <i>m32fp</i> and store result in ST(0).
DC /6	FDIV <i>m64fp</i>	Valid	Valid	Divide ST(0) by <i>m64fp</i> and store result in ST(0).
D8 F0+i	FDIV ST(0), ST(i)	Valid	Valid	Divide ST(0) by ST(i) and store result in ST(0).
DC F8+i	FDIV ST(i), ST(0)	Valid	Valid	Divide ST(i) by ST(0) and store result in ST(i).
DE F8+i	FDIVP ST(i), ST(0)	Valid	Valid	Divide ST(i) by ST(0), store result in ST(i), and pop the register stack.
DE F9	FDIVP	Valid	Valid	Divide ST(1) by ST(0), store result in ST(1), and pop the register stack.
DA /6	FIDIV <i>m32int</i>	Valid	Valid	Divide ST(0) by <i>m32int</i> and store result in ST(0).
DE /6	FIDIV <i>m16int</i>	Valid	Valid	Divide ST(0) by <i>m16int</i> and store result in ST(0).

Description

Divides the destination operand by the source operand and stores the result in the destination location. The destination operand (dividend) is always in an FPU register; the source operand (divisor) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

The no-operand version of the instruction divides the contents of the ST(1) register by the contents of the ST(0) register. The one-operand version divides the contents of the ST(0) register by the contents of a memory location (either a floating-point or an integer value). The two-operand version, divides the contents of the ST(0) register by the contents of the ST(i) register or vice versa.

The FDIVP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIV rather than FDIVP.

The FIDIV instructions convert an integer source operand to double extended-precision floating-point format before performing the division. When the source operand is an integer 0, it is treated as a +0.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-34. FDIV/FDIVP/FIDIV Results

		DEST						
		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
SRC	$-\infty$	*	$+0$	$+0$	-0	-0	*	NaN
	$-F$	$+ \infty$	$+F$	$+0$	-0	$-F$	$-\infty$	NaN
	$-I$	$+ \infty$	$+F$	$+0$	-0	$-F$	$-\infty$	NaN
	-0	$+ \infty$	**	*	*	**	$-\infty$	NaN
	$+0$	$-\infty$	**	*	*	**	$+ \infty$	NaN
	$+I$	$-\infty$	$-F$	-0	$+0$	$+F$	$+ \infty$	NaN
	$+F$	$-\infty$	$-F$	-0	$+0$	$+F$	$+ \infty$	NaN
	$+\infty$	*	-0	-0	$+0$	$+0$	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

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

Operation

```

IF SRC = 0
  THEN
    #Z;
  ELSE
    IF Instruction is FIDIV
      THEN
        DEST ← DEST / ConvertToDoubleExtendedPrecisionFP(SRC);
      ELSE (* Source operand is floating-point value *)
        DEST ← DEST / SRC;
      FI;
    FI;

    IF Instruction = FDIVP
      THEN
        PopRegisterStack;
    FI;
  
```

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.
#IA Operand is an SNaN value or unsupported format.
 $\pm\infty / \pm\infty; \pm0 / \pm0$
#D Source is a denormal value.

#Z	DEST / ± 0 , where DEST is not equal to ± 0 .
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FDIVR/FDIVRP/FIDIVR—Reverse Divide

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /7	FDIVR <i>m32fp</i>	Valid	Valid	Divide <i>m32fp</i> by ST(0) and store result in ST(0).
DC /7	FDIVR <i>m64fp</i>	Valid	Valid	Divide <i>m64fp</i> by ST(0) and store result in ST(0).
D8 F8+i	FDIVR ST(i), ST(0)	Valid	Valid	Divide ST(i) by ST(0) and store result in ST(0).
DC F0+i	FDIVR ST(i), ST(0)	Valid	Valid	Divide ST(0) by ST(i) and store result in ST(i).
DE F0+i	FDIVRP ST(i), ST(0)	Valid	Valid	Divide ST(0) by ST(i), store result in ST(i), and pop the register stack.
DE F1	FDIVRP	Valid	Valid	Divide ST(0) by ST(1), store result in ST(1), and pop the register stack.
DA /7	FIDIVR <i>m32int</i>	Valid	Valid	Divide <i>m32int</i> by ST(0) and store result in ST(0).
DE /7	FIDIVR <i>m16int</i>	Valid	Valid	Divide <i>m16int</i> by ST(0) and store result in ST(0).

Description

Divides the source operand by the destination operand and stores the result in the destination location. The destination operand (divisor) is always in an FPU register; the source operand (dividend) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

These instructions perform the reverse operations of the FDIV, FDIVP, and FIDIV instructions. They are provided to support more efficient coding.

The no-operand version of the instruction divides the contents of the ST(0) register by the contents of the ST(1) register. The one-operand version divides the contents of a memory location (either a floating-point or an integer value) by the contents of the ST(0) register. The two-operand version, divides the contents of the ST(i) register by the contents of the ST(0) register or vice versa.

The FDIVRP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIVR rather than FDIVRP.

The FIDIVR instructions convert an integer source operand to double extended-precision floating-point format before performing the division.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-35. FDIVR/FDIVRP/FIDIVR Results

		DEST						
		– ∞	– F	– 0	+ 0	+ F	+ ∞	NaN
SRC	– ∞	*	+ ∞	+ ∞	– ∞	– ∞	*	NaN
	– F	+ 0	+ F	**	**	– F	– 0	NaN
	– I	+ 0	+ F	**	**	– F	– 0	NaN
	– 0	+ 0	+ 0	*	*	– 0	– 0	NaN
	+ 0	– 0	– 0	*	*	+ 0	+ 0	NaN
	+ I	– 0	– F	**	**	+ F	+ 0	NaN
	+ F	– 0	– F	**	**	+ F	+ 0	NaN
	+ ∞	*	– ∞	– ∞	+ ∞	+ ∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the source operand is an integer 0, it is treated as a +0. This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```

IF DEST = 0
  THEN
    #Z;
  ELSE
    IF Instruction = FIDIVR
      THEN
        DEST ← Convert.ToDoubleExtendedPrecisionFP(SRC) / DEST;
      ELSE (* Source operand is floating-point value *)
        DEST ← SRC / DEST;
    FI;
  FI;

IF Instruction = FDIVRP
  THEN
    PopRegisterStack;
  FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format. ±∞ / ±∞; ±0 / ±0

#D	Source is a denormal value.
#Z	SRC / ± 0 , where SRC is not equal to ± 0 .
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FFREE—Free Floating-Point Register

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DD C0+i	FFREE ST(i)	Valid	Valid	Sets tag for ST(i) to empty.

Description

Sets the tag in the FPU tag register associated with register ST(i) to empty (11B). The contents of ST(i) and the FPU stack-top pointer (TOP) are not affected.

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

Operation

$\text{TAG}(i) \leftarrow 11\text{B};$

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

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.

FICOM/FICOMP—Compare Integer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DE /2	FICOM <i>m16int</i>	Valid	Valid	Compare ST(0) with <i>m16int</i> .
DA /2	FICOM <i>m32int</i>	Valid	Valid	Compare ST(0) with <i>m32int</i> .
DE /3	FICOMP <i>m16int</i>	Valid	Valid	Compare ST(0) with <i>m16int</i> and pop stack register.
DA /3	FICOMP <i>m32int</i>	Valid	Valid	Compare ST(0) with <i>m32int</i> and pop stack register.

Description

Compares the value in ST(0) with an integer source operand and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below). The integer value is converted to double extended-precision floating-point format before the comparison is made.

Table 3-36. FICOM/FICOMP Results

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered	1	1	1

These instructions perform an “unordered comparison.” An unordered comparison also checks the class of the numbers being compared (see “FXAM—Examine ModR/M” in this chapter). If either operand is a NaN or is in an undefined format, the condition flags are set to “unordered.”

The sign of zero is ignored, so that $-0.0 \leftarrow +0.0$.

The FICOMP instructions pop the register stack following the comparison. To pop the register stack, the processor marks the ST(0) register empty and increments the stack pointer (TOP) by 1.

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

Operation

CASE (relation of operands) OF

```

ST(0) > SRC:    C3, C2, C0 ← 000;
ST(0) < SRC:    C3, C2, C0 ← 001;
ST(0) = SRC:    C3, C2, C0 ← 100;
Unordered:      C3, C2, C0 ← 111;

```

ESAC;

IF Instruction = FICOMP

THEN

```

PopRegisterStack;
F;

```

FPU Flags Affected

C1 Set to 0.

C0, C2, C3 See table on previous page.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA One or both operands are NaN values or have unsupported formats.

#D One or both operands are denormal values.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FILD—Load Integer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DF /0	FILD <i>m16int</i>	Valid	Valid	Push <i>m16int</i> onto the FPU register stack.
DB /0	FILD <i>m32int</i>	Valid	Valid	Push <i>m32int</i> onto the FPU register stack.
DF /5	FILD <i>m64int</i>	Valid	Valid	Push <i>m64int</i> onto the FPU register stack.

Description

Converts the signed-integer source operand into double extended-precision floating-point format and pushes the value onto the FPU register stack. The source operand can be a word, doubleword, or quadword integer. It is loaded without rounding errors. The sign of the source operand is preserved.

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

Operation

```
TOP ← TOP – 1;  
ST(0) ← Convert.ToDoubleExtendedPrecisionFP(SRC);
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; set to 0 otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack overflow occurred.
-----	--------------------------

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FINCSTP—Increment Stack-Top Pointer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F7	FINCSTP	Valid	Valid	Increment the TOP field in the FPU status register.

Description

Adds one to the TOP field of the FPU status word (increments the top-of-stack pointer). If the TOP field contains a 7, it is set to 0. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected. This operation is not equivalent to popping the stack, because the tag for the previous top-of-stack register is not marked empty.

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

Operation

```
IF TOP = 7
  THEN TOP ← 0;
  ELSE TOP ← TOP + 1;
FI;
```

FPU Flags Affected

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

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.

FINIT/FNINIT—Initialize Floating-Point Unit

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B DB E3	FINIT	Valid	Valid	Initialize FPU after checking for pending unmasked floating-point exceptions.
DB E3	FNINIT*	Valid	Valid	Initialize FPU without checking for pending unmasked floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Sets the FPU control, status, tag, instruction pointer, and data pointer registers to their default states. The FPU control word is set to 037FH (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, TOP is set to 0). The data registers in the register stack are left unchanged, but they are all tagged as empty (11B). Both the instruction and data pointers are cleared.

The FINIT instruction checks for and handles any pending unmasked floating-point exceptions before performing the initialization; the FNINIT instruction does not.

The assembler issues two instructions for the FINIT instruction (an FWAIT instruction followed by an FNINIT instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNINIT instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNINIT instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

In the Intel387 math coprocessor, the FINIT/FNINIT instruction does not clear the instruction and data pointers.

This instruction affects only the x87 FPU. It does not affect the XMM and MXCSR registers.

Operation

```
FPUControlWord ← 037FH;
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;
```

FPU Flags Affected

C0, C1, C2, C3 set to 0.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

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.

FIST/FISTP—Store Integer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DF /2	FIST <i>m16int</i>	Valid	Valid	Store ST(0) in <i>m16int</i> .
DB /2	FIST <i>m32int</i>	Valid	Valid	Store ST(0) in <i>m32int</i> .
DF /3	FISTP <i>m16int</i>	Valid	Valid	Store ST(0) in <i>m16int</i> and pop register stack.
DB /3	FISTP <i>m32int</i>	Valid	Valid	Store ST(0) in <i>m32int</i> and pop register stack.
DF /7	FISTP <i>m64int</i>	Valid	Valid	Store ST(0) in <i>m64int</i> and pop register stack.

Description

The FIST instruction converts the value in the ST(0) register to a signed integer and stores the result in the destination operand. Values can be stored in word or doubleword integer format. The destination operand specifies the address where the first byte of the destination value is to be stored.

The FISTP instruction performs the same operation as the FIST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FISTP instruction also stores values in quadword integer format.

The following table shows the results obtained when storing various classes of numbers in integer format.

Table 3-37. FIST/FISTP Results

ST(0)	DEST
-∞ or Value Too Large for DEST Format	*
$F \leq -1$	-I
$-1 < F < -0$	**
-0	0
+0	0
$+0 < F < +1$	**
$F \geq +1$	+I
+∞ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-operation (#IA) exception.

** 0 or ±1, depending on the rounding mode.

If the source value is a non-integral value, it is rounded to an integer value, according to the rounding mode specified by the RC field of the FPU control word.

If the converted value is too large for the destination format, or if the source operand is an ∞, SNaN, QNaN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the integer indefinite value is stored in memory.

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

Operation

```
DEST ← Integer(ST(0));
IF Instruction = FISTP
  THEN
    PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction of if the inexact exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
C0, C2, C3	Set to 0 otherwise. Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Converted value is too large for the destination format.
#P	Source operand is an SNaN, QNaN, $\pm\infty$, or unsupported format. Value cannot be represented exactly in destination format.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FISTTP—Store Integer with Truncation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /1	FISTTP $m16int$	Valid	Valid	Store ST(0) in $m16int$ with truncation.
DB /1	FISTTP $m32int$	Valid	Valid	Store ST(0) in $m32int$ with truncation.
DD /1	FISTTP $m64int$	Valid	Valid	Store ST(0) in $m64int$ with truncation.

Description

FISTTP converts the value in ST into a signed integer using truncation (chop) as rounding mode, transfers the result to the destination, and pop ST. FISTTP accepts word, short integer, and long integer destinations.

The following table shows the results obtained when storing various classes of numbers in integer format.

Table 3-38. FISTTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	-I
$-1 < F < +1$	0
$F \geq +1$	+I
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-operation (#IA) exception.

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

Operation

DEST \leftarrow ST;
pop ST;

Flags Affected

C1 is cleared; C0, C2, C3 undefined.

Numeric Exceptions

Invalid, Stack Invalid (stack underflow), Precision.

Protected Mode Exceptions

#GP(0)	If the destination is in a nonwritable segment.
	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#NM	If CR0.EM[bit 2] = 1. If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.SSE3[bit 0] = 0.
 If the LOCK prefix is used.

Real Address Mode Exceptions

GP(0) If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
 If CR0.EM[bit 2] = 1.
 If CR0.TS[bit 3] = 1.
 #UD If CPUID.01H:ECX.SSE3[bit 0] = 0.
 If the LOCK prefix is used.

Virtual 8086 Mode Exceptions

GP(0) If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
 If CR0.EM[bit 2] = 1.
 If CR0.TS[bit 3] = 1.
 #UD If CPUID.01H:ECX.SSE3[bit 0] = 0.
 If the LOCK prefix is used.
 #PF(fault-code) For a page fault.
 #AC(0) For unaligned memory reference if the current privilege is 3.

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.
 #NM CR0.EM[bit 2] or 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.
 If the LOCK prefix is used.

FLD—Load Floating Point Value

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 /0	FLD <i>m32fp</i>	Valid	Valid	Push <i>m32fp</i> onto the FPU register stack.
DD /0	FLD <i>m64fp</i>	Valid	Valid	Push <i>m64fp</i> onto the FPU register stack.
DB /5	FLD <i>m80fp</i>	Valid	Valid	Push <i>m80fp</i> onto the FPU register stack.
D9 C0+i	FLD ST(i)	Valid	Valid	Push ST(i) onto the FPU register stack.

Description

Pushes the source operand onto the FPU register stack. The source operand can be in single-precision, double-precision, or double extended-precision floating-point format. If the source operand is in single-precision or double-precision floating-point format, it is automatically converted to the double extended-precision floating-point format before being pushed on the stack.

The FLD instruction can also push the value in a selected FPU register [ST(i)] onto the stack. Here, pushing register ST(0) duplicates the stack top.

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

Operation

```

IF SRC is ST(i)
  THEN
    temp ← ST(i);
  Fl;
TOP ← TOP – 1;

IF SRC is memory-operand
  THEN
    ST(0) ← Convert.ToDoubleExtendedPrecisionFP(SRC);
  ELSE (* SRC is ST(i) *)
    ST(0) ← temp;
  Fl;
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN. Does not occur if the source operand is in double extended-precision floating-point format (FLD m80fp or FLD ST(i)).
#D	Source operand is a denormal value. Does not occur if the source operand is in double extended-precision floating-point format.

Protected Mode Exceptions

#GP(0)	If 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.

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant

Opcode*	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 E8	FLD1	Valid	Valid	Push +1.0 onto the FPU register stack.
D9 E9	FLDL2T	Valid	Valid	Push $\log_2 10$ onto the FPU register stack.
D9 EA	FLDL2E	Valid	Valid	Push $\log_2 e$ onto the FPU register stack.
D9 EB	FLDPI	Valid	Valid	Push π onto the FPU register stack.
D9 EC	FLDLG2	Valid	Valid	Push $\log_{10} 2$ onto the FPU register stack.
D9 ED	FLDLN2	Valid	Valid	Push $\log_e 2$ onto the FPU register stack.
D9 EE	FLDZ	Valid	Valid	Push +0.0 onto the FPU register stack.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Push one of seven commonly used constants (in double extended-precision floating-point format) onto the FPU register stack. The constants that can be loaded with these instructions include +1.0, +0.0, $\log_2 10$, $\log_2 e$, π , $\log_{10} 2$, and $\log_e 2$. For each constant, an internal 66-bit constant is rounded (as specified by the RC field in the FPU control word) to double extended-precision floating-point format. The inexact-result exception (#P) is not generated as a result of the rounding, nor is the C1 flag set in the x87 FPU status word if the value is rounded up.

See the section titled “Pi” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of the π constant.

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

IA-32 Architecture Compatibility

When the RC field is set to round-to-nearest, the FPU produces the same constants that is produced by the Intel 8087 and Intel 287 math coprocessors.

Operation

```
TOP ← TOP – 1;  
ST(0) ← CONSTANT;
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS Stack overflow occurred.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 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.

FLDCW—Load x87 FPU Control Word

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 /5	FLDCW m2byte	Valid	Valid	Load FPU control word from <i>m2byte</i> .

Description

Loads the 16-bit source operand into the FPU control word. The source operand is a memory location. This instruction is typically used to establish or change the FPU's mode of operation.

If one or more exception flags are set in the FPU status word prior to loading a new FPU control word and the new control word unmasks one or more of those exceptions, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled "Software Exception Handling" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). To avoid raising exceptions when changing FPU operating modes, clear any pending exceptions (using the FCLEX or FNCLEX instruction) before loading the new control word.

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

Operation

FPUControlWord ← SRC;

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-Point Exceptions

None; however, this operation might unmask a pending exception in the FPU status word. That exception is then generated upon execution of the next "waiting" floating-point instruction.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FLDENV—Load x87 FPU Environment

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 /4	FLDENV <i>m14/28byte</i>	Valid	Valid	Load FPU environment from <i>m14byte</i> or <i>m28byte</i> .

Description

Loads the complete x87 FPU operating environment from memory into the FPU registers. The source operand specifies the first byte of the operating-environment data in memory. This data is typically written to the specified memory location by a FSTENV or FNSTENV instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, show the layout in memory of the loaded environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FLDENV instruction should be executed in the same operating mode as the corresponding FSTENV/FNSTENV instruction.

If one or more unmasked exception flags are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled “Software Exception Handling” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). To avoid generating exceptions when loading a new environment, clear all the exception flags in the FPU status word that is being loaded.

If a page or limit fault occurs during the execution of this instruction, the state of the x87 FPU registers as seen by the fault handler may be different than the state being loaded from memory. In such situations, the fault handler should ignore the status of the x87 FPU registers, handle the fault, and return. The FLDENV instruction will then complete the loading of the x87 FPU registers with no resulting context inconsistency.

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

Operation

```
FPUControlWord ← SRC[FPUControlWord];
FPUStatusWord ← SRC[FPUStatusWord];
FPUTagWord ← SRC[FPUTagWord];
FPUDataPointer ← SRC[FPUDataPointer];
FPUInstructionPointer ← SRC[FPUInstructionPointer];
FPULastInstructionOpcode ← SRC[FPULastInstructionOpcode];
```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-Point Exceptions

None; however, if an unmasked exception is loaded in the status word, it is generated upon execution of the next “waiting” floating-point instruction.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
- #UD If the LOCK prefix is used.

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.
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #UD If the LOCK prefix is used.

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.
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used.

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.
- #NM CR0.EM[bit 2] or 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.
- #UD If the LOCK prefix is used.

FMUL/FMULP/FIMUL—Multiply

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /1	FMUL <i>m32fp</i>	Valid	Valid	Multiply ST(0) by <i>m32fp</i> and store result in ST(0).
DC /1	FMUL <i>m64fp</i>	Valid	Valid	Multiply ST(0) by <i>m64fp</i> and store result in ST(0).
D8 C8+i	FMUL ST(0), ST(i)	Valid	Valid	Multiply ST(0) by ST(i) and store result in ST(0).
DC C8+i	FMUL ST(i), ST(0)	Valid	Valid	Multiply ST(i) by ST(0) and store result in ST(i).
DE C8+i	FMULP ST(i), ST(0)	Valid	Valid	Multiply ST(i) by ST(0), store result in ST(i), and pop the register stack.
DE C9	FMULP	Valid	Valid	Multiply ST(1) by ST(0), store result in ST(1), and pop the register stack.
DA /1	FIMUL <i>m32int</i>	Valid	Valid	Multiply ST(0) by <i>m32int</i> and store result in ST(0).
DE /1	FIMUL <i>m16int</i>	Valid	Valid	Multiply ST(0) by <i>m16int</i> and store result in ST(0).

Description

Multiplies the destination and source operands and stores the product in the destination location. The destination operand is always an FPU data register; the source operand can be an FPU data register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction multiplies the contents of the ST(1) register by the contents of the ST(0) register and stores the product in the ST(1) register. The one-operand version multiplies the contents of the ST(0) register by the contents of a memory location (either a floating point or an integer value) and stores the product in the ST(0) register. The two-operand version, multiplies the contents of the ST(0) register by the contents of the ST(i) register, or vice versa, with the result being stored in the register specified with the first operand (the destination operand).

The FMULP instructions perform the additional operation of popping the FPU register stack after storing the product. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point multiply instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FMUL rather than FMULP.

The FIMUL instructions convert an integer source operand to double extended-precision floating-point format before performing the multiplication.

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the values being multiplied is 0 or ∞ . When the source operand is an integer 0, it is treated as a +0.

The following table shows the results obtained when multiplying various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-39. FMUL/FMULP/FIMUL Results

		DEST						
		– ∞	– F	– 0	+ 0	+ F	+ ∞	NaN
SRC	– ∞	+ ∞	+ ∞	*	*	– ∞	– ∞	NaN
	– F	+ ∞	+ F	+ 0	– 0	– F	– ∞	NaN
	– I	+ ∞	+ F	+ 0	– 0	– F	– ∞	NaN
	– 0	*	+ 0	+ 0	– 0	– 0	*	NaN
	+ 0	*	– 0	– 0	+ 0	+ 0	*	NaN
	+ I	– ∞	– F	– 0	+ 0	+ F	+ ∞	NaN
	+ F	– ∞	– F	– 0	+ 0	+ F	+ ∞	NaN
	+ ∞	– ∞	– ∞	*	*	+ ∞	+ ∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means Integer.

* Indicates invalid-arithmetic-operand (#IA) exception.

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

Operation

IF Instruction = FIMUL

THEN

DEST ← DEST * Convert.ToDoubleExtendedPrecisionFP(SRC);

ELSE (* Source operand is floating-point value *)

DEST ← DEST * SRC;

Fl;

IF Instruction = FMULP

THEN

PopRegisterStack;

Fl;

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA Operand is an SNaN value or unsupported format.

 One operand is ± 0 and the other is $\pm\infty$.

#D Source operand is a denormal value.

#U Result is too small for destination format.

#O Result is too large for destination format.

#P Value cannot be represented exactly in destination format.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FNOP—No Operation

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 D0	FNOP	Valid	Valid	No operation is performed.

Description

Performs no FPU operation. This instruction takes up space in the instruction stream but does not affect the FPU or machine context, except the EIP register.

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

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

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.

FPATAN—Partial Arctangent

Opcode*	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F3	FPATAN	Valid	Valid	Replace ST(1) with arctan(ST(1)/ST(0)) and pop the register stack.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Computes the arctangent of the source operand in register ST(1) divided by the source operand in register ST(0), stores the result in ST(1), and pops the FPU register stack. The result in register ST(0) has the same sign as the source operand ST(1) and a magnitude less than $+\pi$.

The FPATAN instruction returns the angle between the X axis and the line from the origin to the point (X,Y), where Y (the ordinate) is ST(1) and X (the abscissa) is ST(0). The angle depends on the sign of X and Y independently, not just on the sign of the ratio Y/X. This is because a point $(-X,Y)$ is in the second quadrant, resulting in an angle between $\pi/2$ and π , while a point $(X,-Y)$ is in the fourth quadrant, resulting in an angle between 0 and $-\pi/2$. A point $(-X,-Y)$ is in the third quadrant, giving an angle between $-\pi/2$ and $-\pi$.

The following table shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur.

Table 3-40. FPATAN Results

		ST(0)						
		$-\infty$	$-F$	-0	$+0$	$+F$	$+\infty$	NaN
ST(1)	$-\infty$	$-3\pi/4^*$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/4^*$	NaN
	$-F$	$-p$	$-\pi \text{ to } -\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2 \text{ to } -0$	-0	NaN
	-0	$-p$	$-p$	$-p^*$	-0^*	-0	-0	NaN
	$+0$	$+p$	$+p$	$+\pi^*$	$+0^*$	$+0$	$+0$	NaN
	$+F$	$+p$	$+\pi \text{ to } +\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2 \text{ to } +0$	$+0$	NaN
	$+\infty$	$+3\pi/4^*$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/4^*$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Table 8-10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, specifies that the ratios $0/0$ and ∞/∞ generate the floating-point invalid arithmetic-operation exception and, if this exception is masked, the floating-point QNaN indefinite value is returned. With the FPATAN instruction, the $0/0$ or ∞/∞ value is actually not calculated using division. Instead, the arctangent of the two variables is derived from a standard mathematical formulation that is generalized to allow complex numbers as arguments. In this complex variable formulation, $\arctan(0,0)$ etc. has well defined values. These values are needed to develop a library to compute transcendental functions with complex arguments, based on the FPU functions that only allow floating-point values as arguments.

There is no restriction on the range of source operands that FPATAN can accept.

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

IA-32 Architecture Compatibility

The source operands for this instruction are restricted for the 80287 math coprocessor to the following range:

$$0 \leq |ST(1)| < |ST(0)| < +\infty$$

Operation

$ST(1) \leftarrow \arctan(ST(1) / ST(0));$
 PopRegisterStack;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FPREM—Partial Remainder

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F8	FPREM	Valid	Valid	Replace ST(0) with the remainder obtained from dividing ST(0) by ST(1).

Description

Computes the remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

$$\text{Remainder} \leftarrow \text{ST}(0) - (\text{Q} * \text{ST}(1))$$

Here, Q is an integer value that is obtained by truncating the floating-point number quotient of [ST(0) / ST(1)] toward zero. The sign of the remainder is the same as the sign of the dividend. The magnitude of the remainder is less than that of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the inexact-result exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 3-41. FPREM Results

		ST(1)						
		-∞	-F	-0	+0	+F	+∞	NaN
ST(0)	-∞	*	*	*	*	*	*	NaN
	-F	ST(0)	-F or -0	**	**	-F or -0	ST(0)	NaN
	-0	-0	-0	*	*	-0	-0	NaN
	+0	+0	+0	*	*	+0	+0	NaN
	+F	ST(0)	+F or +0	**	**	+F or +0	ST(0)	NaN
	+∞	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞ , the result is equal to the value in ST(0).

The FPREM instruction does not compute the remainder specified in IEEE Std 754. The IEEE specified remainder can be computed with the FPREM1 instruction. The FPREM instruction is provided for compatibility with the Intel 8087 and Intel287 math coprocessors.

The FPREM instruction gets its name “partial remainder” because of the way it computes the remainder. This instruction arrives at a remainder through iterative subtraction. It can, however, reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU

status word. This information is important in argument reduction for the tangent function (using a modulus of $\pi/4$), because it locates the original angle in the correct one of eight sectors of the unit circle.

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

Operation

```
D ← exponent(ST(0)) - exponent(ST(1));
IF D < 64
  THEN
    Q ← Integer(TruncateTowardZero(ST(0) / ST(1)));
    ST(0) ← ST(0) - (ST(1) * Q);
    C2 ← 0;
    C0, C3, C1 ← LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
  ELSE
    C2 ← 1;
    N ← An implementation-dependent number between 32 and 63;
    QQ ← Integer(TruncateTowardZero((ST(0) / ST(1)) / 2(D-N)));
    ST(0) ← ST(0) - (ST(1) * QQ * 2(D-N));
FI;
```

FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, modulus is 0, dividend is ∞ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FPREM1—Partial Remainder

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F5	FPREM1	Valid	Valid	Replace ST(0) with the IEEE remainder obtained from dividing ST(0) by ST(1).

Description

Computes the IEEE remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

$$\text{Remainder} \leftarrow \text{ST}(0) - (\text{Q} * \text{ST}(1))$$

Here, Q is an integer value that is obtained by rounding the floating-point number quotient of [ST(0) / ST(1)] toward the nearest integer value. The magnitude of the remainder is less than or equal to half the magnitude of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 3-42. FPREM1 Results

		ST(1)						
		− ∞	− F	− 0	+ 0	+ F	+ ∞	NaN
ST(0)	− ∞	*	*	*	*	*	NaN	
	− F	ST(0)	±F or −0	**	**	±F or −0	ST(0)	NaN
	− 0	− 0	− 0	*	*	− 0	− 0	NaN
	+ 0	+ 0	+ 0	*	*	+ 0	+ 0	NaN
	+ F	ST(0)	± F or + 0	**	**	± F or + 0	ST(0)	NaN
	+ ∞	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞ , the result is equal to the value in ST(0).

The FPREM1 instruction computes the remainder specified in IEEE Standard 754. This instruction operates differently from the FPREM instruction in the way that it rounds the quotient of ST(0) divided by ST(1) to an integer (see the “Operation” section below).

Like the FPREM instruction, FPREM1 computes the remainder through iterative subtraction, but can reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than one half the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM1 instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU

status word. This information is important in argument reduction for the tangent function (using a modulus of $\pi/4$), because it locates the original angle in the correct one of eight sectors of the unit circle.

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

Operation

```
D ← exponent(ST(0)) - exponent(ST(1));
IF D < 64
  THEN
    Q ← Integer(RoundTowardNearestInteger(ST(0) / ST(1)));
    ST(0) ← ST(0) - (ST(1) * Q);
    C2 ← 0;
    C0, C3, C1 ← LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
  ELSE
    C2 ← 1;
    N ← An implementation-dependent number between 32 and 63;
    QQ ← Integer(TruncateTowardZero((ST(0) / ST(1)) / 2(D-N)));
    ST(0) ← ST(0) - (ST(1) * QQ * 2(D-N));
FI;
```

FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, modulus (divisor) is 0, dividend is ∞ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FPTAN—Partial Tangent

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F2	FPTAN	Valid	Valid	Replace ST(0) with its tangent and push 1 onto the FPU stack.

Description

Computes the tangent of the source operand in register ST(0), stores the result in ST(0), and pushes a 1.0 onto the FPU register stack. The source operand must be given in radians and must be less than $\pm 2^{63}$. The following table shows the unmasked results obtained when computing the partial tangent of various classes of numbers, assuming that underflow does not occur.

Table 3-43. FPTAN Results

ST(0) SRC	ST(0) DEST
$-\infty$	*
$-F$	$-F$ to $+F$
-0	-0
$+0$	$+0$
$+F$	$-F$ to $+F$
$+\infty$	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π or by using the FPREM instruction with a divisor of 2π . See the section titled "Pi" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a discussion of the proper value to use for π in performing such reductions.

The value 1.0 is pushed onto the register stack after the tangent has been computed to maintain compatibility with the Intel 8087 and Intel287 math coprocessors. This operation also simplifies the calculation of other trigonometric functions. For instance, the cotangent (which is the reciprocal of the tangent) can be computed by executing a FDIVR instruction after the FPTAN instruction.

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

Operation

```

IF ST(0) <  $2^{63}$ 
  THEN
    C2 ← 0;
    ST(0) ← tan(ST(0));
    TOP ← TOP - 1;
    ST(0) ← 1.0;
  ELSE (* Source operand is out-of-range *)
    C2 ← 1;
  Fl;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
	Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range ($-2^{63} < \text{source operand} < +2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, ∞ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FRNDINT—Round to Integer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 FC	FRNDINT	Valid	Valid	Round ST(0) to an integer.

Description

Rounds the source value in the ST(0) register to the nearest integral value, depending on the current rounding mode (setting of the RC field of the FPU control word), and stores the result in ST(0).

If the source value is ∞ , the value is not changed. If the source value is not an integral value, the floating-point inexact-result exception (#P) is generated.

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

Operation

$ST(0) \leftarrow \text{RoundToIntegralValue}(ST(0));$

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#P	Source operand is not an integral value.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FRSTOR—Restore x87 FPU State

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DD /4	FRSTOR <i>m94/108byte</i>	Valid	Valid	Load FPU state from <i>m94byte</i> or <i>m108byte</i> .

Description

Loads the FPU state (operating environment and register stack) from the memory area specified with the source operand. This state data is typically written to the specified memory location by a previous FSAVE/FNSAVE instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately following the operating environment image.

The FRSTOR instruction should be executed in the same operating mode as the corresponding FSAVE/FNSAVE instruction.

If one or more unmasked exception bits are set in the new FPU status word, a floating-point exception will be generated. To avoid raising exceptions when loading a new operating environment, clear all the exception flags in the FPU status word that is being loaded.

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

Operation

```

FPUControlWord ← SRC[FPUControlWord];
FPUStatusWord ← SRC[FPUStatusWord];
FPUTagWord ← SRC[FPUTagWord];
FPUDataPointer ← SRC[FPUDataPointer];
FPUInstructionPointer ← SRC[FPUInstructionPointer];
FPULastInstructionOpcode ← SRC[FPULastInstructionOpcode];

ST(0) ← SRC[ST(0)];
ST(1) ← SRC[ST(1)];
ST(2) ← SRC[ST(2)];
ST(3) ← SRC[ST(3)];
ST(4) ← SRC[ST(4)];
ST(5) ← SRC[ST(5)];
ST(6) ← SRC[ST(6)];
ST(7) ← SRC[ST(7)];

```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-Point Exceptions

None; however, this operation might unmask an existing exception that has been detected but not generated, because it was masked. Here, the exception is generated at the completion of the instruction.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

FSAVE/FNSAVE—Store x87 FPU State

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B DD /6	FSAVE <i>m94/108byte</i>	Valid	Valid	Store FPU state to <i>m94byte</i> or <i>m108byte</i> after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.
DD /6	FNSAVE* <i>m94/108byte</i>	Valid	Valid	Store FPU environment to <i>m94byte</i> or <i>m108byte</i> without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Stores the current FPU state (operating environment and register stack) at the specified destination in memory, and then re-initializes the FPU. The FSAVE instruction checks for and handles pending unmasked floating-point exceptions before storing the FPU state; the FNSAVE instruction does not.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The saved image reflects the state of the FPU after all floating-point instructions preceding the FSAVE/FNSAVE instruction in the instruction stream have been executed.

After the FPU state has been saved, the FPU is reset to the same default values it is set to with the FINIT/FNINIT instructions (see “FINIT/FNINIT—Initialize Floating-Point Unit” in this chapter).

The FSAVE/FNSAVE instructions are typically used when the operating system needs to perform a context switch, an exception handler needs to use the FPU, or an application program needs to pass a “clean” FPU to a procedure.

The assembler issues two instructions for the FSAVE instruction (an FWAIT instruction followed by an FNSAVE instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

For Intel math coprocessors and FPUs prior to the Intel Pentium processor, an FWAIT instruction should be executed before attempting to read from the memory image stored with a prior FSAVE/FNSAVE instruction. This FWAIT instruction helps ensure that the storage operation has been completed.

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSAVE instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNSAVE instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

Operation

(* Save FPU State and Registers *)

```
DEST[FPUControlWord] ← FPUControlWord;
DEST[FPUStatusWord] ← FPUStatusWord;
DEST[FPUTagWord] ← FPUTagWord;
DEST[FPUDataPointer] ← FPUDataPointer;
DEST[FPUIstructionPointer] ← FPUIstructionPointer;
DEST[FPULastInstructionOpcode] ← FPULastInstructionOpcode;

DEST[ST(0)] ← ST(0);
DEST[ST(1)] ← ST(1);
DEST[ST(2)] ← ST(2);
DEST[ST(3)] ← ST(3);
DEST[ST(4)] ← ST(4);
DEST[ST(5)] ← ST(5);
DEST[ST(6)] ← ST(6);
DEST[ST(7)] ← ST(7);
```

(* Initialize FPU *)

```
FPUControlWord ← 037FH;
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUIstructionPointer ← 0;
FPULastInstructionOpcode ← 0;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are saved and then cleared.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If 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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.

FSCALE—Scale

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 FD	FSCALE	Valid	Valid	Scale ST(0) by ST(1).

Description

Truncates the value in the source operand (toward 0) to an integral value and adds that value to the exponent of the destination operand. The destination and source operands are floating-point values located in registers ST(0) and ST(1), respectively. This instruction provides rapid multiplication or division by integral powers of 2. The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-44. FSCALE Results

ST(1)		-∞	-F	-0	+0	+F	+∞	NaN
ST(0)	-∞	NaN	-∞	-∞	-∞	-∞	-∞	NaN
	-F	-0	-F	-F	-F	-F	-∞	NaN
	-0	-0	-0	-0	-0	-0	NaN	NaN
	+0	+0	+0	+0	+0	+0	NaN	NaN
	+F	+0	+F	+F	+F	+F	+∞	NaN
	+∞	NaN	+∞	+∞	+∞	+∞	+∞	NaN
	NaN							

NOTES:

F Means finite floating-point value.

In most cases, only the exponent is changed and the mantissa (significand) remains unchanged. However, when the value being scaled in ST(0) is a denormal value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

The FSCALE instruction can also be used to reverse the action of the FXTRACT instruction, as shown in the following example:

```
FXTRACT;
FSCALE;
FSTP ST(1);
```

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value before the FXTRACT operation was performed. The FSTP ST(1) instruction overwrites the exponent (extracted by the FXTRACT instruction) with the recreated value, which returns the stack to its original state with only one register [ST(0)] occupied.

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

Operation

$ST(0) \leftarrow ST(0) * 2^{\text{RoundTowardZero}(ST(1))};$

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FSIN—Sine

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 FE	FSIN	Valid	Valid	Replace ST(0) with its sine.

Description

Computes the sine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the sine of various classes of numbers, assuming that underflow does not occur.

Table 3-45. FSIN Results

SRC (ST(0))	DEST (ST(0))
$-\infty$	*
$-F$	$-1 \text{ to } +1$
-0	-0
$+0$	$+0$
$+F$	$-1 \text{ to } +1$
$+\infty$	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π or by using the FPREM instruction with a divisor of 2π . See the section titled “Pi” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for π in performing such reductions.

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

Operation

```
IF  $-2^{63} < ST(0) < 2^{63}$ 
  THEN
    C2  $\leftarrow 0$ ;
    ST(0)  $\leftarrow \sin(ST(0))$ ;
  ELSE (* Source operand out of range *)
    C2  $\leftarrow 1$ ;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range ($-2^{63} < \text{source operand} < +2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞ , or unsupported format.
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FSINCOS—Sine and Cosine

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 FB	FSINCOS	Valid	Valid	Compute the sine and cosine of ST(0); replace ST(0) with the sine, and push the cosine onto the register stack.

Description

Computes both the sine and the cosine of the source operand in register ST(0), stores the sine in ST(0), and pushes the cosine onto the top of the FPU register stack. (This instruction is faster than executing the FSIN and FCOS instructions in succession.)

The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the sine and cosine of various classes of numbers, assuming that underflow does not occur.

Table 3-46. FSINCOS Results

SRC	DEST	
	ST(1) Cosine	ST(0) Sine
$-\infty$	*	*
$-F$	$-1 \text{ to } +1$	$-1 \text{ to } +1$
-0	$+1$	-0
$+0$	$+1$	$+0$
$+F$	$-1 \text{ to } +1$	$-1 \text{ to } +1$
$+\infty$	*	*
NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π or by using the FPREM instruction with a divisor of 2π . See the section titled “Pi” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for π in performing such reductions.

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

Operation

```

IF ST(0) <  $2^{63}$ 
  THEN
    C2  $\leftarrow$  0;
    TEMP  $\leftarrow$  cosine(ST(0));
    ST(0)  $\leftarrow$  sine(ST(0));
    TOP  $\leftarrow$  TOP - 1;
    ST(0)  $\leftarrow$  TEMP;
  ELSE (* Source operand out of range *)
    C2  $\leftarrow$  1;
  Fl;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurs.
	Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range ($-2^{63} < \text{source operand} < +2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, ∞ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FSQRT—Square Root

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 FA	FSQRT	Valid	Valid	Computes square root of ST(0) and stores the result in ST(0).

Description

Computes the square root of the source value in the ST(0) register and stores the result in ST(0).

The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-47. FSQRT Results

SRC (ST(0))	DEST (ST(0))
$-\infty$	*
$-F$	*
-0	-0
$+0$	$+0$
$+F$	$+F$
$+\infty$	$+\infty$
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

$ST(0) \leftarrow \text{SquareRoot}(ST(0));$

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format. Source operand is a negative value (except for -0).
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FST/FSTP—Store Floating Point Value

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 /2	FST <i>m32fp</i>	Valid	Valid	Copy ST(0) to <i>m32fp</i> .
DD /2	FST <i>m64fp</i>	Valid	Valid	Copy ST(0) to <i>m64fp</i> .
DD D0+i	FST ST(i)	Valid	Valid	Copy ST(0) to ST(i).
D9 /3	FSTP <i>m32fp</i>	Valid	Valid	Copy ST(0) to <i>m32fp</i> and pop register stack.
DD /3	FSTP <i>m64fp</i>	Valid	Valid	Copy ST(0) to <i>m64fp</i> and pop register stack.
DB /7	FSTP <i>m80fp</i>	Valid	Valid	Copy ST(0) to <i>m80fp</i> and pop register stack.
DD D8+i	FSTP ST(i)	Valid	Valid	Copy ST(0) to ST(i) and pop register stack.

Description

The FST instruction copies the value in the ST(0) register to the destination operand, which can be a memory location or another register in the FPU register stack. When storing the value in memory, the value is converted to single-precision or double-precision floating-point format.

The FSTP instruction performs the same operation as the FST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FSTP instruction can also store values in memory in double extended-precision floating-point format.

If the destination operand is a memory location, the operand specifies the address where the first byte of the destination value is to be stored. If the destination operand is a register, the operand specifies a register in the register stack relative to the top of the stack.

If the destination size is single-precision or double-precision, the significand of the value being stored is rounded to the width of the destination (according to the rounding mode specified by the RC field of the FPU control word), and the exponent is converted to the width and bias of the destination format. If the value being stored is too large for the destination format, a numeric overflow exception (#O) is generated and, if the exception is unmasked, no value is stored in the destination operand. If the value being stored is a denormal value, the denormal exception (#D) is not generated. This condition is simply signaled as a numeric underflow exception (#U) condition.

If the value being stored is ± 0 , $\pm \infty$, or a NaN, the least-significant bits of the significand and the exponent are truncated to fit the destination format. This operation preserves the value's identity as a 0, ∞ , or NaN.

If the destination operand is a non-empty register, the invalid-operation exception is not generated.

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

Operation

```

DEST ← ST(0);
IF Instruction = FSTP
  THEN
    PopRegisterStack;
  FI;

```

FPU Flags Affected

- | | |
|------------|--|
| C1 | Set to 0 if stack underflow occurred. |
| | Indicates rounding direction of if the floating-point inexact exception (#P) is generated: 0 ← not roundup; 1 ← roundup. |
| C0, C2, C3 | Undefined. |

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	If destination result is an SNaN value or unsupported format, except when the destination format is in double extended-precision floating-point format.
#U	Result is too small for the destination format.
#O	Result is too large for the destination format.
#P	Value cannot be represented exactly in destination format.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FSTCW/FNSTCW—Store x87 FPU Control Word

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B D9 /7	FSTCW <i>m2byte</i>	Valid	Valid	Store FPU control word to <i>m2byte</i> after checking for pending unmasked floating-point exceptions.
D9 /7	FNSTCW* <i>m2byte</i>	Valid	Valid	Store FPU control word to <i>m2byte</i> without checking for pending unmasked floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Stores the current value of the FPU control word at the specified destination in memory. The FSTCW instruction checks for and handles pending unmasked floating-point exceptions before storing the control word; the FNSTCW instruction does not.

The assembler issues two instructions for the FSTCW instruction (an FWAIT instruction followed by an FNSTCW instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTCW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSTCW instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

Operation

DEST ← FPUControlWord;

FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FSTENV/FNSTENV—Store x87 FPU Environment

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B D9 /6	FSTENV <i>m14/28byte</i>	Valid	Valid	Store FPU environment to <i>m14byte</i> or <i>m28byte</i> after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.
D9 /6	FNSTENV* <i>m14/28byte</i>	Valid	Valid	Store FPU environment to <i>m14byte</i> or <i>m28byte</i> without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FSTENV instruction checks for and handles any pending unmasked floating-point exceptions before storing the FPU environment; the FNSTENV instruction does not. The saved image reflects the state of the FPU after all floating-point instructions preceding the FSTENV/FNSTENV instruction in the instruction stream have been executed.

These instructions are often used by exception handlers because they provide access to the FPU instruction and data pointers. The environment is typically saved in the stack. Masking all exceptions after saving the environment prevents floating-point exceptions from interrupting the exception handler.

The assembler issues two instructions for the FSTENV instruction (an FWAIT instruction followed by an FNSTENV instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTENV instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNSTENV instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

Operation

```
DEST[FPUControlWord] ← FPUCtrlWord;
DEST[FPUStatusWord] ← FPUStrStatusWord;
DEST[FPUTagWord] ← FPUTagWord;
DEST[FPUDataPointer] ← FPUDataPointer;
DEST[FPUIstructionPointer] ← FPUIstructionPointer;
DEST[FPULastInstructionOpcode] ← FPULastInstructionOpcode;
```

FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

Floating-Point Exceptions

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FSTSW/FNSTSW—Store x87 FPU Status Word

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B DD /7	FSTSW <i>m2byte</i>	Valid	Valid	Store FPU status word at <i>m2byte</i> after checking for pending unmasked floating-point exceptions.
9B DF E0	FSTSW AX	Valid	Valid	Store FPU status word in AX register after checking for pending unmasked floating-point exceptions.
DD /7	FNSTSW* <i>m2byte</i>	Valid	Valid	Store FPU status word at <i>m2byte</i> without checking for pending unmasked floating-point exceptions.
DF E0	FNSTSW* AX	Valid	Valid	Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Stores the current value of the x87 FPU status word in the destination location. The destination operand can be either a two-byte memory location or the AX register. The FSTSW instruction checks for and handles pending unmasked floating-point exceptions before storing the status word; the FNSTSW instruction does not.

The FNSTSW AX form of the instruction is used primarily in conditional branching (for instance, after an FPU comparison instruction or an FPREM, FPREM1, or FXAM instruction), where the direction of the branch depends on the state of the FPU condition code flags. (See the section titled “Branching and Conditional Moves on FPU Condition Codes” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.) This instruction can also be used to invoke exception handlers (by examining the exception flags) in environments that do not use interrupts. When the FNSTSW AX instruction is executed, the AX register is updated before the processor executes any further instructions. The status stored in the AX register is thus guaranteed to be from the completion of the prior FPU instruction.

The assembler issues two instructions for the FSTSW instruction (an FWAIT instruction followed by an FNSTSW instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTSW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNSTSW instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

Operation

DEST ← FPUStatusWord;

FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

Floating-Point Exceptions

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FSUB/FSUBP/FISUB—Subtract

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /4	FSUB <i>m32fp</i>	Valid	Valid	Subtract <i>m32fp</i> from ST(0) and store result in ST(0).
DC /4	FSUB <i>m64fp</i>	Valid	Valid	Subtract <i>m64fp</i> from ST(0) and store result in ST(0).
D8 E0+i	FSUB ST(0), ST(i)	Valid	Valid	Subtract ST(i) from ST(0) and store result in ST(0).
DC E8+i	FSUB ST(i), ST(0)	Valid	Valid	Subtract ST(0) from ST(i) and store result in ST(i).
DE E8+i	FSUBP ST(i), ST(0)	Valid	Valid	Subtract ST(0) from ST(i), store result in ST(i), and pop register stack.
DE E9	FSUBP	Valid	Valid	Subtract ST(0) from ST(1), store result in ST(1), and pop register stack.
DA /4	FISUB <i>m32int</i>	Valid	Valid	Subtract <i>m32int</i> from ST(0) and store result in ST(0).
DE /4	FISUB <i>m16int</i>	Valid	Valid	Subtract <i>m16int</i> from ST(0) and store result in ST(0).

Description

Subtracts the source operand from the destination operand and stores the difference in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction subtracts the contents of the ST(0) register from the ST(1) register and stores the result in ST(1). The one-operand version subtracts the contents of a memory location (either a floating-point or an integer value) from the contents of the ST(0) register and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(0) register from the ST(i) register or vice versa.

The FSUBP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUB rather than FSUBP.

The FISUB instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

Table 3-48 shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the SRC value is subtracted from the DEST value (DEST – SRC = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward $-\infty$ mode, in which case the result is -0. This instruction also guarantees that $+0 - (-0) = +0$, and that $-0 - (+0) = -0$. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞ , the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

Table 3-48. FSUB/FSUBP/FISUB Results

		SRC						
		– ∞	– F or – I	– 0	+ 0	+ F or + I	+ ∞	NaN
DEST	– ∞	*	– ∞	– ∞	– ∞	– ∞	– ∞	NaN
	– F	+ ∞	±F or ±0	DEST	DEST	– F	– ∞	NaN
	– 0	+ ∞	–SRC	±0	– 0	– SRC	– ∞	NaN
	+ 0	+ ∞	–SRC	+ 0	±0	– SRC	– ∞	NaN
	+ F	+ ∞	+ F	DEST	DEST	±F or ±0	– ∞	NaN
	+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

IF Instruction = FISUB

THEN

DEST ← DEST – Convert.ToDoubleExtendedPrecisionFP(SRC);

ELSE (* Source operand is floating-point value *)

DEST ← DEST – SRC;

Fl;

IF Instruction = FSUBP

THEN

PopRegisterStack;

Fl;

FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.
 Set if result was rounded up; cleared otherwise.
 C0, C2, C3 Undefined.

Floating-Point Exceptions

- #IS Stack underflow occurred.
 #IA Operand is an SNaN value or unsupported format.
 Operands are infinities of like sign.
 #D Source operand is a denormal value.
 #U Result is too small for destination format.
 #O Result is too large for destination format.
 #P Value cannot be represented exactly in destination format.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FSUBR/FSUBRP/FISUBR—Reverse Subtract

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /5	FSUBR <i>m32fp</i>	Valid	Valid	Subtract ST(0) from <i>m32fp</i> and store result in ST(0).
DC /5	FSUBR <i>m64fp</i>	Valid	Valid	Subtract ST(0) from <i>m64fp</i> and store result in ST(0).
D8 E8+i	FSUBR ST(0), ST(i)	Valid	Valid	Subtract ST(0) from ST(i) and store result in ST(0).
DC E0+i	FSUBR ST(i), ST(0)	Valid	Valid	Subtract ST(i) from ST(0) and store result in ST(i).
DE E0+i	FSUBRP ST(i), ST(0)	Valid	Valid	Subtract ST(i) from ST(0), store result in ST(i), and pop register stack.
DE E1	FSUBRP	Valid	Valid	Subtract ST(1) from ST(0), store result in ST(1), and pop register stack.
DA /5	FISUBR <i>m32int</i>	Valid	Valid	Subtract ST(0) from <i>m32int</i> and store result in ST(0).
DE /5	FISUBR <i>m16int</i>	Valid	Valid	Subtract ST(0) from <i>m16int</i> and store result in ST(0).

Description

Subtracts the destination operand from the source operand and stores the difference in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

These instructions perform the reverse operations of the FSUB, FSUBP, and FISUB instructions. They are provided to support more efficient coding.

The no-operand version of the instruction subtracts the contents of the ST(1) register from the ST(0) register and stores the result in ST(1). The one-operand version subtracts the contents of the ST(0) register from the contents of a memory location (either a floating-point or an integer value) and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(i) register from the ST(0) register or vice versa.

The FSUBRP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point reverse subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUBR rather than FSUBRP.

The FISUBR instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the DEST value is subtracted from the SRC value (SRC – DEST = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward $-\infty$ mode, in which case the result is -0. This instruction also guarantees that $+0 - (-0) = +0$, and that $-0 - (+0) = -0$. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞ , the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

Table 3-49. FSUBR/FSUBRP/FISUBR Results

		SRC						
		–∞	–F or –I	–0	+0	+F or +I	+∞	NaN
DEST	–∞	*	+∞	+∞	+∞	+∞	+∞	NaN
	–F	–∞	±F or ±0	–DEST	–DEST	+F	+∞	NaN
	–0	–∞	SRC	±0	+0	SRC	+∞	NaN
	+0	–∞	SRC	–0	±0	SRC	+∞	NaN
	+F	–∞	–F	–DEST	–DEST	±F or ±0	+∞	NaN
	+∞	–∞	–∞	–∞	–∞	–∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

IF Instruction = FISUBR

THEN

DEST ← Convert.ToDoubleExtendedPrecisionFP(SRC) – DEST;

ELSE (* Source operand is floating-point value *)

DEST ← SRC – DEST; Fl;

IF Instruction = FSUBRP

THEN

PopRegisterStack; Fl;

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA Operand is an SNaN value or unsupported format.

Operands are infinities of like sign.

#D Source operand is a denormal value.

#U Result is too small for destination format.

#O Result is too large for destination format.

#P Value cannot be represented exactly in destination format.

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.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#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.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

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.
#NM	CR0.EM[bit 2] or 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.
#UD	If the LOCK prefix is used.

FTST—TEST

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 E4	FTST	Valid	Valid	Compare ST(0) with 0.0.

Description

Compares the value in the ST(0) register with 0.0 and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below).

Table 3-50. FTST Results

Condition	C3	C2	C0
ST(0) > 0.0	0	0	0
ST(0) < 0.0	0	0	1
ST(0) = 0.0	1	0	0
Unordered	1	1	1

This instruction performs an “unordered comparison.” An unordered comparison also checks the class of the numbers being compared (see “FXAM—Examine ModR/M” in this chapter). If the value in register ST(0) is a NaN or is in an undefined format, the condition flags are set to “unordered” and the invalid operation exception is generated.

The sign of zero is ignored, so that $(-0.0 \leftarrow +0.0)$.

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

Operation

CASE (relation of operands) OF

Not comparable: C3, C2, C0 $\leftarrow 111$;
 ST(0) > 0.0: C3, C2, C0 $\leftarrow 000$;
 ST(0) < 0.0: C3, C2, C0 $\leftarrow 001$;
 ST(0) = 0.0: C3, C2, C0 $\leftarrow 100$;

ESAC;

FPU Flags Affected

C1 Set to 0.
 C0, C2, C3 See Table 3-50.

Floating-Point Exceptions

#IS Stack underflow occurred.
 #IA The source operand is a NaN value or is in an unsupported format.
 #D The source operand is a denormal value.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
 #MF If there is a pending x87 FPU exception.
 #UD If the LOCK prefix is used.

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.

FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DD E0+i	FUCOM ST(i)	Valid	Valid	Compare ST(0) with ST(i).
DD E1	FUCOM	Valid	Valid	Compare ST(0) with ST(1).
DD E8+i	FUCOMP ST(i)	Valid	Valid	Compare ST(0) with ST(i) and pop register stack.
DD E9	FUCOMP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack.
DA E9	FUCOMPP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack twice.

Description

Performs an unordered comparison of the contents of register ST(0) and ST(i) and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). If no operand is specified, the contents of registers ST(0) and ST(1) are compared. The sign of zero is ignored, so that -0.0 is equal to +0.0.

Table 3-51. FUCOM/FUCOMP/FUCOMPP Results

Comparison Results*	C3	C2	C0
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered	1	1	1

NOTES:

* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see “FXAM—Examine ModR/M” in this chapter). The FUCOM/FUCOMP/FUCOMPP instructions perform the same operations as the FCOM/FCOMP/FCOMPP instructions. The only difference is that the FUCOM/FUCOMP/FUCOMPP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOM/FCOMP/FCOMPP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

As with the FCOM/FCOMP/FCOMPP instructions, if the operation results in an invalid-arithmetic-operand exception being raised, the condition code flags are set only if the exception is masked.

The FUCOMP instruction pops the register stack following the comparison operation and the FUCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

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

Operation

CASE (relation of operands) OF

ST > SRC: C3, C2, C0 ← 000;
 ST < SRC: C3, C2, C0 ← 001;
 ST = SRC: C3, C2, C0 ← 100;

ESAC;

IF ST(0) or SRC = QNaN, but not SNaN or unsupported format

THEN

 C3, C2, C0 ← 111;

ELSE (* ST(0) or SRC is SNaN or unsupported format *)

 #IA;

```

IF FPUControlWord.IM = 1
  THEN
    C3, C2, C0 ← 111;
  Fl;
Fl;

IF Instruction = FUCOMP
  THEN
    PopRegisterStack;
Fl;

IF Instruction = FUCOMPP
  THEN
    PopRegisterStack;
Fl;

```

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.
 C0, C2, C3 See Table 3-51.

Floating-Point Exceptions

#IS Stack underflow occurred.
 #IA One or both operands are SNaN values or have unsupported formats. Detection of a QNaN value in and of itself does not raise an invalid-operand exception.
 #D One or both operands are denormal values.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
 #MF If there is a pending x87 FPU exception.
 #UD If the LOCK prefix is used.

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.

FXAM—Examine ModR/M

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 E5	FXAM	Valid	Valid	Classify value or number in ST(0).

Description

Examines the contents of the ST(0) register and sets the condition code flags C0, C2, and C3 in the FPU status word to indicate the class of value or number in the register (see the table below).

Table 3-52. FXAM Results

Class	C3	C2	C0
Unsupported	0	0	0
NaN	0	0	1
Normal finite number	0	1	0
Infinity	0	1	1
Zero	1	0	0
Empty	1	0	1
Denormal number	1	1	0

The C1 flag is set to the sign of the value in ST(0), regardless of whether the register is empty or full.

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

Operation

$C1 \leftarrow \text{sign bit of ST}; (* 0 for positive, 1 for negative *)$

CASE (class of value or number in ST(0)) OF

```

Unsupported: C3, C2, C0 ← 000;
NaN:          C3, C2, C0 ← 001;
Normal:       C3, C2, C0 ← 010;
Infinity:     C3, C2, C0 ← 011;
Zero:          C3, C2, C0 ← 100;
Empty:         C3, C2, C0 ← 101;
Denormal:      C3, C2, C0 ← 110;
ESAC;
```

FPU Flags Affected

C1 Sign of value in ST(0).

C0, C2, C3 See Table 3-52.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

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.

FXCH—Exchange Register Contents

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 C8+i	FXCH ST(i)	Valid	Valid	Exchange the contents of ST(0) and ST(i).
D9 C9	FXCH	Valid	Valid	Exchange the contents of ST(0) and ST(1).

Description

Exchanges the contents of registers ST(0) and ST(i). If no source operand is specified, the contents of ST(0) and ST(1) are exchanged.

This instruction provides a simple means of moving values in the FPU register stack to the top of the stack [ST(0)], so that they can be operated on by those floating-point instructions that can only operate on values in ST(0). For example, the following instruction sequence takes the square root of the third register from the top of the register stack:

```
FXCH ST(3);
FSQRT;
FXCH ST(3);
```

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

Operation

IF (Number-of-operands) is 1

THEN

```
temp ← ST(0);
ST(0) ← SRC;
SRC ← temp;
```

ELSE

```
temp ← ST(0);
ST(0) ← ST(1);
ST(1) ← temp;
```

FI;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, set to 1.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
-----	---------------------------

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FXRSTOR—Restore x87 FPU, MMX , XMM, and MXCSR State

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF AE /1	M	Valid	Valid	Restore the x87 FPU, MMX, XMM, and MXCSR register state from <i>m512byte</i> .
FXRSTOR <i>m512byte</i>				
REX.W+ OF AE /1	M	Valid	N.E.	Restore the x87 FPU, MMX, XMM, and MXCSR register state from <i>m512byte</i> .
FXRSTOR64 <i>m512byte</i>				

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Reloads the x87 FPU, MMX technology, XMM, and MXCSR registers from the 512-byte memory image specified in the source operand. This data should have been written to memory previously using the FXSAVE instruction, and in the same format as required by the operating modes. The first byte of the data should be located on a 16-byte boundary. There are three distinct layouts of the FXSAVE state map: one for legacy and compatibility mode, a second format for 64-bit mode FXSAVE/FXRSTOR with REX.W=0, and the third format is for 64-bit mode with FXSAVE64/FXRSTOR64. Table 3-53 shows the layout of the legacy/compatibility mode state information in memory and describes the fields in the memory image for the FXRSTOR and FXSAVE instructions. Table 3-56 shows the layout of the 64-bit mode state information when REX.W is set (FXSAVE64/FXRSTOR64). Table 3-57 shows the layout of the 64-bit mode state information when REX.W is clear (FXSAVE/FXRSTOR).

The state image referenced with an FXRSTOR instruction must have been saved using an FXSAVE instruction or be in the same format as required by Table 3-53, Table 3-56, or Table 3-57. Referencing a state image saved with an FSAVE, FNSAVE instruction or incompatible field layout will result in an incorrect state restoration.

The FXRSTOR instruction does not flush pending x87 FPU exceptions. To check and raise exceptions when loading x87 FPU state information with the FXRSTOR instruction, use an FWAIT instruction after the FXRSTOR instruction.

If the OSFXSR bit in control register CR4 is not set, the FXRSTOR instruction may not restore the states of the XMM and MXCSR registers. This behavior is implementation dependent.

If the MXCSR state contains an unmasked exception with a corresponding status flag also set, loading the register with the FXRSTOR instruction will not result in a SIMD floating-point error condition being generated. Only the next occurrence of this unmasked exception will result in the exception being generated.

Bits 16 through 32 of the MXCSR register are defined as reserved and should be set to 0. Attempting to write a 1 in any of these bits from the saved state image will result in a general protection exception (#GP) being generated.

Bytes 464:511 of an FXSAVE image are available for software use. FXRSTOR ignores the content of bytes 464:511 in an FXSAVE state image.

Operation

(x87 FPU, MMX, XMM7-XMM0, MXCSR) ← Load(SRC);

x87 FPU and 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. (See alignment check exception [#AC] below.) |
| | For an attempt to set reserved bits in MXCSR. |

#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If instruction is preceded by a LOCK prefix.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	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. For an attempt to set reserved bits in MXCSR.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.
#UD	If the LOCK prefix is used.

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. For an attempt to set reserved bits in MXCSR.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If instruction is preceded by a LOCK prefix.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a

general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

FXSAVE—Save x87 FPU, MMX Technology, and SSE State

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF AE /0	M	Valid	Valid	Save the x87 FPU, MMX, XMM, and MXCSR register state to <i>m512byte</i> .
FXSAVE <i>m512byte</i>				
REX.W+ OF AE /0	M	Valid	N.E.	Save the x87 FPU, MMX, XMM, and MXCSR register state to <i>m512byte</i> .
FXSAVE64 <i>m512byte</i>				

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Saves the current state of the x87 FPU, MMX technology, XMM, and MXCSR registers to a 512-byte memory location specified in the destination operand. The content layout of the 512 byte region depends on whether the processor is operating in non-64-bit operating modes or 64-bit sub-mode of IA-32e mode.

Bytes 464:511 are available to software use. The processor does not write to bytes 464:511 of an FXSAVE area. The operation of FXSAVE in non-64-bit modes is described first.

Non-64-Bit Mode Operation

Table 3-53 shows the layout of the state information in memory when the processor is operating in legacy modes.

Table 3-53. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR Memory Region

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Rsvd	FPU CS			FPU IP		FOP	Rsvd	FTW		FSW		FCW		0		
MXCSR_MASK				MXCSR		Rsvd		FPU DS			FPU DP			16		
Reserved								ST0/MM0						32		
Reserved								ST1/MM1						48		
Reserved								ST2/MM2						64		
Reserved								ST3/MM3						80		
Reserved								ST4/MM4						96		
Reserved								ST5/MM5						112		
Reserved								ST6/MM6						128		
Reserved								ST7/MM7						144		
						XMM0								160		
						XMM1								176		
						XMM2								192		
						XMM3								208		
						XMM4								224		
						XMM5								240		
						XMM6								256		
						XMM7								272		

Table 3-53. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR Memory Region (Contd.)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved															288	
Reserved															304	
Reserved															320	
Reserved															336	
Reserved															352	
Reserved															368	
Reserved															384	
Reserved															400	
Reserved															416	
Reserved															432	
Reserved															448	
Available															464	
Available															480	
Available															496	

The destination operand contains the first byte of the memory image, and it must be aligned on a 16-byte boundary. A misaligned destination operand will result in a general-protection (#GP) exception being generated (or in some cases, an alignment check exception [#AC]).

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to save and examine the current state of the x87 FPU, MMX technology, and/or XMM and MXCSR registers.

The fields in Table 3-53 are defined in Table 3-54.

Table 3-54. Field Definitions

Field	Definition
FCW	x87 FPU Control Word (16 bits). See Figure 8-6 in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for the layout of the x87 FPU control word.
FSW	x87 FPU Status Word (16 bits). See Figure 8-4 in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for the layout of the x87 FPU status word.
Abridged FTW	x87 FPU Tag Word (8 bits). The tag information saved here is abridged, as described in the following paragraphs.
FOP	x87 FPU Opcode (16 bits). The lower 11 bits of this field contain the opcode, upper 5 bits are reserved. See Figure 8-8 in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for the layout of the x87 FPU opcode field.
FPU IP	x87 FPU Instruction Pointer Offset (32 bits). The contents of this field differ depending on the current addressing mode (32-bit or 16-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode – 32-bit IP offset. 16-bit mode – low 16 bits are IP offset; high 16 bits are reserved. See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for a description of the x87 FPU instruction pointer.
FPU CS	x87 FPU Instruction Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates the FPU CS and FPU DS values, and this field is saved as 0000H.

Table 3-54. Field Definitions (Contd.)

Field	Definition
FPU DP	x87 FPU Instruction Operand (Data) Pointer Offset (32 bits). The contents of this field differ depending on the current addressing mode (32-bit or 16-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit DP offset. 16-bit mode — low 16 bits are DP offset; high 16 bits are reserved. See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for a description of the x87 FPU operand pointer.
FPU DS	x87 FPU Instruction Operand (Data) Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates the FPU CS and FPU DS values, and this field is saved as 0000H.
MXCSR	MXCSR Register State (32 bits). See Figure 10-3 in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for the layout of the MXCSR register. If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save this register. This behavior is implementation dependent.
MXCSR_MASK	MXCSR_MASK (32 bits). This mask can be used to adjust values written to the MXCSR register, ensuring that reserved bits are set to 0. Set the mask bits and flags in MXCSR to the mode of operation desired for SSE and SSE2 SIMD floating-point instructions. See “Guidelines for Writing to the MXCSR Register” in Chapter 11 of the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for instructions for how to determine and use the MXCSR_MASK value.
ST0/MM0 through ST7/MM7	x87 FPU or MMX technology registers. These 80-bit fields contain the x87 FPU data registers or the MMX technology registers, depending on the state of the processor prior to the execution of the FXSAVE instruction. If the processor had been executing x87 FPU instruction prior to the FXSAVE instruction, the x87 FPU data registers are saved; if it had been executing MMX instructions (or SSE or SSE2 instructions that operated on the MMX technology registers), the MMX technology registers are saved. When the MMX technology registers are saved, the high 16 bits of the field are reserved.
XMM0 through XMM7	XMM registers (128 bits per field). If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save these registers. This behavior is implementation dependent.

The FXSAVE instruction saves an abridged version of the x87 FPU tag word in the FTW field (unlike the FSAVE instruction, which saves the complete tag word). The tag information is saved in physical register order (R0 through R7), rather than in top-of-stack (TOS) order. With the FXSAVE instruction, however, only a single bit (1 for valid or 0 for empty) is saved for each tag. For example, assume that the tag word is currently set as follows:

R7	R6	R5	R4	R3	R2	R1	R0
11	xx	xx	xx	11	11	11	11

Here, 11B indicates empty stack elements and “xx” indicates valid (00B), zero (01B), or special (10B).

For this example, the FXSAVE instruction saves only the following 8 bits of information:

R7	R6	R5	R4	R3	R2	R1	R0
0	1	1	1	0	0	0	0

Here, a 1 is saved for any valid, zero, or special tag, and a 0 is saved for any empty tag.

The operation of the FXSAVE instruction differs from that of the FSAVE instruction, the as follows:

- FXSAVE instruction does not check for pending unmasked floating-point exceptions. (The FXSAVE operation in this regard is similar to the operation of the FNSAVE instruction).
- After the FXSAVE instruction has saved the state of the x87 FPU, MMX technology, XMM, and MXCSR registers, the processor retains the contents of the registers. Because of this behavior, the FXSAVE instruction cannot be used by an application program to pass a “clean” x87 FPU state to a procedure, since it retains the current state. To clean the x87 FPU state, an application must explicitly execute an FINIT instruction after an FXSAVE instruction to reinitialize the x87 FPU state.
- The format of the memory image saved with the FXSAVE instruction is the same regardless of the current addressing mode (32-bit or 16-bit) and operating mode (protected, real address, or system management).

This behavior differs from the FSAVE instructions, where the memory image format is different depending on the addressing mode and operating mode. Because of the different image formats, the memory image saved with the FXSAVE instruction cannot be restored correctly with the FRSTOR instruction, and likewise the state saved with the FSAVE instruction cannot be restored correctly with the FXRSTOR instruction.

The FSAVE format for FTW can be recreated from the FTW valid bits and the stored 80-bit FP data (assuming the stored data was not the contents of MMX technology registers) using Table 3-55.

Table 3-55. Recreating FSAVE Format

Exponent all 1's	Exponent all 0's	Fraction all 0's	J and M bits	FTW valid bit	x87 FTW	
0	0	0	0x	1	Special	10
0	0	0	1x	1	Valid	00
0	0	1	00	1	Special	10
0	0	1	10	1	Valid	00
0	1	0	0x	1	Special	10
0	1	0	1x	1	Special	10
0	1	1	00	1	Zero	01
0	1	1	10	1	Special	10
1	0	0	1x	1	Special	10
1	0	0	1x	1	Special	10
1	0	1	00	1	Special	10
1	0	1	10	1	Special	10
For all legal combinations above.				0	Empty	11

The J-bit is defined to be the 1-bit binary integer to the left of the decimal place in the significand. The M-bit is defined to be the most significant bit of the fractional portion of the significand (i.e., the bit immediately to the right of the decimal place).

When the M-bit is the most significant bit of the fractional portion of the significand, it must be 0 if the fraction is all 0's.

IA-32e Mode Operation

In compatibility sub-mode of IA-32e mode, legacy SSE registers, XMM0 through XMM7, are saved according to the legacy FXSAVE map. In 64-bit mode, all of the SSE registers, XMM0 through XMM15, are saved. Additionally, there are two different layouts of the FXSAVE map in 64-bit mode, corresponding to FXSAVE64 (which requires REX.W=1) and FXSAVE (REX.W=0). In the FXSAVE64 map (Table 3-56), the FPU IP and FPU DP pointers are 64-bit wide. In the FXSAVE map for 64-bit mode (Table 3-57), the FPU IP and FPU DP pointers are 32-bits.

**Table 3-56. Layout of the 64-bit-mode FXSAVE64 Map
(requires REX.W = 1)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
FPU IP				FOP	Reserved	FTW	FSW	FCW								0
MXCSR_MASK		MXCSR				FPU DP										16
Reserved				ST0/MM0												32
Reserved				ST1/MM1												48
Reserved				ST2/MM2												64
Reserved				ST3/MM3												80
Reserved				ST4/MM4												96
Reserved				ST5/MM5												112

**Table 3-56. Layout of the 64-bit-mode FXSAVE64 Map
(requires REX.W = 1) (Contd.)**

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0										
Reserved			ST6/MM6								128							
Reserved			ST7/MM7								144							
XMM0												160						
XMM1												176						
XMM2												192						
XMM3												208						
XMM4												224						
XMM5												240						
XMM6												256						
XMM7												272						
XMM8												288						
XMM9												304						
XMM10												320						
XMM11												336						
XMM12												352						
XMM13												368						
XMM14												384						
XMM15												400						
Reserved												416						
Reserved												432						
Reserved												448						
Available												464						
Available												480						
Available												496						

Table 3-57. Layout of the 64-bit-mode FXSAVE Map (REX.W = 0)

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0							
Reserved	FPU CS	FPU IP			FOP	Reserved	FTW	FSW	FCW	0					
MXCSR_MASK		MXCSR			Reserved	FPU DS		FPU DP		16					
Reserved										32					
Reserved										48					
Reserved										64					
Reserved										80					
Reserved										96					
Reserved										112					
Reserved										128					
Reserved										144					
XMM0										160					

Table 3-57. Layout of the 64-bit-mode FXSAVE Map (REX.W = 0) (Contd.) (Contd.)

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
XMM1									176
XMM2									192
XMM3									208
XMM4									224
XMM5									240
XMM6									256
XMM7									272
XMM8									288
XMM9									304
XMM10									320
XMM11									336
XMM12									352
XMM13									368
XMM14									384
XMM15									400
Reserved									416
Reserved									432
Reserved									448
Available									464
Available									480
Available									496

Operation

```

IF 64-Bit Mode
  THEN
    IF REX.W = 1
      THEN
        DEST ← Save64BitPromotedFxsave(x87 FPU, MMX, XMM7-XMM0,
                                         MXCSR);
      ELSE
        DEST ← Save64BitDefaultFxsave(x87 FPU, MMX, XMM7-XMM0, MXCSR);
      FI;
    ELSE
      DEST ← SaveLegacyFxsave(x87 FPU, MMX, XMM7-XMM0, MXCSR);
    FI;
  
```

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. (See the description of the alignment check exception [#AC] below.)
- #SS(0) For an illegal address in the SS segment.
- #PF(fault-code) For a page fault.

#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0.
#UD	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	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. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.
#UD	If the LOCK prefix is used.

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. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Implementation Note

The order in which the processor signals general-protection (#GP) and page-fault (#PF) exceptions when they both occur on an instruction boundary is given in Table 5-2 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. This order vary for FXSAVE for different processor implementations.

FXTRACT—Extract Exponent and Significand

Opcode/ Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F4 FXTRACT	Valid	Valid	Separate value in ST(0) into exponent and significand, store exponent in ST(0), and push the significand onto the register stack.

Description

Separates the source value in the ST(0) register into its exponent and significand, stores the exponent in ST(0), and pushes the significand onto the register stack. Following this operation, the new top-of-stack register ST(0) contains the value of the original significand expressed as a floating-point value. The sign and significand of this value are the same as those found in the source operand, and the exponent is 3FFFH (biased value for a true exponent of zero). The ST(1) register contains the value of the original operand's true (unbiased) exponent expressed as a floating-point value. (The operation performed by this instruction is a superset of the IEEE-recommended $\log_2(x)$ function.)

This instruction and the F2XM1 instruction are useful for performing power and range scaling operations. The FXTRACT instruction is also useful for converting numbers in double extended-precision floating-point format to decimal representations (e.g., for printing or displaying).

If the floating-point zero-divide exception (#Z) is masked and the source operand is zero, an exponent value of $-\infty$ is stored in register ST(1) and 0 with the sign of the source operand is stored in register ST(0).

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

Operation

```
TEMP ← Significand(ST(0));
ST(0) ← Exponent(ST(0));
TOP ← TOP - 1;
ST(0) ← TEMP;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#Z	ST(0) operand is ± 0 .
#D	Source operand is a denormal value.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FYL2X—Compute $y * \log_2 x$

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F1	FYL2X	Valid	Valid	Replace ST(1) with $(ST(1) * \log_2 ST(0))$ and pop the register stack.

Description

Computes $(ST(1) * \log_2 (ST(0)))$, stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be a non-zero positive number.

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-58. FYL2X Results

		ST(0)							
		$-\infty$	$-F$	± 0	$+0 < +F < +1$	$+1$	$+F > +1$	$+\infty$	NaN
ST(1)	$-\infty$	*	*	$+\infty$	$+\infty$	*	$-\infty$	$-\infty$	NaN
	$-F$	*	*	**	$+F$	-0	$-F$	$-\infty$	NaN
	-0	*	*	*	$+0$	-0	-0	*	NaN
	$+0$	*	*	*	-0	$+0$	$+0$	*	NaN
	$+F$	*	*	**	$-F$	$+0$	$+F$	$+\infty$	NaN
	$+\infty$	*	*	$-\infty$	$-\infty$	*	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

If the divide-by-zero exception is masked and register ST(0) contains ± 0 , the instruction returns ∞ with a sign that is the opposite of the sign of the source operand in register ST(1).

The FYL2X instruction is designed with a built-in multiplication to optimize the calculation of logarithms with an arbitrary positive base (b):

$$\log_b x \leftarrow (\log_2 b)^{-1} * \log_2 x$$

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

Operation

$ST(1) \leftarrow ST(1) * \log_2 ST(0);$
PopRegisterStack;

FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.
C0, C2, C3 Undefined.

Floating-Point Exceptions

- #IS Stack underflow occurred.

#IA	Either operand is an SNaN or unsupported format.
	Source operand in register ST(0) is a negative finite value (not -0).
#Z	Source operand in register ST(0) is ± 0 .
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

FYL2XP1—Compute $y * \log_2(x + 1)$

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F9	FYL2XP1	Valid	Valid	Replace ST(1) with $ST(1) * \log_2(ST(0) + 1.0)$ and pop the register stack.

Description

Computes $(ST(1) * \log_2(ST(0) + 1.0))$, stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be in the range:

$$-(1 - \sqrt{2}/2) \text{ to } (1 - \sqrt{2}/2)$$

The source operand in ST(1) can range from $-\infty$ to $+\infty$. If the ST(0) operand is outside of its acceptable range, the result is undefined and software should not rely on an exception being generated. Under some circumstances exceptions may be generated when ST(0) is out of range, but this behavior is implementation specific and not guaranteed.

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that underflow does not occur.

Table 3-59. FYL2XP1 Results

		ST(0)				
		-(1 - ($\sqrt{2}/2$)) to -0	-0	+0	+0 to +(1 - ($\sqrt{2}/2$))	NaN
ST(1)	- ∞	+ ∞	*	*	- ∞	NaN
	- F	+F	+0	-0	- F	NaN
	- 0	+0	+0	-0	- 0	NaN
	+0	- 0	- 0	+0	+0	NaN
	+F	- F	- 0	+0	+F	NaN
	+ ∞	- ∞	*	*	+ ∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

This instruction provides optimal accuracy for values of epsilon [the value in register ST(0)] that are close to 0. For small epsilon (ϵ) values, more significant digits can be retained by using the FYL2XP1 instruction than by using $(\epsilon+1)$ as an argument to the FYL2X instruction. The $(\epsilon+1)$ expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in the ST(1) source operand. The following equation is used to calculate the scale factor for a particular logarithm base, where n is the logarithm base desired for the result of the FYL2XP1 instruction:

$$\text{scale factor} \leftarrow \log_n 2$$

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

Operation

$ST(1) \leftarrow ST(1) * \log_2(ST(0) + 1.0);$
PopRegisterStack;

FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

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.

HADDPD—Packed Double-FP Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7C /r HADDPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE3	Horizontal add packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 7C /r VHADDPD <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Horizontal add packed double-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 7C /r VHADDPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Horizontal add packed double-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

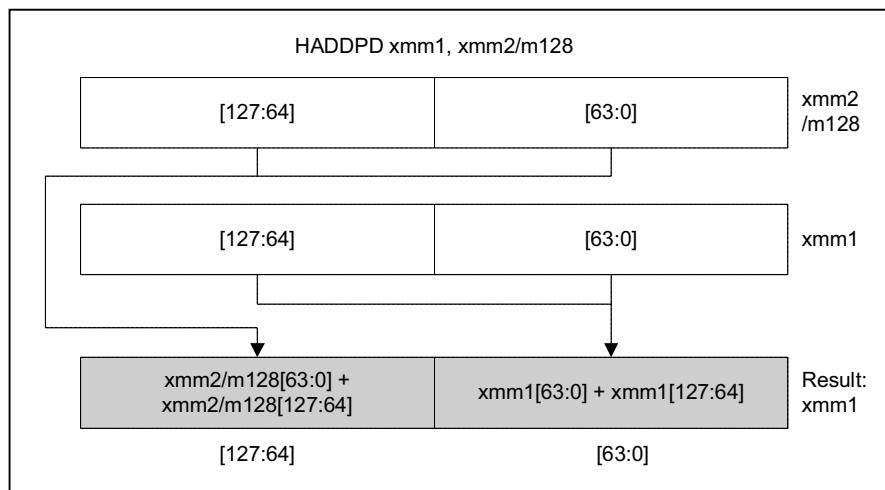
Description

Adds the double-precision floating-point values in the high and low quadwords of the destination operand and stores the result in the low quadword of the destination operand.

Adds the double-precision floating-point values in the high and low quadwords of the source operand and stores the result in the high quadword of the destination operand.

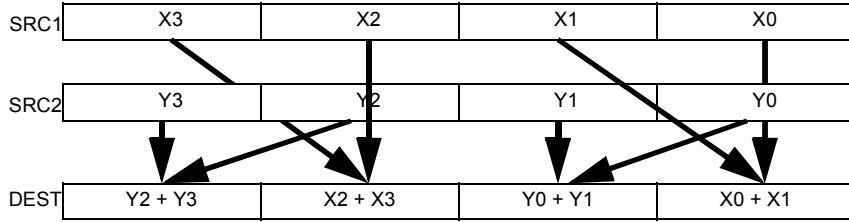
In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-15 for HADDPD; see Figure 3-16 for VHADDPD.



OM15993

Figure 3-15. HADDPD—Packed Double-FP Horizontal Add

**Figure 3-16. VHADDPD operation**

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

HADDPD (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[VLMAX-1:128] (Unmodified)
```

VHADDPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[VLMAX-1:128] ← 0
```

VHADDPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[191:128] ← SRC1[255:192] + SRC1[191:128]
DEST[255:192] ← SRC2[255:192] + SRC2[191:128]
```

Intel C/C++ Compiler Intrinsic Equivalent

VHADDPD: `__m256d _mm256_hadd_pd (__m256d a, __m256d b);`

HADDPD: `__m128d _mm_hadd_pd (__m128d a, __m128d b);`

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

HADDPS—Packed Single-FP Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 7C /r HADDPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE3	Horizontal add packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.F2.0F.WIG 7C /r VHADDPS <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Horizontal add packed single-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.F2.0F.WIG 7C /r VHADDPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Horizontal add packed single-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Adds the single-precision floating-point values in the first and second dwords of the destination operand and stores the result in the first dword of the destination operand.

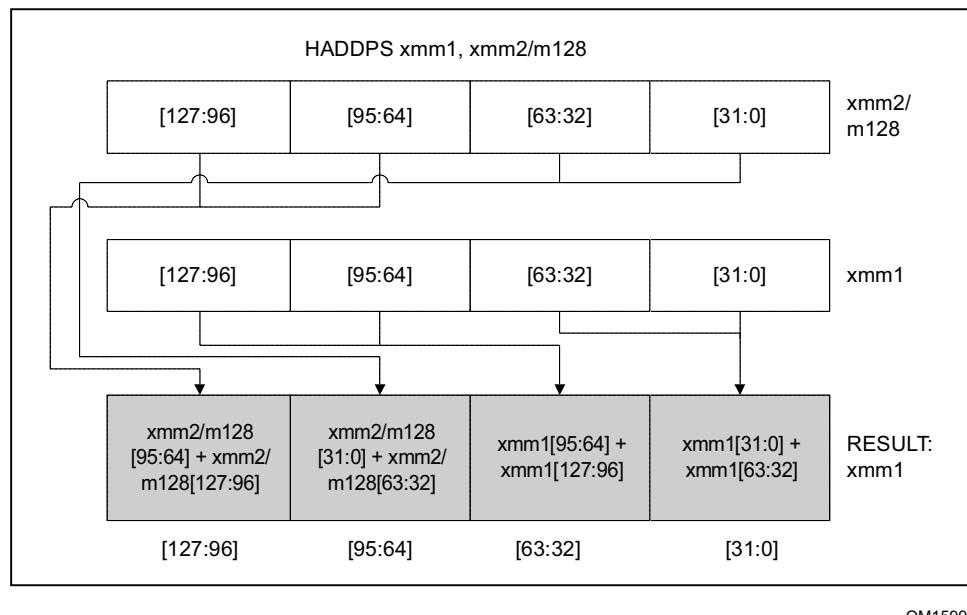
Adds single-precision floating-point values in the third and fourth dword of the destination operand and stores the result in the second dword of the destination operand.

Adds single-precision floating-point values in the first and second dword of the source operand and stores the result in the third dword of the destination operand.

Adds single-precision floating-point values in the third and fourth dword of the source operand and stores the result in the fourth dword of the destination operand.

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

See Figure 3-17 for HADDPS; see Figure 3-18 for VHADDPS.



OM15994

Figure 3-17. HADDPS—Packed Single-FP Horizontal Add

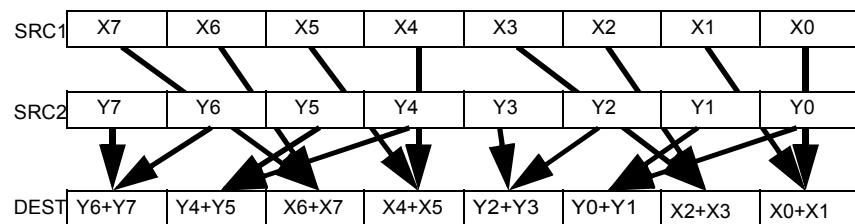


Figure 3-18. VHADDPS operation

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

HADDPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[63:32] + SRC1[31:0]
DEST[63:32] ← SRC1[127:96] + SRC1[95:64]
DEST[95:64] ← SRC2[63:32] + SRC2[31:0]
DEST[127:96] ← SRC2[127:96] + SRC2[95:64]
DEST[VLMAX-1:128] (Unmodified)
```

VHADDPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[63:32] + SRC1[31:0]
DEST[63:32] ← SRC1[127:96] + SRC1[95:64]
DEST[95:64] ← SRC2[63:32] + SRC2[31:0]
DEST[127:96] ← SRC2[127:96] + SRC2[95:64]
DEST[VLMAX-1:128] ← 0
```

VHADDPS (VEX.256 encoded version)

```
DEST[31:0] ← SRC1[63:32] + SRC1[31:0]
DEST[63:32] ← SRC1[127:96] + SRC1[95:64]
DEST[95:64] ← SRC2[63:32] + SRC2[31:0]
DEST[127:96] ← SRC2[127:96] + SRC2[95:64]
DEST[159:128] ← SRC1[191:160] + SRC1[159:128]
DEST[191:160] ← SRC1[255:224] + SRC1[223:192]
DEST[223:192] ← SRC2[191:160] + SRC2[159:128]
DEST[255:224] ← SRC2[255:224] + SRC2[223:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
HADDPS: __m128 _mm_hadd_ps (__m128 a, __m128 b);
VHADDPS: __m256 _mm256_hadd_ps (__m256 a, __m256 b);
```

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

HLT—Halt

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F4	HLT	NP	Valid	Valid	Halt

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

When a HLT instruction is executed on an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.

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

Operation

Enter Halt state;

Flags Affected

None.

Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

None.

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.

HSUBPD—Packed Double-FP Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7D /r HSUBPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE3	Horizontal subtract packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 7D /r VHSUBPD <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Horizontal subtract packed double-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 7D /r VHSUBPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Horizontal subtract packed double-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

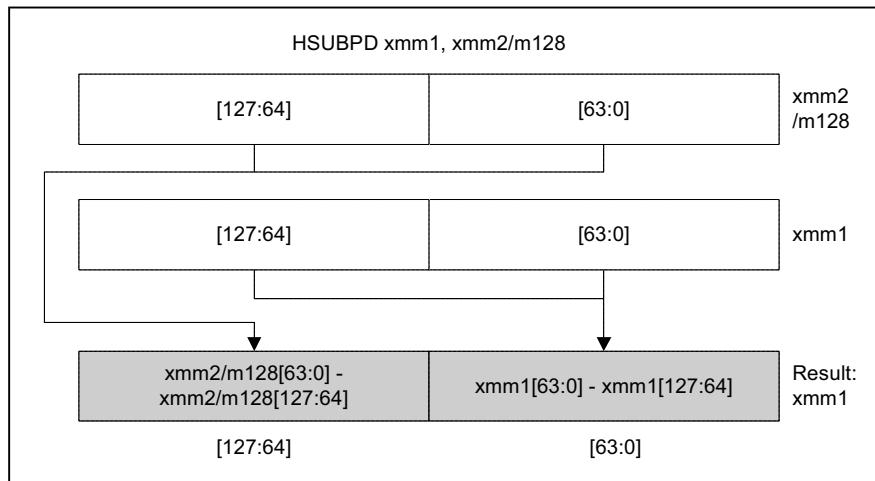
The HSUBPD instruction subtracts horizontally the packed DP FP numbers of both operands.

Subtracts the double-precision floating-point value in the high quadword of the destination operand from the low quadword of the destination operand and stores the result in the low quadword of the destination operand.

Subtracts the double-precision floating-point value in the high quadword of the source operand from the low quadword of the source operand and stores the result in the high quadword of the destination operand.

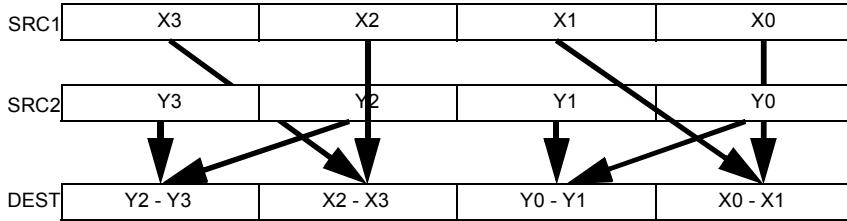
In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-19 for HSUBPD; see Figure 3-20 for VHSUBPD.



OM15995

Figure 3-19. HSUBPD—Packed Double-FP Horizontal Subtract

**Figure 3-20. VHSUBPD operation**

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

HSUBPD (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[VLMAX-1:128] (Unmodified)
```

VHSUBPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```

VHSUBPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[191:128] ← SRC1[191:128] - SRC1[255:192]
DEST[255:192] ← SRC2[191:128] - SRC2[255:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
HSUBPD: __m128d _mm_hsub_pd(__m128d a, __m128d b)
VHSUBPD: __m256d _mm256_hsub_pd (__m256d a, __m256d b);
```

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

HSUBPS—Packed Single-FP Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 7D /r HSUBPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE3	Horizontal subtract packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.F2.0F.WIG 7D /r VHSUBPS <i>xmm1, xmm2, xmm3/m128</i>		RVM V/V	AVX	Horizontal subtract packed single-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.F2.0F.WIG 7D /r VHSUBPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Horizontal subtract packed single-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtracts the single-precision floating-point value in the second dword of the destination operand from the first dword of the destination operand and stores the result in the first dword of the destination operand.

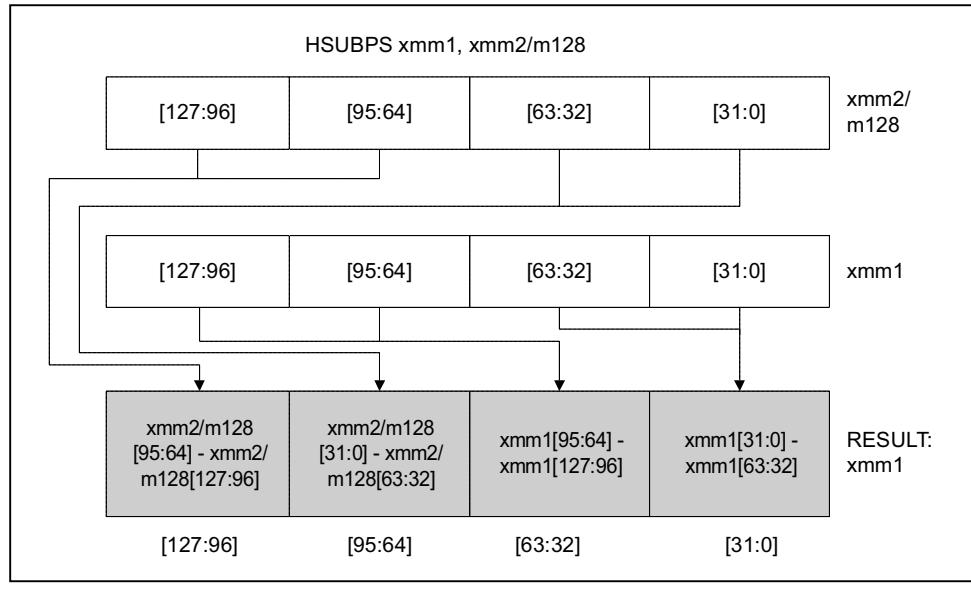
Subtracts the single-precision floating-point value in the fourth dword of the destination operand from the third dword of the destination operand and stores the result in the second dword of the destination operand.

Subtracts the single-precision floating-point value in the second dword of the source operand from the first dword of the source operand and stores the result in the third dword of the destination operand.

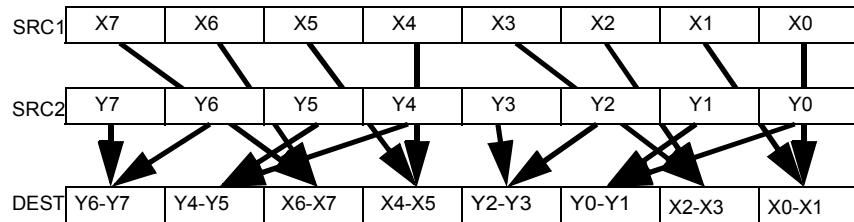
Subtracts the single-precision floating-point value in the fourth dword of the source operand from the third dword of the source operand and stores the result in the fourth dword of the destination operand.

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

See Figure 3-21 for HSUBPS; see Figure 3-22 for VHSUBPS.



OM15996

Figure 3-21. HSUBPS—Packed Single-FP Horizontal Subtract**Figure 3-22. VHSUBPS operation**

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

HSUBPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$
 $\text{DEST}[VLMAX-1:128] \text{ (Unmodified)}$

VHSUBPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

VHSUBPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] - \text{SRC1}[191:160]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[223:192] - \text{SRC1}[255:224]$
 $\text{DEST}[223:192] \leftarrow \text{SRC2}[159:128] - \text{SRC2}[191:160]$
 $\text{DEST}[255:224] \leftarrow \text{SRC2}[223:192] - \text{SRC2}[255:224]$

Intel C/C++ Compiler Intrinsic Equivalent

HSUBPS: `__m128 _mm_hsub_ps(__m128 a, __m128 b);`
 VHSUBPS: `__m256 _mm256_hsub_ps (__m256 a, __m256 b);`

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

IDIV—Signed Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /7	IDIV r/m8	M	Valid	Valid	Signed divide AX by r/m8, with result stored in: AL \leftarrow Quotient, AH \leftarrow Remainder.
REX + F6 /7	IDIV r/m8*	M	Valid	N.E.	Signed divide AX by r/m8, with result stored in: AL \leftarrow Quotient, AH \leftarrow Remainder.
F7 /7	IDIV r/m16	M	Valid	Valid	Signed divide DX:AX by r/m16, with result stored in AX \leftarrow Quotient, DX \leftarrow Remainder.
F7 /7	IDIV r/m32	M	Valid	Valid	Signed divide EDX:EAX by r/m32, with result stored in EAX \leftarrow Quotient, EDX \leftarrow Remainder.
REX.W + F7 /7	IDIV r/m64	M	Valid	N.E.	Signed divide RDX:RAX by r/m64, with result stored in RAX \leftarrow Quotient, RDX \leftarrow Remainder.

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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Divides the (signed) value in the AX, DX:AX, or EDX:EAX (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor).

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the signed value in RDX:RAX by the source operand. RAX contains a 64-bit quotient; RDX contains a 64-bit remainder.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-60.

Table 3-60. IDIV Results

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	r/m8	AL	AH	-128 to +127
Doubleword/word	DX:AX	r/m16	AX	DX	-32,768 to +32,767
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	-2^{31} to $2^{32} - 1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	-2^{63} to $2^{64} - 1$

Operation

```

IF SRC = 0
    THEN #DE; (* Divide error *)
FI;

IF OperandSize = 8 (* Word/byte operation *)
THEN
    temp ← AX / SRC; (* Signed division *)
    IF (temp > 7FH) or (temp < 80H)
        (* If a positive result is greater than 7FH or a negative result is less than 80H *)
        THEN #DE; (* Divide error *)
    ELSE
        AL ← temp;
        AH ← AX SignedModulus SRC;
    FI;
ELSE IF OperandSize = 16 (* Doubleword/word operation *)
THEN
    temp ← DX:AX / SRC; (* Signed division *)
    IF (temp > 7FFFFH) or (temp < 80000H)
        (* If a positive result is greater than 7FFFFH
        or a negative result is less than 80000H *)
        THEN
            #DE; (* Divide error *)
        ELSE
            AX ← temp;
            DX ← DX:AX SignedModulus SRC;
        FI;
    FI;
ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
temp ← EDX:EAX / SRC; (* Signed division *)
IF (temp > 7FFFFFFFH) or (temp < 800000000H)
    (* If a positive result is greater than 7FFFFFFFH
    or a negative result is less than 800000000H *)
    THEN
        #DE; (* Divide error *)
    ELSE
        EAX ← temp;
        EDX ← EDX:EAX SignedModulus SRC;
    FI;
FI;
ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
temp ← RDX:RAX / SRC; (* Signed division *)
IF (temp > 7FFFFFFFFFFFFFFFH) or (temp < 8000000000000000H)
    (* If a positive result is greater than 7FFFFFFFFFFFFFFFH
    or a negative result is less than 8000000000000000H *)
    THEN
        #DE; (* Divide error *)
    ELSE
        RAX ← temp;
        RDX ← RDX:RAX SignedModulus SRC;
    FI;
FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#DE	If the source operand (divisor) is 0. The signed result (quotient) is too large for the destination.
#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.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#DE	If the source operand (divisor) is 0. The signed result (quotient) is too large for the destination.
#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.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#DE	If the source operand (divisor) is 0. The signed result (quotient) is too large for the destination.
#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.
#UD	If the LOCK prefix is used.

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.
#DE	If the source operand (divisor) is 0 If the quotient is too large for the designated register.
#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.
#UD	If the LOCK prefix is used.

IMUL—Signed Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /5	IMUL r/m8*	M	Valid	Valid	AX← AL * r/m byte.
F7 /5	IMUL r/m16	M	Valid	Valid	DX:AX ← AX * r/m word.
F7 /5	IMUL r/m32	M	Valid	Valid	EDX:EAX ← EAX * r/m32.
REX.W + F7 /5	IMUL r/m64	M	Valid	N.E.	RDX:RAX ← RAX * r/m64.
OF AF /r	IMUL r16, r/m16	RM	Valid	Valid	word register ← word register * r/m16.
OF AF /r	IMUL r32, r/m32	RM	Valid	Valid	doubleword register ← doubleword register * r/m32.
REX.W + OF AF /r	IMUL r64, r/m64	RM	Valid	N.E.	Quadword register ← Quadword register * r/m64.
6B /r ib	IMUL r16, r/m16, imm8	RMI	Valid	Valid	word register ← r/m16 * sign-extended immediate byte.
6B /r ib	IMUL r32, r/m32, imm8	RMI	Valid	Valid	doubleword register ← r/m32 * sign-extended immediate byte.
REX.W + 6B /r ib	IMUL r64, r/m64, imm8	RMI	Valid	N.E.	Quadword register ← r/m64 * sign-extended immediate byte.
69 /r iw	IMUL r16, r/m16, imm16	RMI	Valid	Valid	word register ← r/m16 * immediate word.
69 /r id	IMUL r32, r/m32, imm32	RMI	Valid	Valid	doubleword register ← r/m32 * immediate doubleword.
REX.W + 69 /r id	IMUL r64, r/m64, imm32	RMI	Valid	N.E.	Quadword register ← r/m64 * immediate doubleword.

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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8/16/32	NA

Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form** — This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and the product is stored in the AX, DX:AX, EDX:EAX, or RDX:RAX registers, respectively.
- **Two-operand form** — With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The product is then stored in the destination operand location.
- **Three-operand form** — This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The product is then stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when significant bit (including the sign bit) are carried into the upper half of the result. The CF and OF flags are cleared when the result (including the sign bit) fits exactly in the lower half of the result.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. Use of REX.W modifies the three forms of the instruction as follows.

- One-operand form —The source operand (in a 64-bit general-purpose register or memory location) is multiplied by the value in the RAX register and the product is stored in the RDX:RAX registers.
- **Two-operand form** — The source operand is promoted to 64 bits if it is a register or a memory location. The destination operand is promoted to 64 bits.
- **Three-operand form** — The first source operand (either a register or a memory location) and destination operand are promoted to 64 bits. If the source operand is an immediate, it is sign extended to 64 bits.

Operation

```

IF (NumberOfOperands = 1)
  THEN IF (OperandSize = 8)
    THEN
      AX ← AL * SRC (* Signed multiplication *)
      IF AL = AX
        THEN CF ← 0; OF ← 0;
        ELSE CF ← 1; OF ← 1; Fl;
    ELSE IF OperandSize = 16
      THEN
        DX:AX ← AX * SRC (* Signed multiplication *)
        IF sign_extend_to_32 (AX) = DX:AX
          THEN CF ← 0; OF ← 0;
          ELSE CF ← 1; OF ← 1; Fl;
    ELSE IF OperandSize = 32
      THEN
        EDX:EAX ← EAX * SRC (* Signed multiplication *)
        IF EAX = EDX:EAX
          THEN CF ← 0; OF ← 0;
          ELSE CF ← 1; OF ← 1; Fl;
    ELSE (* OperandSize = 64 *)
      RDX:RAX ← RAX * SRC (* Signed multiplication *)
      IF RAX = RDX:RAX
        THEN CF ← 0; OF ← 0;
        ELSE CF ← 1; OF ← 1; Fl;
      Fl;
  Fl;
ELSE IF (NumberOfOperands = 2)
  THEN
    temp ← DEST * SRC (* Signed multiplication; temp is double DEST size *)
  
```

```

DEST ← DEST * SRC (* Signed multiplication *)
IF temp ≠ DEST
    THEN CF ← 1; OF ← 1;
    ELSE CF ← 0; OF ← 0; Fl;
ELSE (* NumberOfOperands = 3 *)
    DEST ← SRC1 * SRC2 (* Signed multiplication *)
    temp ← SRC1 * SRC2 (* Signed multiplication; temp is double SRC1 size *)
    IF temp ≠ DEST
        THEN CF ← 1; OF ← 1;
        ELSE CF ← 0; OF ← 0; Fl;
    Fl;
Fl;

```

Flags Affected

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are 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 is used to access memory and it contains a NULL 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

IN—Input from Port

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E4 <i>ib</i>	IN AL, <i>imm8</i>	I	Valid	Valid	Input byte from <i>imm8</i> I/O port address into AL.
E5 <i>ib</i>	IN AX, <i>imm8</i>	I	Valid	Valid	Input word from <i>imm8</i> I/O port address into AX.
E5 <i>ib</i>	IN EAX, <i>imm8</i>	I	Valid	Valid	Input dword from <i>imm8</i> I/O port address into EAX.
EC	IN AL,DX	NP	Valid	Valid	Input byte from I/O port in DX into AL.
ED	IN AX,DX	NP	Valid	Valid	Input word from I/O port in DX into AX.
ED	IN EAX,DX	NP	Valid	Valid	Input doubleword from I/O port in DX into EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	imm8	NA	NA	NA
NP	NA	NA	NA	NA

Description

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

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size. 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 16, "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.

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; (* Read from selected I/O port *)
    FI;
  ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
    DEST ← SRC; (* Read from 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.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #UD If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit 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.
- #UD If the LOCK prefix is used.

INC—Increment by 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FE /0	INC r/m8	M	Valid	Valid	Increment r/m byte by 1.
REX + FE /0	INC r/m8*	M	Valid	N.E.	Increment r/m byte by 1.
FF /0	INC r/m16	M	Valid	Valid	Increment r/m word by 1.
FF /0	INC r/m32	M	Valid	Valid	Increment r/m doubleword by 1.
REX.W + FF /0	INC r/m64	M	Valid	N.E.	Increment r/m quadword by 1.
40+ rw**	INC r16	O	N.E.	Valid	Increment word register by 1.
40+ rd	INC r32	O	N.E.	Valid	Increment doubleword register by 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.

** 40H through 47H are REX prefixes in 64-bit mode.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
O	opcode + rd (r, w)	NA	NA	NA

Description

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

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

In 64-bit mode, INC r16 and INC r32 are not encodable (because opcodes 40H through 47H are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

Operation

DEST ← DEST + 1;

AFlags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

- #GP(0) If the destination 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 is used to access memory and it contains a NULLsegment 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.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

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.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

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.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

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.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

INS/INSB/INSW/INSD—Input from Port to String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
6C	INS <i>m8</i> , DX	NP	Valid	Valid	Input byte from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.*
6D	INS <i>m16</i> , DX	NP	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹
6D	INS <i>m32</i> , DX	NP	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹
6C	INSB	NP	Valid	Valid	Input byte from I/O port specified in DX into memory location specified with ES:(E)DI or RDI. ¹
6D	INSW	NP	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹
6D	INSD	NP	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. ¹

NOTES:

- * 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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Copies the data from the I/O port specified with the source operand (second operand) to the destination operand (first operand). The source operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The destination operand is a memory location, the address of which is read from either the ES:DI, ES:EDI or the RDI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The ES segment cannot be overridden with a segment override prefix.) 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 INS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand must be “DX,” and the destination operand should be a symbol that indicates the size of the I/O port and the destination address. This 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 registers, which must be loaded correctly before the INS instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the INS instructions. Here also DX is assumed by the processor to be the source operand and ES:(E)DI is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: INSB (byte), INSW (word), or INSD (doubleword).

After the byte, word, or doubleword is transfer from the I/O port to the memory location, the DI/EDI/RDI 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 (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The INS, INSB, INSW, and INSD 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 Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a description of the REP prefix.

These instructions are only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 16, "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, default address size is 64 bits, 32 bit address size is supported using the prefix 67H. The address of the memory destination is specified by RDI or EDI. 16-bit address size is not supported in 64-bit mode. The operand size is not promoted.

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; (* Read from I/O port *)
    FI;
  ELSE (Real Mode or Protected Mode with CPL IOPL *)
    DEST ← SRC; (* Read from I/O port *)
FI;
```

Non-64-bit Mode:

```

IF (Byte transfer)
  THEN IF DF = 0
    THEN (E)DI ← (E)DI + 1;
    ELSE (E)DI ← (E)DI - 1; FI;
  ELSE IF (Word transfer)
    THEN IF DF = 0
      THEN (E)DI ← (E)DI + 2;
      ELSE (E)DI ← (E)DI - 2; FI;
    ELSE (* Doubleword transfer *)
      THEN IF DF = 0
        THEN (E)DI ← (E)DI + 4;
        ELSE (E)DI ← (E)DI - 4; FI;
    FI;
FI;
```

FI64-bit Mode:

```

IF (Byte transfer)
  THEN IF DF = 0
    THEN (E|R)DI ← (E|R)DI + 1;
    ELSE (E|R)DI ← (E|R)DI - 1; FI;
  ELSE IF (Word transfer)
    THEN IF DF = 0
      THEN (E)DI ← (E)DI + 2;
      ELSE (E)DI ← (E)DI - 2; FI;
    ELSE (* Doubleword transfer *)
      THEN IF DF = 0
        THEN (E|R)DI ← (E|R)DI + 4;
        ELSE (E|R)DI ← (E|R)DI - 4; FI;
```

Fl;
Fl;

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 the destination is located in a non-writable segment. If an illegal memory operand effective address in the ES segments 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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 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.
#UD	If the LOCK prefix is used.

INSERTPS – Insert Packed Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 21 /r ib INSERTPS <i>xmm1, xmm2/m32, imm8</i>	RMI	V/V	SSE4_1	Insert a single precision floating-point value selected by <i>imm8</i> from <i>xmm2/m32</i> into <i>xmm1</i> at the specified destination element specified by <i>imm8</i> and zero out destination elements in <i>xmm1</i> as indicated in <i>imm8</i> .
VEX.NDS.128.66.0F3A.WIG 21 /r ib VININSERTPS <i>xmm1, xmm2, xmm3/m32, imm8</i>	RVMI	V/V	AVX	Insert a single precision floating point value selected by <i>imm8</i> from <i>xmm3/m32</i> and merge into <i>xmm2</i> at the specified destination element specified by <i>imm8</i> and zero out destination elements in <i>xmm1</i> as indicated in <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(register source form)

Select a single precision floating-point element from second source as indicated by Count_S bits of the immediate operand and insert it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

(memory source form)

Load a floating-point element from a 32-bit memory location and insert it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version. The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

If VININSERTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

INSERTPS (128-bit Legacy SSE version)

```

IF (SRC = REG) THEN COUNT_S ← imm8[7:6]
  ELSE COUNT_S ← 0
COUNT_D ← imm8[5:4]
ZMASK ← imm8[3:0]
CASE (COUNT_S) OF
  0: TMP ← SRC[31:0]
  1: TMP ← SRC[63:32]
  2: TMP ← SRC[95:64]
  3: TMP ← SRC[127:96]
ESAC;

CASE (COUNT_D) OF
  0: TMP2[31:0] ← TMP
    TMP2[127:32] ← DEST[127:32]
  1: TMP2[63:32] ← TMP
    TMP2[31:0] ← DEST[31:0]
    TMP2[127:64] ← DEST[127:64]
  2: TMP2[95:64] ← TMP
    TMP2[63:0] ← DEST[63:0]
    TMP2[127:96] ← DEST[127:96]
  3: TMP2[127:96] ← TMP
    TMP2[95:0] ← DEST[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H
  ELSE DEST[31:0] ← TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] ← 00000000H
  ELSE DEST[63:32] ← TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] ← 00000000H
  ELSE DEST[95:64] ← TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] ← 00000000H
  ELSE DEST[127:96] ← TMP2[127:96]
DEST[VLMAX-1:128] (Unmodified)

```

VINSERTPS (VEX.128 encoded version)

```

IF (SRC = REG) THEN COUNT_S ← imm8[7:6]
  ELSE COUNT_S ← 0
COUNT_D ← imm8[5:4]
ZMASK ← imm8[3:0]
CASE (COUNT_S) OF
  0: TMP ← SRC2[31:0]
  1: TMP ← SRC2[63:32]
  2: TMP ← SRC2[95:64]
  3: TMP ← SRC2[127:96]
ESAC;

CASE (COUNT_D) OF
  0: TMP2[31:0] ← TMP
    TMP2[127:32] ← SRC1[127:32]
  1: TMP2[63:32] ← TMP
    TMP2[31:0] ← SRC1[31:0]
    TMP2[127:64] ← SRC1[127:64]

```

```

2: TMP2[95:64] ← TMP
   TMP2[63:0] ← SRC1[63:0]
   TMP2[127:96] ← SRC1[127:96]
3: TMP2[127:96] ← TMP
   TMP2[95:0] ← SRC1[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H
ELSE DEST[31:0] ← TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] ← 00000000H
ELSE DEST[63:32] ← TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] ← 00000000H
ELSE DEST[95:64] ← TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] ← 00000000H
ELSE DEST[127:96] ← TMP2[127:96]
DEST[VLMAX-1:128] ← 0

```

Intel C/C++ Compiler Intrinsic Equivalent

INSERTPS: `__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx);`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5.

INT *n*/INTO/INT 3—Call to Interrupt Procedure

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CC	INT 3	NP	Valid	Valid	Interrupt 3—trap to debugger.
CD <i>ib</i>	INT <i>imm8</i>	I	Valid	Valid	Interrupt vector number specified by immediate byte.
CE	INTO	NP	Invalid	Valid	Interrupt 4—if overflow flag is 1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA
I	imm8	NA	NA	NA

Description

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled “Interrupts and Exceptions” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). The destination operand specifies an interrupt vector number from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each interrupt vector number provides an index to a gate descriptor in the IDT. The first 32 interrupt vector numbers are reserved by Intel for system use. Some of these interrupts are used for internally generated exceptions.

The INT *n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), interrupt vector number 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code). To further support its function as a debug breakpoint, the interrupt generated with the CC opcode also differs from the regular software interrupts as follows:

- Interrupt redirection does not happen when in VME mode; the interrupt is handled by a protected-mode handler.
- The virtual-8086 mode IOPL checks do not occur. The interrupt is taken without faulting at any IOPL level.

Note that the “normal” 2-byte opcode for INT 3 (CD03) does not have these special features. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.

The action of the INT *n* instruction (including the INTO and INT 3 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT *n* instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

The interrupt vector number specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the **interrupt vector table**, and its pointers are called interrupt vectors.)

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in

the “Operation” section for this instruction (except #GP).

Table 3-61. Decision Table

PE	0	1	1	1	1	1	1	1
VM	-	-	-	-	-	0	1	1
IOPL	-	-	-	-	-	-	<3	=3
DPL/CPL RELATIONSHIP	-	DPL < CPL	-	DPL > CPL	DPL = CPL or C	DPL < CPL & NC	-	-
INTERRUPT TYPE	-	S/W	-	-	-	-	-	-
GATE TYPE	-	-	Task	Trap or Interrupt				
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

NOTES:

- Don’t Care.
- Y Yes, action taken.
- Blank Action not taken.

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT n instruction. If the IOPL is less than 3, the processor generates a #GP(selector) exception; if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate’s DPL must be set to 3 and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

Operation

The following operational description applies not only to the INT n and INTO instructions, but also to external interrupts, nonmaskable interrupts (NMIs), and exceptions. Some of these events push onto the stack an error code.

The operational description specifies numerous checks whose failure may result in delivery of a nested exception. In these cases, the original event is not delivered.

The operational description specifies the error code delivered by any nested exception. In some cases, the error code is specified with a pseudofunction `error_code(num,idt,ext)`, where idt and ext are bit values. The pseudofunction produces an error code as follows: (1) if idt is 0, the error code is `(num & FCH) | ext`; (2) if idt is 1, the error code is `(num << 3) | 2 | ext`.

In many cases, the pseudofunction `error_code` is invoked with a pseudovariable EXT. The value of EXT depends on the nature of the event whose delivery encountered a nested exception: if that event is a software interrupt, EXT is 0; otherwise, EXT is 1.

INSTRUCTION SET REFERENCE, A-M

```

IF PE = 0
  THEN
    GOTO REAL-ADDRESS-MODE;
  ELSE (* PE = 1 *)
    IF (VM = 1 and IOPL < 3 AND INT n)
      THEN
        #GP(0); (* Bit 0 of error code is 0 because INT n *)
      ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
        IF (IA32_EFER.LMA = 0)
          THEN (* Protected mode, or virtual-8086 mode interrupt *)
            GOTO PROTECTED-MODE;
        ELSE (* IA-32e mode interrupt *)
          GOTO IA-32e-MODE;
        FI;
      FI;
    FI;
REAL-ADDRESS-MODE:
  IF ((vector_number << 2) + 3) is not within IDT limit
    THEN #GP; FI;
  IF stack not large enough for a 6-byte return information
    THEN #SS; FI;
  Push (EFLAGS[15:0]);
  IF ← 0; (* Clear interrupt flag *)
  TF ← 0; (* Clear trap flag *)
  AC ← 0; (* Clear AC flag *)
  Push(CS);
  Push(IP);
  (* No error codes are pushed in real-address mode*)
  CS ← IDT(Descriptor (vector_number << 2), selector));
  EIP ← IDT(Descriptor (vector_number << 2), offset)); (* 16 bit offset AND 0000FFFFH *)
END;
PROTECTED-MODE:
  IF ((vector_number << 3) + 7) is not within IDT limits
  or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
    THEN #GP(error_code(vector_number,1,EXT)); FI;
    (* idt operand to error_code set because vector is used *)
  IF software interrupt (* Generated by INT n, INT3, or INTO *)
    THEN
      IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
        THEN #GP(error_code(vector_number,1,0)); FI;
        (* idt operand to error_code set because vector is used *)
        (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
    FI;
  IF gate not present
    THEN #NP(error_code(vector_number,1,EXT)); FI;
    (* idt operand to error_code set because vector is used *)
  IF task gate (* Specified in the selected interrupt table descriptor *)
    THEN GOTO TASK-GATE;
    ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
  FI;
END;
IA-32e-MODE:
  IF INTO and CS.L = 1 (64-bit mode)
    THEN #UD;

```

```

Fl;
IF ((vector_number << 4) + 15) is not in IDT limits
or selected IDT descriptor is not an interrupt-, or trap-gate type
    THEN #GP(error_code(vector_number,1,EXT));
        (* idt operand to error_code set because vector is used *)
Fl;
IF software interrupt (* Generated by INT n, INT 3, or INTO *)
    THEN
        IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
            THEN #GP(error_code(vector_number,1,0));
                (* idt operand to error_code set because vector is used *)
                (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
Fl;
Fl;
IF gate not present
    THEN #NP(error_code(vector_number,1,EXT));
        (* idt operand to error_code set because vector is used *)
Fl;
GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
TASK-GATE: (* PE = 1, task gate *)
    Read TSS selector in task gate (IDT descriptor);
    IF local/global bit is set to local or index not within GDT limits
        THEN #GP(error_code(TSS selector,0,EXT)); Fl;
            (* idt operand to error_code is 0 because selector is used *)
    Access TSS descriptor in GDT;
    IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
        THEN #GP(TSS selector,0,EXT)); Fl;
            (* idt operand to error_code is 0 because selector is used *)
    IF TSS not present
        THEN #NP(TSS selector,0,EXT)); Fl;
            (* idt operand to error_code is 0 because selector is used *)
SWITCH-TASKS (with nesting) to TSS;
IF interrupt caused by fault with error code
    THEN
        IF stack limit does not allow push of error code
            THEN #SS(EXT); Fl;
            Push(error code);
Fl;
IF EIP not within code segment limit
    THEN #GP(EXT); Fl;
END;
TRAP-OR-INTERRUPT-GATE:
    Read new code-segment selector for trap or interrupt gate (IDT descriptor);
    IF new code-segment selector is NULL
        THEN #GP(EXT); Fl; (* Error code contains NULL selector *)
    IF new code-segment selector is not within its descriptor table limits
        THEN #GP(error_code(new code-segment selector,0,EXT)); Fl;
            (* idt operand to error_code is 0 because selector is used *)
    Read descriptor referenced by new code-segment selector;
    IF descriptor does not indicate a code segment or new code-segment DPL > CPL
        THEN #GP(error_code(new code-segment selector,0,EXT)); Fl;
            (* idt operand to error_code is 0 because selector is used *)
    IF new code-segment descriptor is not present,

```

```

THEN #NP(error_code(new code-segment selector,0,EXT)); Fl;
(* idt operand to error_code is 0 because selector is used *)
IF new code segment is non-conforming with DPL < CPL
THEN
  IF VM = 0
  THEN
    GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
    (* PE = 1, VM = 0, interrupt or trap gate, nonconforming code segment,
       DPL < CPL *)
  ELSE (* VM = 1 *)
    IF new code-segment DPL ≠ 0
      THEN #GP(error_code(new code-segment selector,0,EXT));
      (* idt operand to error_code is 0 because selector is used *)
      GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE; Fl;
      (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
    Fl;
  ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
    IF VM = 1
      THEN #GP(error_code(new code-segment selector,0,EXT));
      (* idt operand to error_code is 0 because selector is used *)
    IF new code segment is conforming or new code-segment DPL = CPL
    THEN
      GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
    ELSE (* PE = 1, interrupt or trap gate, nonconforming code segment, DPL > CPL *)
      #GP(error_code(new code-segment selector,0,EXT));
      (* idt operand to error_code is 0 because selector is used *)
    Fl;
  Fl;
END;
INTER-PRIVILEGE-LEVEL-INTERRUPT:
(* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
THEN
  (* Identify stack-segment selector for new privilege level in current TSS *)
  IF current TSS is 32-bit
  THEN
    TSSstackAddress ← (new code-segment DPL ‡ 3) + 4;
    IF (TSSstackAddress + 5) > current TSS limit
      THEN #TS(error_code(current TSS selector,0,EXT)); Fl;
      (* idt operand to error_code is 0 because selector is used *)
    NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
    NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
  ELSE (* current TSS is 16-bit *)
    TSSstackAddress ← (new code-segment DPL ‡ 2) + 2
    IF (TSSstackAddress + 3) > current TSS limit
      THEN #TS(error_code(current TSS selector,0,EXT)); Fl;
      (* idt operand to error_code is 0 because selector is used *)
    NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
    NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
  Fl;
  IF NewSS is NULL
    THEN #TS(EXT); Fl;
  IF NewSS index is not within its descriptor-table limits
  or NewSS RPL ≠ new code-segment DPL

```

```

        THEN #TS(error_code(NewSS,0,EXT)); Fl;
        (* idt operand to error_code is 0 because selector is used *)
    Read new stack-segment descriptor for NewSS in GDT or LDT;
    IF new stack-segment DPL ≠ new code-segment DPL
    or new stack-segment Type does not indicate writable data segment
        THEN #TS(error_code(NewSS,0,EXT)); Fl;
        (* idt operand to error_code is 0 because selector is used *)
    IF NewSS is not present
        THEN #SS(error_code(NewSS,0,EXT)); Fl;
        (* idt operand to error_code is 0 because selector is used *)
    ELSE (* IA-32e mode *)
        IF IDT-gate IST = 0
            THEN TSSstackAddress ← (new code-segment DPL << 3) + 4;
            ELSE TSSstackAddress ← (IDT gate IST << 3) + 28;
        Fl;
        IF (TSSstackAddress + 7) > current TSS limit
            THEN #TS(error_code(current TSS selector,0,EXT)); Fl;
            (* idt operand to error_code is 0 because selector is used *)
        NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
        NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
    Fl;
    IF IDT gate is 32-bit
        THEN
            IF new stack does not have room for 24 bytes (error code pushed)
            or 20 bytes (no error code pushed)
                THEN #SS(error_code(NewSS,0,EXT)); Fl;
                (* idt operand to error_code is 0 because selector is used *)
        Fl
    ELSE
        IF IDT gate is 16-bit
            THEN
                IF new stack does not have room for 12 bytes (error code pushed)
                or 10 bytes (no error code pushed);
                    THEN #SS(error_code(NewSS,0,EXT)); Fl;
                    (* idt operand to error_code is 0 because selector is used *)
            ELSE (* 64-bit IDT gate*)
                IF StackAddress is non-canonical
                    THEN #SS(EXT); Fl; (* Error code contains NULL selector *)
            Fl;
    Fl;
    IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
            IF instruction pointer from IDT gate is not within new code-segment limits
                THEN #GP(EXT); Fl; (* Error code contains NULL selector *)
            ESP ← NewESP;
            SS ← NewSS; (* Segment descriptor information also loaded *)
        ELSE (* IA-32e mode *)
            IF instruction pointer from IDT gate contains a non-canonical address
                THEN #GP(EXT); Fl; (* Error code contains NULL selector *)
            RSP ← NewRSP & FFFFFFFFFFFFFFOH;
            SS ← NewSS;
        Fl;
    IF IDT gate is 32-bit
        THEN

```

INSTRUCTION SET REFERENCE, A-M

```
    CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
ELSE
    IF IDT gate 16-bit
        THEN
            CS:IP ← Gate(CS:IP);
            (* Segment descriptor information also loaded *)
        ELSE (* 64-bit IDT gate *)
            CS:RIP ← Gate(CS:RIP);
            (* Segment descriptor information also loaded *)
        FI;
    FI;
    IF IDT gate is 32-bit
        THEN
            Push(far pointer to old stack);
            (* Old SS and ESP, 3 words padded to 4 *)
            Push(EFLAGS);
            Push(far pointer to return instruction);
            (* Old CS and EIP, 3 words padded to 4 *)
            Push(ErrorCode); (* If needed, 4 bytes *)
        ELSE
            IF IDT gate 16-bit
                THEN
                    Push(far pointer to old stack);
                    (* Old SS and SP, 2 words *)
                    Push(EFLAGS(15-0]);
                    Push(far pointer to return instruction);
                    (* Old CS and IP, 2 words *)
                    Push(ErrorCode); (* If needed, 2 bytes *)
                ELSE (* 64-bit IDT gate *)
                    Push(far pointer to old stack);
                    (* Old SS and SP, each an 8-byte push *)
                    Push(RFLAGS); (* 8-byte push *)
                    Push(far pointer to return instruction);
                    (* Old CS and RIP, each an 8-byte push *)
                    Push(ErrorCode); (* If needed, 8-bytes *)
                FI;
            FI;
            CPL ← new code-segment DPL;
            CS(RPL) ← CPL;
            IF IDT gate is interrupt gate
                THEN IF ← 0 (* Interrupt flag set to 0, interrupts disabled *); FI;
                TF ← 0;
                VM ← 0;
                RF ← 0;
                NT ← 0;
            END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
(* Identify stack-segment selector for privilege level 0 in current TSS *)
IF current TSS is 32-bit
    THEN
        IF TSS limit < 9
            THEN #TS(error_code(current TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        NewSS ← 2 bytes loaded from (current TSS base + 8);
```

```

NewESP ← 4 bytes loaded from (current TSS base + 4);
ELSE (* current TSS is 16-bit *)
    IF TSS limit < 5
        THEN #TS(error_code(current TSS selector,0,EXT)); Fl;
        (* idt operand to error_code is 0 because selector is used *)
    NewSS ← 2 bytes loaded from (current TSS base + 4);
    NewESP ← 2 bytes loaded from (current TSS base + 2);
Fl;
IF NewSS is NULL
    THEN #TS(EXT); Fl; (* Error code contains NULL selector *)
IF NewSS index is not within its descriptor table limits
or NewSS RPL ≠ 0
    THEN #TS(error_code(NewSS,0,EXT)); Fl;
    (* idt operand to error_code is 0 because selector is used *)
Read new stack-segment descriptor for NewSS in GDT or LDT;
IF new stack-segment DPL ≠ 0 or stack segment does not indicate writable data segment
    THEN #TS(error_code(NewSS,0,EXT)); Fl;
    (* idt operand to error_code is 0 because selector is used *)
IF new stack segment not present
    THEN #SS(error_code(NewSS,0,EXT)); Fl;
    (* idt operand to error_code is 0 because selector is used *)
IF IDT gate is 32-bit
    THEN
        IF new stack does not have room for 40 bytes (error code pushed)
        or 36 bytes (no error code pushed)
            THEN #SS(error_code(NewSS,0,EXT)); Fl;
            (* idt operand to error_code is 0 because selector is used *)
        ELSE (* IDT gate is 16-bit)
            IF new stack does not have room for 20 bytes (error code pushed)
            or 18 bytes (no error code pushed)
                THEN #SS(error_code(NewSS,0,EXT)); Fl;
                (* idt operand to error_code is 0 because selector is used *)
Fl;
IF instruction pointer from IDT gate is not within new code-segment limits
    THEN #GP(EXT); Fl; (* Error code contains NULL selector *)
tempEFLAGS ← EFLAGS;
VM ← 0;
TF ← 0;
RF ← 0;
NT ← 0;
IF service through interrupt gate
    THEN IF = 0; Fl;
TempSS ← SS;
TempESP ← ESP;
SS ← NewSS;
ESP ← NewESP;
(* Following pushes are 16 bits for 16-bit IDT gates and 32 bits for 32-bit IDT gates;
Segment selector pushes in 32-bit mode are padded to two words *)
Push(GS);
Push(FS);
Push(DS);
Push(ES);
Push(TempSS);
Push(TempESP);

```

INSTRUCTION SET REFERENCE, A-M

```

Push(TempEFlags);
Push(CS);
Push(EIP);
GS ← 0; (* Segment registers made NULL, invalid for use in protected mode *)
FS ← 0;
DS ← 0;
ES ← 0;
CS:IP ← Gate(CS); (* Segment descriptor information also loaded *)
IF OperandSize = 32
    THEN
        EIP ← Gate(instruction pointer);
    ELSE (* OperandSize is 16 *)
        EIP ← Gate(instruction pointer) AND 0000FFFFH;
FI;
(* Start execution of new routine in Protected Mode *)
END;
INTRAPRIVILEGE-LEVEL-INTERRUPT:
(* PE = 1, DPL = CPL or conforming segment *)
IF IA32_EFER.LMA = 1 (* IA-32e mode *)
    IF IDT-descriptor IST ≠ 0
        THEN
            TSSstackAddress ← (IDT-descriptor IST << 3) + 28;
            IF (TSSstackAddress + 7) > TSS limit
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
        FI;
        IF 32-bit gate (* implies IA32_EFER.LMA = 0 *)
            THEN
                IF current stack does not have room for 16 bytes (error code pushed)
                or 12 bytes (no error code pushed)
                    THEN #SS(EXT); FI; (* Error code contains NULL selector *)
                ELSE IF 16-bit gate (* implies IA32_EFER.LMA = 0 *)
                    IF current stack does not have room for 8 bytes (error code pushed)
                    or 6 bytes (no error code pushed)
                        THEN #SS(EXT); FI; (* Error code contains NULL selector *)
                    ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                        IF NewRSP contains a non-canonical address
                            THEN #SS(EXT); (* Error code contains NULL selector *)
                        FI;
                FI;
                IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
                    THEN
                        IF instruction pointer from IDT gate is not within new code-segment limit
                            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
                        ELSE
                            IF instruction pointer from IDT gate contains a non-canonical address
                                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
                            RSP ← NewRSP & FFFFFFFFFFFFFFOH;
                    FI;
                    IF IDT gate is 32-bit (* implies IA32_EFER.LMA = 0 *)
                        THEN
                            Push (EFLAGS);
                            Push (far pointer to return instruction); (* 3 words padded to 4 *)

```

```

CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
Push (ErrorCode); (* If any *)
ELSE
  IF IDT gate is 16-bit (* implies IA32_EFER.LMA = 0 *)
    THEN
      Push (FLAGS);
      Push (far pointer to return location); (* 2 words *)
      CS:IP ← Gate(CS:IP);
      (* Segment descriptor information also loaded *)
      Push (ErrorCode); (* If any *)
    ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
      Push(far pointer to old stack);
      (* Old SS and SP, each an 8-byte push *)
      Push(RFLAGS); (* 8-byte push *)
      Push(far pointer to return instruction);
      (* Old CS and RIP, each an 8-byte push *)
      Push(ErrorCode); (* If needed, 8 bytes *)
      CS:RIP ← GATE(CS:RIP);
      (* Segment descriptor information also loaded *)
    FI;
  FI;
CS(RPL) ← CPL;
IF IDT gate is interrupt gate
  THEN IF ← 0; FI; (* Interrupt flag set to 0; interrupts disabled *)
TF ← 0;
NT ← 0;
VM ← 0;
RF ← 0;
END;

```

Flags Affected

The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see the “Operation” section). If the interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task’s TSS.

Protected Mode Exceptions

- #GP(error_code) If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
 If the segment selector in the interrupt-, trap-, or task gate is NULL.
 If an interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
 If the interrupt vector number is outside the IDT limits.
 If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.
 If an interrupt is generated by the INT *n*, INT 3, or INTO instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.
 If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
 If the segment selector for a TSS has its local/global bit set for local.
 If a TSS segment descriptor specifies that the TSS is busy or not available.
- #SS(error_code) If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.
 If the SS register is being loaded and the segment pointed to is marked not present.

	If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.
#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is NULL.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the interrupt vector number is outside the IDT limits.
#SS	If stack limit violation on push.
	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(error_code)	(For INT <i>n</i> , INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt-, trap-, or task-gate descriptor is not equal to 3. If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits. If the segment selector in the interrupt-, trap-, or task gate is NULL. If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits. If the interrupt vector number is outside the IDT limits. If an IDT descriptor is not an interrupt-, trap-, or task-descriptor. If an interrupt is generated by the INT <i>n</i> instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL. If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment. If the segment selector for a TSS has its local/global bit set for local.
#SS(error_code)	If the SS register is being loaded and the segment pointed to is marked not present. If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.
#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate. If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate. If the stack segment selector in the TSS is NULL. If the stack segment for the TSS is not a writable data segment. If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.

#BP	If the INT 3 instruction is executed.
#OF	If the INTO instruction is executed and the OF flag is set.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(error_code)	If the instruction pointer in the 64-bit interrupt gate or 64-bit trap gate is non-canonical. If the segment selector in the 64-bit interrupt or trap gate is NULL. If the interrupt vector number is outside the IDT limits. If the interrupt vector number points to a gate which is in non-canonical space. If the interrupt vector number points to a descriptor which is not a 64-bit interrupt gate or 64-bit trap gate. If the descriptor pointed to by the gate selector is outside the descriptor table limit. If the descriptor pointed to by the gate selector is in non-canonical space. If the descriptor pointed to by the gate selector is not a code segment. If the descriptor pointed to by the gate selector doesn't have the L-bit set, or has both the L-bit and D-bit set. If the descriptor pointed to by the gate selector has DPL > CPL.
#SS(error_code)	If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch. If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST).
#NP(error_code)	If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.
#TS(error_code)	If an attempt to load RSP from the TSS causes an access to non-canonical space. If the RSP from the TSS is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

INVD—Invalidate Internal Caches

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 08	INVD	NP	Valid	Valid	Flush internal caches; initiate flushing of external caches.

NOTES:

- * See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Invalidates (flushes) the processor's internal caches and issues a special-function bus cycle that directs external caches to also flush themselves. Data held in internal caches is not written back to main memory.

After executing this instruction, the processor does not wait for the external caches to complete their flushing operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache flush signal.

The INVD 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.

The INVD instruction may be used when the cache is used as temporary memory and the cache contents need to be invalidated rather than written back to memory. When the cache is used as temporary memory, no external device should be actively writing data to main memory.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Note that any data from an external device to main memory (for example, via a PCIWrite) can be temporarily stored in the caches; these data can be lost when an INVD instruction is executed. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, temporary memory, testing, or fault recovery where cache coherency with main memory is not a concern), software should instead use the WBINVD instruction.

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

IA-32 Architecture Compatibility

The INVD instruction is implementation dependent; it may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

```
Flush(InternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution *)
```

Flags Affected

None.

Protected Mode Exceptions

- | | |
|--------|--|
| #GP(0) | If the current privilege level is not 0. |
| #UD | If the LOCK prefix is used. |

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) The INVD instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

INVLPG—Invalidate TLB Entry

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 01/7	INVLPG <i>m</i>	M	Valid	Valid	Invalidate TLB Entry for page that contains <i>m</i> .

NOTES:

- * See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Invalidates (flushes) the translation lookaside buffer (TLB) entry specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes the TLB entry for that page.

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL must be 0 to execute this instruction.

The INVLPG instruction normally flushes the TLB entry only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction is guaranteed to invalidates only TLB entries associated with the current PCID. (If PCIDs are disabled — CR4.PCIDE = 0 — the current PCID is 000H.) The instruction also invalidates any global TLB entries for the specified page, regardless of PCID.

For more details on operations that flush the TLB, see “MOV—Move to/from Control Registers” and Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

This instruction’s operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

IA-32 Architecture Compatibility

The INVLPG instruction is implementation dependent, and its function may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

Flush(RelevantTLBEntries);
Continue; (* Continue execution *)

Flags Affected

None.

Protected Mode Exceptions

- | | |
|--------|--|
| #GP(0) | If the current privilege level is not 0. |
| #UD | Operand is a register. |
| | If the LOCK prefix is used. |

Real-Address Mode Exceptions

- | | |
|-----|-----------------------------|
| #UD | Operand is a register. |
| | If the LOCK prefix is used. |

Virtual-8086 Mode Exceptions

#GP(0) The INVLPG instruction cannot be executed at the virtual-8086 mode.

64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0.

#UD Operand is a register.

If the LOCK prefix is used.

INVPCID—Invalidate Process-Context Identifier

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Feature Flag	Description
66 0F 38 82 /r INVPCID r32, m128		RM	NE/V	INVPCID Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r32 and descriptor in m128.
66 0F 38 82 /r INVPCID r64, m128		RM	V/NE	INVPCID Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r64 and descriptor in m128.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (R)	ModRM:r/m (R)	NA	NA

Description

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches based on process-context identifier (PCID). (See Section 4.10, “Caching Translation Information,” in *Intel 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A*.) Invalidations are based on the INVPCID type specified in the register operand and the INVPCID descriptor specified in the memory operand.

Outside 64-bit mode, the register operand is always 32 bits, regardless of the value of CS.D. In 64-bit mode the register operand has 64 bits.

There are four INVPCID types currently defined:

- Individual-address invalidation: If the INVPCID type is 0, the logical processor invalidates mappings—except global translations—for the linear address and PCID specified in the INVPCID descriptor. In some cases, the instruction may invalidate global translations or mappings for other linear addresses (or other PCIDs) as well.
- Single-context invalidation: If the INVPCID type is 1, the logical processor invalidates all mappings—except global translations—associated with the PCID specified in the INVPCID descriptor. In some cases, the instruction may invalidate global translations or mappings for other PCIDs as well.
- All-context invalidation, including global translations: If the INVPCID type is 2, the logical processor invalidates all mappings—including global translations—associated with any PCID.
- All-context invalidation: If the INVPCID type is 3, the logical processor invalidates all mappings—except global translations—associated with any PCID. In some case, the instruction may invalidate global translations as well.

The INVPCID descriptor comprises 128 bits and consists of a PCID and a linear address as shown in Figure 3-23. For INVPCID type 0, the processor uses the full 64 bits of the linear address even outside 64-bit mode; the linear address is not used for other INVPCID types.

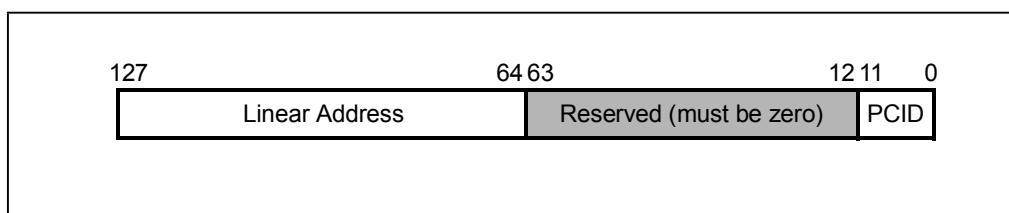


Figure 3-23. INVPCID Descriptor

If CR4.PCIDE = 0, a logical processor does not cache information for any PCID other than 000H. In this case, executions with INVPCID types 0 and 1 are allowed only if the PCID specified in the INVPCID descriptor is 000H;

executions with INVPCID types 2 and 3 invalidate mappings only for PCID 000H. Note that CR4.PCIDE must be 0 outside 64-bit mode (see Chapter 4.10.1, “Process-Context Identifiers (PCIDs),” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

Operation

```

INVPCID_TYPE ← value of register operand;      // must be in the range of 0-3
INVPCID_DESC ← value of memory operand;
CASE INVPCID_TYPE OF
  0:           // individual-address invalidation
    PCID ← INVPCID_DESC[11:0];
    L_ADDR ← INVPCID_DESC[127:64];
    Invalidate mappings for L_ADDR associated with PCID except global translations;
    BREAK;
  1:           // single PCID invalidation
    PCID ← INVPCID_DESC[11:0];
    Invalidate all mappings associated with PCID except global translations;
    BREAK;
  2:           // all PCID invalidation including global translations
    Invalidate all mappings for all PCIDs, including global translations;
    BREAK;
  3:           // all PCID invalidation retaining global translations
    Invalidate all mappings for all PCIDs except global translations;
    BREAK;
ESAC;

```

Intel C/C++ Compiler Intrinsic Equivalent

INVPCID: void _invpcid(unsigned __int32 type, void * descriptor);

SIMD Floating-Point Exceptions

None

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 source operand is located in an execute-only code segment. If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3. If bits 63:12 of INVPCID_DESC are not all zero. If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero. If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#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 if CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0. If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3. If bits 63:12 of INVPCID_DESC are not all zero.
-----	---

- If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.
- If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
- #UD If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.
- If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #GP(0) The INVPCID 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 memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
 - If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
 - If bits 63:12 of INVPCID_DESC are not all zero.
 - If CR4.PCIDE=0, INVPCID_TYPE is either 0 or 1, and INVPCID_DESC[11:0] is not zero.
 - If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
- #PF(fault-code) If a page fault occurs in accessing the memory 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 the LOCK prefix is used.
- If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.

IRET/IROTD—Interrupt Return

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CF	IRET	NP	Valid	Valid	Interrupt return (16-bit operand size).
CF	IROTD	NP	Valid	Valid	Interrupt return (32-bit operand size).
REX.W + CF	IRETQ	NP	Valid	N.E.	Interrupt return (64-bit operand size).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled “Task Linking” in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

IRET and IROTD are mnemonics for the same opcode. The IROTD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction performs a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack).

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or interrupted task. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is re-entered later, the code that follows the IRET instruction is executed.

If the NT flag is set and the processor is in IA-32e mode, the IRET instruction causes a general protection exception.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.W prefix promotes operation to 64 bits (IRETQ). See the summary chart at the beginning of this section for encoding data and limits.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```

IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE
        IF (IA32_EFER.LMA = 0)
            THEN (* Protected mode *)
                GOTO PROTECTED-MODE;
            ELSE (* IA-32e mode *)
                GOTO IA-32e-MODE;
        Fi;
    Ff;
REAL-ADDRESS-MODE;
IF OperandSize = 32
    THEN
        IF top 12 bytes of stack not within stack limits
            THEN #SS; Fi;
        tempEIP ← 4 bytes at end of stack
        IF tempEIP[31:16] is not zero THEN #GP(0); Fi;
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        tempEFLAGS ← Pop();
        EFLAGS ← (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
    ELSE (* OperandSize = 16 *)
        IF top 6 bytes of stack are not within stack limits
            THEN #SS; Fi;
        EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
        CS ← Pop(); (* 16-bit pop *)
        EFLAGS[15:0] ← Pop();
    Fi;
END;
PROTECTED-MODE:
IF VM = 1 (* Virtual-8086 mode: PE = 1, VM = 1 *)
    THEN
        GOTO RETURN-FROM-VIRTUAL-8086-MODE; (* PE = 1, VM = 1 *)
    Ff;
IF NT = 1
    THEN
        GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
    Ff;
IF OperandSize = 32
    THEN
        IF top 12 bytes of stack not within stack limits
            THEN #SS(0); Fi;
        tempEIP ← Pop();
        tempCS ← Pop();
        tempEFLAGS ← Pop();
    ELSE (* OperandSize = 16 *)
        IF top 6 bytes of stack are not within stack limits

```

```

        THEN #SS(0); Fl;
tempEIP ← Pop();
tempCS ← Pop();
tempEFLAGS ← Pop();
tempEIP ← tempEIP AND FFFFH;
tempEFLAGS ← tempEFLAGS AND FFFFH;

Fl;
IF tempEFLAGS(VM) = 1 and CPL = 0
    THEN
        GOTO RETURN-TO-VIRTUAL-8086-MODE;
    ELSE
        GOTO PROTECTED-MODE-RETURN;
Fl;
IA-32e-MODE:
IF NT = 1
    THEN #GP(0);
ELSE IF OperandSize = 32
    THEN
        IF top 12 bytes of stack not within stack limits
            THEN #SS(0); Fl;
        tempEIP ← Pop();
        tempCS ← Pop();
        tempEFLAGS ← Pop();
    ELSE IF OperandSize = 16
        THEN
            IF top 6 bytes of stack are not within stack limits
                THEN #SS(0); Fl;
            tempEIP ← Pop();
            tempCS ← Pop();
            tempEFLAGS ← Pop();
            tempEIP ← tempEIP AND FFFFH;
            tempEFLAGS ← tempEFLAGS AND FFFFH;

        Fl;
    ELSE (* OperandSize = 64 *)
        THEN
            tempRIP ← Pop();
            tempCS ← Pop();
            tempEFLAGS ← Pop();
            tempRSP ← Pop();
            tempSS ← Pop();

        Fl;
    GOTO IA-32e-MODE-RETURN;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
IF IOPL = 3 (* Virtual mode: PE = 1, VM = 1, IOPL = 3 *)
    THEN IF OperandSize = 32
        THEN
            IF top 12 bytes of stack not within stack limits
                THEN #SS(0); Fl;
            IF instruction pointer not within code segment limits
                THEN #GP(0); Fl;
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)

```

INSTRUCTION SET REFERENCE, A-M

```
EFLAGS ← Pop();
(* VM, IOPL,VIP and VIF EFLAG bits not modified by pop *)
ELSE (* OperandSize = 16 *)
    IF top 6 bytes of stack are not within stack limits
        THEN #SS(0); Fl;
    IF instruction pointer not within code segment limits
        THEN #GP(0); Fl;
    EIP ← Pop();
    EIP ← EIP AND 0000FFFFH;
    CS ← Pop(); (* 16-bit pop *)
    EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS not modified by pop *)
Fl;
ELSE
    #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
Fl;
END;
```

RETURN-TO-VIRTUAL-8086-MODE:

```
(* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
IF top 24 bytes of stack are not within stack segment limits
    THEN #SS(0); Fl;
IF instruction pointer not within code segment limits
    THEN #GP(0); Fl;
CS ← tempCS;
EIP ← tempEIP & FFFFH;
EFLAGS ← tempEFLAGS;
TempESP ← Pop();
TempSS ← Pop();
ES ← Pop(); (* Pop 2 words; throw away high-order word *)
DS ← Pop(); (* Pop 2 words; throw away high-order word *)
FS ← Pop(); (* Pop 2 words; throw away high-order word *)
GS ← Pop(); (* Pop 2 words; throw away high-order word *)
SS:ESP ← TempSS:TempESP;
CPL ← 3;
(* Resume execution in Virtual-8086 mode *)
END;
```

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)

```
Read segment selector in link field of current TSS;
IF local/global bit is set to local
or index not within GDT limits
    THEN #TS (TSS selector); Fl;
Access TSS for task specified in link field of current TSS;
IF TSS descriptor type is not TSS or if the TSS is marked not busy
    THEN #TS (TSS selector); Fl;
IF TSS not present
    THEN #NP(TSS selector); Fl;
SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
Mark the task just abandoned as NOT BUSY;
IF EIP is not within code segment limit
    THEN #GP(0); Fl;
END;
```

PROTECTED-MODE-RETURN: (* PE = 1 *)

```

IF return code segment selector is NULL
    THEN GP(0); Fl;
IF return code segment selector addresses descriptor beyond descriptor table limit
    THEN GP(selector); Fl;
Read segment descriptor pointed to by the return code segment selector;
IF return code segment descriptor is not a code segment
    THEN #GP(selector); Fl;
IF return code segment selector RPL < CPL
    THEN #GP(selector); Fl;
IF return code segment descriptor is conforming
and return code segment DPL > return code segment selector RPL
    THEN #GP(selector); Fl;
IF return code segment descriptor is not present
    THEN #NP(selector); Fl;
IF return code segment selector RPL > CPL
    THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
    ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; Fl;
END;

```

```

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, RPL = CPL *)
IF new mode ≠ 64-Bit Mode
    THEN
        IF tempEIP is not within code segment limits
            THEN #GP(0); Fl;
            EIP ← tempEIP;
        ELSE (* new mode = 64-bit mode *)
            IF tempRIP is non-canonical
                THEN #GP(0); Fl;
            RIP ← tempRIP;
        Fl;
        CS ← tempCS; (* Segment descriptor information also loaded *)
        EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
        IF OperandSize = 32 or OperandSize = 64
            THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; Fl;
        IF CPL ≤ IOPL
            THEN EFLAGS(IF) ← tempEFLAGS; Fl;
        IF CPL = 0
            THEN (* VM = 0 in flags image *)
                EFLAGS(IOPL) ← tempEFLAGS;
            IF OperandSize = 32 or OperandSize = 64
                THEN EFLAGS(VIF, VIP) ← tempEFLAGS; Fl;
        Fl;
    END;

```

```

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
IF OperandSize = 32
    THEN
        IF top 8 bytes on stack are not within limits
            THEN #SS(0); Fl;
        ELSE (* OperandSize = 16 *)
            IF top 4 bytes on stack are not within limits
                THEN #SS(0); Fl;
    Fl;
Read return segment selector;

```

INSTRUCTION SET REFERENCE, A-M

IF stack segment selector is NULL
 THEN #GP(0); Fl;
IF return stack segment selector index is not within its descriptor table limits
 THEN #GP(SSselector); Fl;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or the stack segment descriptor does not indicate a a writable data segment;
or the stack segment DPL ≠ RPL of the return code segment selector
 THEN #GP(SS selector); Fl;
IF stack segment is not present
 THEN #SS(SS selector); Fl;
IF new mode ≠ 64-Bit Mode
 THEN
 IF tempEIP is not within code segment limits
 THEN #GP(0); Fl;
 EIP ← tempEIP;
 ELSE (* new mode = 64-bit mode *)
 IF tempRIP is non-canonical
 THEN #GP(0); Fl;
 RIP ← tempRIP;
 Fl;
 CS ← tempCS;
 EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
 IF OperandSize = 32
 THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; Fl;
 IF CPL ≤ IOPL
 THEN EFLAGS(IF) ← tempEFLAGS; Fl;
 IF CPL = 0
 THEN
 EFLAGS(IOPL) ← tempEFLAGS;
 IF OperandSize = 32
 THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS; Fl;
 IF OperandSize = 64
 THEN EFLAGS(VIF, VIP) ← tempEFLAGS; Fl;
 Fl;
 CPL ← RPL of the return code segment selector;
 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 (* Stored in hidden part of segment register *)
 THEN (* Segment register invalid *)
 SegmentSelector ← 0; (* NULL segment selector *)
 Fl;
 OD;
 END;
IA-32e-MODE-RETURN: (* IA32_EFER.LMA = 1, PE = 1 *)
IF ((return code segment selector is NULL) or (return RIP is non-canonical) or
 (SS selector is NULL going back to compatibility mode) or
 (SS selector is NULL going back to CPL3 64-bit mode) or
 (RPL <> CPL going back to non-CPL3 64-bit mode for a NULL SS selector))
 THEN GP(0); Fl;
IF return code segment selector addresses descriptor beyond descriptor table limit
 THEN GP(selector); Fl;

```

Read segment descriptor pointed to by the return code segment selector;
IF return code segment descriptor is not a code segment
    THEN #GP(selector); Fl;
IF return code segment selector RPL < CPL
    THEN #GP(selector); Fl;
IF return code segment descriptor is conforming
and return code segment DPL > return code segment selector RPL
    THEN #GP(selector); Fl;
IF return code segment descriptor is not present
    THEN #NP(selector); Fl;
IF return code segment selector RPL > CPL
    THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
    ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; Fl;
END;

```

Flags Affected

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task's TSS.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is NULL. If the return instruction pointer is not within the return code segment limit.
#GP(selector)	If a segment selector index is outside its descriptor table limits. If the return code segment selector RPL is less than the CPL. If the DPL of a conforming-code segment is greater than the return code segment selector RPL. If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector. If the stack segment descriptor DPL is not equal to the RPL of the return code 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 segment descriptor for a code segment does not indicate it is a code segment. If the segment selector for a TSS has its local/global bit set for local. If a TSS segment descriptor specifies that the TSS is not busy. If a TSS segment descriptor specifies that the TSS is not available.
#SS(0)	If the top bytes of stack are not within stack limits.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

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.
--------	--

	IF IOPL not equal to 3.
#PF(fault-code)	If a page fault occurs.
#SS(0)	If the top bytes of stack are not within stack limits.
#AC(0)	If an unaligned memory reference occurs and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#GP(0) If EFLAGS.NT[bit 14] = 1.

Other exceptions same as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If EFLAGS.NT[bit 14] = 1. If the return code segment selector is NULL. 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 instruction pointer is not within the return code segment limit. If the return instruction pointer is non-canonical.
#GP(Selector)	If a segment selector index is outside its descriptor table limits. If a segment descriptor memory address is non-canonical. If the 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 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.
#SS(0)	If the stack segment selector RPL is not equal to the RPL of the return code segment selector. 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 an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Jcc—Jump if Condition Is Met

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
77 cb	JA rel8	D	Valid	Valid	Jump short if above (CF=0 and ZF=0).
73 cb	JAE rel8	D	Valid	Valid	Jump short if above or equal (CF=0).
72 cb	JB rel8	D	Valid	Valid	Jump short if below (CF=1).
76 cb	JBE rel8	D	Valid	Valid	Jump short if below or equal (CF=1 or ZF=1).
72 cb	JC rel8	D	Valid	Valid	Jump short if carry (CF=1).
E3 cb	JCXZ rel8	D	N.E.	Valid	Jump short if CX register is 0.
E3 cb	JECXZ rel8	D	Valid	Valid	Jump short if ECX register is 0.
E3 cb	JRCXZ rel8	D	Valid	N.E.	Jump short if RCX register is 0.
74 cb	JE rel8	D	Valid	Valid	Jump short if equal (ZF=1).
7F cb	JG rel8	D	Valid	Valid	Jump short if greater (ZF=0 and SF=OF).
7D cb	JGE rel8	D	Valid	Valid	Jump short if greater or equal (SF=OF).
7C cb	JL rel8	D	Valid	Valid	Jump short if less (SF≠ OF).
7E cb	JLE rel8	D	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠ OF).
76 cb	JNA rel8	D	Valid	Valid	Jump short if not above (CF=1 or ZF=1).
72 cb	JNAE rel8	D	Valid	Valid	Jump short if not above or equal (CF=1).
73 cb	JNB rel8	D	Valid	Valid	Jump short if not below (CF=0).
77 cb	JNBE rel8	D	Valid	Valid	Jump short if not below or equal (CF=0 and ZF=0).
73 cb	JNC rel8	D	Valid	Valid	Jump short if not carry (CF=0).
75 cb	JNE rel8	D	Valid	Valid	Jump short if not equal (ZF=0).
7E cb	JNG rel8	D	Valid	Valid	Jump short if not greater (ZF=1 or SF≠ OF).
7C cb	JNGE rel8	D	Valid	Valid	Jump short if not greater or equal (SF≠ OF).
7D cb	JNL rel8	D	Valid	Valid	Jump short if not less (SF=OF).
7F cb	JNLE rel8	D	Valid	Valid	Jump short if not less or equal (ZF=0 and SF=OF).
71 cb	JNO rel8	D	Valid	Valid	Jump short if not overflow (OF=0).
7B cb	JNP rel8	D	Valid	Valid	Jump short if not parity (PF=0).
79 cb	JNS rel8	D	Valid	Valid	Jump short if not sign (SF=0).
75 cb	JNZ rel8	D	Valid	Valid	Jump short if not zero (ZF=0).
70 cb	JO rel8	D	Valid	Valid	Jump short if overflow (OF=1).
7A cb	JP rel8	D	Valid	Valid	Jump short if parity (PF=1).
7A cb	JPE rel8	D	Valid	Valid	Jump short if parity even (PF=1).
7B cb	JPO rel8	D	Valid	Valid	Jump short if parity odd (PF=0).
78 cb	JS rel8	D	Valid	Valid	Jump short if sign (SF=1).
74 cb	JZ rel8	D	Valid	Valid	Jump short if zero (ZF ← 1).
0F 87 cw	JA rel16	D	N.S.	Valid	Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 cd	JAE rel32	D	Valid	Valid	Jump near if above (CF=0 and ZF=0).
0F 83 cw	JAE rel16	D	N.S.	Valid	Jump near if above or equal (CF=0). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 83 cd	JAE rel32	D	Valid	Valid	Jump near if above or equal (CF=0).
0F 82 cw	JB rel16	D	N.S.	Valid	Jump near if below (CF=1). Not supported in 64-bit mode.
0F 82 cd	JB rel32	D	Valid	Valid	Jump near if below (CF=1).
0F 86 cw	JBE rel16	D	N.S.	Valid	Jump near if below or equal (CF=1 or ZF=1). Not supported in 64-bit mode.
0F 86 cd	JBE rel32	D	Valid	Valid	Jump near if below or equal (CF=1 or ZF=1).
0F 82 cw	JC rel16	D	N.S.	Valid	Jump near if carry (CF=1). Not supported in 64-bit mode.
0F 82 cd	JC rel32	D	Valid	Valid	Jump near if carry (CF=1).
0F 84 cw	JE rel16	D	N.S.	Valid	Jump near if equal (ZF=1). Not supported in 64-bit mode.
0F 84 cd	JE rel32	D	Valid	Valid	Jump near if equal (ZF=1).
0F 84 cw	JZ rel16	D	N.S.	Valid	Jump near if 0 (ZF=1). Not supported in 64-bit mode.
0F 84 cd	JZ rel32	D	Valid	Valid	Jump near if 0 (ZF=1).
0F 8F cw	JG rel16	D	N.S.	Valid	Jump near if greater (ZF=0 and SF=OF). Not supported in 64-bit mode.
0F 8F cd	JG rel32	D	Valid	Valid	Jump near if greater (ZF=0 and SF=OF).
0F 8D cw	JGE rel16	D	N.S.	Valid	Jump near if greater or equal (SF=OF). Not supported in 64-bit mode.
0F 8D cd	JGE rel32	D	Valid	Valid	Jump near if greater or equal (SF=OF).
0F 8C cw	JL rel16	D	N.S.	Valid	Jump near if less (SF≠ OF). Not supported in 64-bit mode.
0F 8C cd	JL rel32	D	Valid	Valid	Jump near if less (SF≠ OF).
0F 8E cw	JLE rel16	D	N.S.	Valid	Jump near if less or equal (ZF=1 or SF≠ OF). Not supported in 64-bit mode.
0F 8E cd	JLE rel32	D	Valid	Valid	Jump near if less or equal (ZF=1 or SF≠ OF).
0F 86 cw	JNA rel16	D	N.S.	Valid	Jump near if not above (CF=1 or ZF=1). Not supported in 64-bit mode.
0F 86 cd	JNA rel32	D	Valid	Valid	Jump near if not above (CF=1 or ZF=1).
0F 82 cw	JNAE rel16	D	N.S.	Valid	Jump near if not above or equal (CF=1). Not supported in 64-bit mode.
0F 82 cd	JNAE rel32	D	Valid	Valid	Jump near if not above or equal (CF=1).
0F 83 cw	JNB rel16	D	N.S.	Valid	Jump near if not below (CF=0). Not supported in 64-bit mode.
0F 83 cd	JNB rel32	D	Valid	Valid	Jump near if not below (CF=0).
0F 87 cw	JNBE rel16	D	N.S.	Valid	Jump near if not below or equal (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 cd	JNBE rel32	D	Valid	Valid	Jump near if not below or equal (CF=0 and ZF=0).
0F 83 cw	JNC rel16	D	N.S.	Valid	Jump near if not carry (CF=0). Not supported in 64-bit mode.
0F 83 cd	JNC rel32	D	Valid	Valid	Jump near if not carry (CF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 85 cw	JNE rel16	D	N.S.	Valid	Jump near if not equal (ZF=0). Not supported in 64-bit mode.
0F 85 cd	JNE rel32	D	Valid	Valid	Jump near if not equal (ZF=0).
0F 8E cw	JNG rel16	D	N.S.	Valid	Jump near if not greater (ZF=1 or SF ≠ OF). Not supported in 64-bit mode.
0F 8E cd	JNG rel32	D	Valid	Valid	Jump near if not greater (ZF=1 or SF ≠ OF).
0F 8C cw	JNGE rel16	D	N.S.	Valid	Jump near if not greater or equal (SF ≠ OF). Not supported in 64-bit mode.
0F 8C cd	JNGE rel32	D	Valid	Valid	Jump near if not greater or equal (SF ≠ OF).
0F 8D cw	JNL rel16	D	N.S.	Valid	Jump near if not less (SF=OF). Not supported in 64-bit mode.
0F 8D cd	JNL rel32	D	Valid	Valid	Jump near if not less (SF=OF).
0F 8F cw	JNLE rel16	D	N.S.	Valid	Jump near if not less or equal (ZF=0 and SF=OF). Not supported in 64-bit mode.
0F 8F cd	JNLE rel32	D	Valid	Valid	Jump near if not less or equal (ZF=0 and SF=OF).
0F 81 cw	JNO rel16	D	N.S.	Valid	Jump near if not overflow (OF=0). Not supported in 64-bit mode.
0F 81 cd	JNO rel32	D	Valid	Valid	Jump near if not overflow (OF=0).
0F 8B cw	JNP rel16	D	N.S.	Valid	Jump near if not parity (PF=0). Not supported in 64-bit mode.
0F 8B cd	JNP rel32	D	Valid	Valid	Jump near if not parity (PF=0).
0F 89 cw	JNS rel16	D	N.S.	Valid	Jump near if not sign (SF=0). Not supported in 64-bit mode.
0F 89 cd	JNS rel32	D	Valid	Valid	Jump near if not sign (SF=0).
0F 85 cw	JNZ rel16	D	N.S.	Valid	Jump near if not zero (ZF=0). Not supported in 64-bit mode.
0F 85 cd	JNZ rel32	D	Valid	Valid	Jump near if not zero (ZF=0).
0F 80 cw	JO rel16	D	N.S.	Valid	Jump near if overflow (OF=1). Not supported in 64-bit mode.
0F 80 cd	JO rel32	D	Valid	Valid	Jump near if overflow (OF=1).
0F 8A cw	JP rel16	D	N.S.	Valid	Jump near if parity (PF=1). Not supported in 64-bit mode.
0F 8A cd	JP rel32	D	Valid	Valid	Jump near if parity (PF=1).
0F 8A cw	JPE rel16	D	N.S.	Valid	Jump near if parity even (PF=1). Not supported in 64-bit mode.
0F 8A cd	JPE rel32	D	Valid	Valid	Jump near if parity even (PF=1).
0F 8B cw	JPO rel16	D	N.S.	Valid	Jump near if parity odd (PF=0). Not supported in 64-bit mode.
0F 8B cd	JPO rel32	D	Valid	Valid	Jump near if parity odd (PF=0).
0F 88 cw	JS rel16	D	N.S.	Valid	Jump near if sign (SF=1). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 88 cd	JS rel32	D	Valid	Valid	Jump near if sign (SF=1).
0F 84 cw	JZ rel16	D	N.S.	Valid	Jump near if 0 (ZF=1). Not supported in 64-bit mode.
0F 84 cd	JZ rel32	D	Valid	Valid	Jump near if 0 (ZF=1).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA

Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the Jcc instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (*rel8*, *rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of -128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

The conditions for each Jcc mnemonic are given in the "Description" column of the table on the preceding page. The terms "less" and "greater" are used for comparisons of signed integers and the terms "above" and "below" are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The Jcc instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the Jcc instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

```
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

```
JNZ BEYOND;  
JMP FARLABEL;  
BEYOND:
```

The JR CXZ, JECXZ and JCXZ instructions differ from other Jcc instructions because they do not check status flags. Instead, they check RCX, ECX or CX for 0. The register checked is determined by the address-size attribute. These instructions are useful when used at the beginning of a loop that terminates with a conditional loop instruction (such as LOOPNE). They can be used to prevent an instruction sequence from entering a loop when RCX, ECX or CX is 0. This would cause the loop to execute 2^{64} , 2^{32} or 64K times (not zero times).

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

In 64-bit mode, operand size is fixed at 64 bits. JMP Short is RIP = RIP + 8-bit offset sign extended to 64 bits. JMP Near is RIP = RIP + 32-bit offset sign extended to 64-bits.

Operation

```

IF condition
  THEN
    tempEIP ← EIP + SignExtend(DEST);
    IF OperandSize = 16
      THEN tempEIP ← tempEIP AND 0000FFFFH;
    FI;
  IF tempEIP is not within code segment limit
    THEN #GP(0);
    ELSE EIP ← tempEIP
  FI;
FI;

```

Protected Mode Exceptions

- #GP(0) If the offset being jumped to is beyond the limits of the CS segment.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #GP If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.
- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address 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.
- #UD If the LOCK prefix is used.

JMP—Jump

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
EB cb	JMP rel8	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 cw	JMP rel16	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 cd	JMP rel32	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits
FF /4	JMP r/m16	M	N.S.	Valid	Jump near, absolute indirect, address = zero-extended r/m16. Not supported in 64-bit mode.
FF /4	JMP r/m32	M	N.S.	Valid	Jump near, absolute indirect, address given in r/m32. Not supported in 64-bit mode.
FF /4	JMP r/m64	M	Valid	N.E.	Jump near, absolute indirect, RIP = 64-Bit offset from register or memory
EA cd	JMP ptr16:16	D	Inv.	Valid	Jump far, absolute, address given in operand
EA cp	JMP ptr16:32	D	Inv.	Valid	Jump far, absolute, address given in operand
FF /5	JMP m16:16	D	Valid	Valid	Jump far, absolute indirect, address given in m16:16
FF /5	JMP m16:32	D	Valid	Valid	Jump far, absolute indirect, address given in m16:32.
REX.W + FF /5	JMP m16:64	D	Valid	N.E.	Jump far, absolute indirect, address given in m16:64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2		Operand 3	Operand 4
		NA	NA		
D	Offset			NA	NA
M	ModRM:r/m (r)	NA		NA	NA

Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- Short jump—A near jump where the jump range is limited to -128 to +127 from the current EIP value.
- Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- Task switch—A jump to an instruction located in a different task.

A task switch can only be executed in protected mode (see Chapter 7, in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for information on performing task switches with the JMP instruction).

Near and Short Jumps. When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current

value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (*rel8*) is referred to as a short jump. The CS register is not changed on near and short jumps.

An absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16* or *r/m32*). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

A relative offset (*rel8*, *rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

Far Jumps in Real-Address or Virtual-8086 Mode. When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

Far Jumps in Protected Mode. When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps:

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.
- A task switch.

(The JMP instruction cannot be used to perform inter-privilege-level far jumps.)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of jump to be performed.

If the selected descriptor is for a code segment, a far jump to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far jump to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making jumps between 16-bit and 32-bit code segments.

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

Executing a task switch with the JMP instruction is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into the EIP register so that the task begins executing again at this next instruction.

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7 in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for detailed information on the mechanics of a task switch.

Note that when you execute a task switch with a JMP instruction, the nested task flag (NT) is not set in the EFLAGS register and the new TSS's previous task link field is not loaded with the old task's TSS selector. A return to the previous task can thus not be carried out by executing the IRET instruction. Switching tasks with the JMP instruction differs in this regard from the CALL instruction which does set the NT flag and save the previous task link information, allowing a return to the calling task with an IRET instruction.

In 64-Bit Mode — The instruction's operation size is fixed at 64 bits. If a selector points to a gate, then RIP equals the 64-bit displacement taken from gate; else RIP equals the zero-extended offset from the far pointer referenced in the instruction.

See the summary chart at the beginning of this section for encoding data and limits.

Operation

```

IF near jump
  IF 64-bit Mode
    THEN
      IF near relative jump
        THEN
          tempRIP ← RIP + DEST; (* RIP is instruction following JMP instruction*)
        ELSE (* Near absolute jump *)
          tempRIP ← DEST;
        FI;
    ELSE
      IF near relative jump
        THEN
          tempEIP ← EIP + DEST; (* EIP is instruction following JMP instruction*)
        ELSE (* Near absolute jump *)
          tempEIP ← DEST;
        FI;
    FI;
  IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode)
  and tempEIP outside code segment limit
    THEN #GP(0); FI
  IF 64-bit mode and tempRIP is not canonical
    THEN #GP(0);
  FI;
  IF OperandSize = 32
    THEN
      EIP ← tempEIP;
    ELSE
      IF OperandSize = 16
        THEN (* OperandSize = 16 *)
          EIP ← tempEIP AND 0000FFFFH;
        ELSE (* OperandSize = 64)
          RIP ← tempRIP;
        FI;
    FI;
  FI;
IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
  THEN
    tempEIP ← DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
    IF tempEIP is beyond code segment limit
      THEN #GP(0); FI;
    CS ← DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
    IF OperandSize = 32

```

```

THEN
    EIP ← tempEIP; (* DEST is ptr16:32 or [m16:32] *)
ELSE (* OperandSize = 16 *)
    EIP ← tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
FI;
IF far jump and (PE = 1 and VM = 0)
(* IA-32e mode or protected mode, not virtual-8086 mode *)
THEN
    IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
    or segment selector in target operand NULL
        THEN #GP(0); FI;
    IF segment selector index not within descriptor table limits
        THEN #GP(new selector); FI;
    Read type and access rights of segment descriptor;
    IF (EFER.LMA = 0)
        THEN
            IF segment type is not a conforming or nonconforming code
            segment, call gate, task gate, or TSS
                THEN #GP(segment selector); FI;
        ELSE
            IF segment type is not a conforming or nonconforming code segment
            call gate
                THEN #GP(segment selector); FI;
        FI;
    Depending on type and access rights:
        GO TO CONFORMING-CODE-SEGMENT;
        GO TO NONCONFORMING-CODE-SEGMENT;
        GO TO CALL-GATE;
        GO TO TASK-GATE;
        GO TO TASK-STATE-SEGMENT;
ELSE
    #GP(segment selector);
FI;
CONFORMING-CODE-SEGMENT:
IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); FI;
IF DPL > CPL
    THEN #GP(segment selector); FI;
IF segment not present
    THEN #NP(segment selector); FI;
tempEIP ← DEST(Offset);
IF OperandSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH;
FI;
IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and
tempEIP outside code segment limit
    THEN #GP(0); FI
IF tempEIP is non-canonical
    THEN #GP(0); FI;
CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
CS(RPL) ← CPL
EIP ← tempEIP;
END;

```

INSTRUCTION SET REFERENCE, A-M

NONCONFORMING-CODE-SEGMENT:

```
IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); Fl;
IF (RPL > CPL) OR (DPL ≠ CPL)
    THEN #GP(code segment selector); Fl;
IF segment not present
    THEN #NP(segment selector); Fl;
tempEIP ← DEST(Offset);
IF OperandSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH; Fl;
IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode)
and tempEIP outside code segment limit
    THEN #GP(0); Fl
IF tempEIP is non-canonical THEN #GP(0); Fl;
CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
CS(RPL) ← CPL;
EIP ← tempEIP;
END;
```

CALL-GATE:

```
IF call gate DPL < CPL
or call gate DPL < call gate segment-selector RPL
    THEN #GP(call gate selector); Fl;
IF call gate not present
    THEN #NP(call gate selector); Fl;
IF call gate code-segment selector is NULL
    THEN #GP(0); Fl;
IF call gate code-segment selector index outside descriptor table limits
    THEN #GP(code segment selector); Fl;
Read code segment descriptor;
IF code-segment segment descriptor does not indicate a code segment
or code-segment segment descriptor is conforming and DPL > CPL
or code-segment segment descriptor is non-conforming and DPL ≠ CPL
    THEN #GP(code segment selector); Fl;
IF IA32_EFER.LMA = 1 and (code-segment descriptor is not a 64-bit code segment
or code-segment segment descriptor has both L-Bit and D-bit set)
    THEN #GP(code segment selector); Fl;
```

```
IF code segment is not present
    THEN #NP(code-segment selector); Fl;
IF instruction pointer is not within code-segment limit
    THEN #GP(0); Fl;
tempEIP ← DEST(Offset);
IF GateSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH; Fl;
IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
outside code segment limit
    THEN #GP(0); Fl
CS ← DEST[SegmentSelector]; (* Segment descriptor information also loaded *)
CS(RPL) ← CPL;
EIP ← tempEIP;
```

```
END;
```

TASK-GATE:

```
IF task gate DPL < CPL
or task gate DPL < task gate segment-selector RPL
```

```

THEN #GP(task gate selector); Fl;
IF task gate not present
    THEN #NP(gate selector); Fl;
Read the TSS segment selector in the task-gate descriptor;
IF TSS segment selector local/global bit is set to local
or index not within GDT limits
or TSS descriptor specifies that the TSS is busy
    THEN #GP(TSS selector); Fl;
IF TSS not present
    THEN #NP(TSS selector); Fl;
SWITCH-TASKS to TSS;
IF EIP not within code segment limit
    THEN #GP(0); Fl;
END;
TASK-STATE-SEGMENT:
IF TSS DPL < CPL
or TSS DPL < TSS segment-selector RPL
or TSS descriptor indicates TSS not available
    THEN #GP(TSS selector); Fl;
IF TSS is not present
    THEN #NP(TSS selector); Fl;
SWITCH-TASKS to TSS;
IF EIP not within code segment limit
    THEN #GP(0); Fl;
END;

```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions

#GP(0)	If offset in target operand, call gate, or TSS is beyond the code segment limits. If the segment selector in the destination operand, call gate, task gate, or TSS is NULL. 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 the segment selector index is outside descriptor table limits. If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment. If the DPL for a nonconforming-code segment is not equal to the CPL (When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL. If the DPL for a conforming-code segment is greater than the CPL. If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector. If the segment descriptor for selector in a call gate does not indicate it is a code segment. If the segment descriptor for the segment selector in a task gate does not indicate an available TSS. If the segment selector for a TSS has its local/global bit set for local. If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#NP (selector)	If the code segment being accessed is not present.
	If call gate, task gate, or TSS 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. (Only occurs when fetching target from memory.)
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#GP(0)	If the target operand is beyond the code segment limits.
	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. (Only occurs when fetching target from memory.)
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical.
	If target offset in destination operand is non-canonical.
	If target offset in destination operand is beyond the new code segment limit.
	If the segment selector in the destination operand is NULL.
	If the code segment selector in the 64-bit gate is NULL.
#GP(selector)	If the code segment or 64-bit call gate is outside descriptor table limits.
	If the code segment or 64-bit call gate overlaps non-canonical space.
	If the segment descriptor from a 64-bit call gate is in non-canonical space.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, 64-bit call gate.
	If the segment descriptor pointed to by the segment selector in the destination operand is a code segment, and has both the D-bit and the L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
	If the upper type field of a 64-bit call gate is not 0x0.
	If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
	If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
	If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.

- If the code segment is non-confirming and CPL ≠ DPL.
- If the code segment is confirming and CPL < DPL.
- If a code segment or 64-bit call gate is not present.
- (64-bit mode only) If a far jump is direct to an absolute address in memory.
- If the LOCK prefix is used.
- If a page fault occurs.
- If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

LAHF—Load Status Flags into AH Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9F	LAHF	NP	Invalid*	Valid	Load: AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF).

NOTES:

*Valid in specific steppings. See Description section.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

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 64-Bit Mode
THEN
  IF CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1;
    THEN AH ← RFLAGS(SF:ZF:0:AF:0:PF:1:CF);
    ELSE #UD;
  FI;
ELSE
  AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF);
FI;
```

Flags Affected

None. The state of the flags in the EFLAGS register is not affected.

Protected Mode Exceptions

#UD If the LOCK prefix is used.

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

#UD If CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 0.
 If the LOCK prefix is used.

LAR—Load Access Rights Byte

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 02 /r	LAR <i>r16, r16/m16</i>	RM	Valid	Valid	<i>r16</i> ← access rights referenced by <i>r16/m16</i>
OF 02 /r	LAR <i>reg, r32/m16</i> ¹	RM	Valid	Valid	<i>reg</i> ← access rights referenced by <i>r32/m16</i>

NOTES:

1. For all loads (regardless of source or destination sizing) only bits 16-0 are used. Other bits are ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the flag register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. If the source operand is a memory address, only 16 bits of data are accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can perform additional checks on the access rights information.

The access rights for a segment descriptor include fields located in the second doubleword (bytes 4–7) of the segment descriptor. The following fields are loaded by the LAR instruction:

- Bits 7:0 are returned as 0
- Bits 11:8 return the segment type.
- Bit 12 returns the S flag.
- Bits 14:13 return the DPL.
- Bit 15 returns the P flag.
- The following fields are returned only if the operand size is greater than 16 bits:
 - Bits 19:16 are undefined.
 - Bit 20 returns the software-available bit in the descriptor.
 - Bit 21 returns the L flag.
 - Bit 22 returns the D/B flag.
 - Bit 23 returns the G flag.
 - Bits 31:24 are returned as 0.

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in Table 3-62.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.

The LAR instruction can only be executed in protected mode and IA-32e mode.

Table 3-62. Segment and Gate Types

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Reserved	No
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT	No
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	Yes	Reserved	No
5	16-bit/32-bit task gate	Yes	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	Available 64-bit TSS	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS	Yes
C	32-bit call gate	Yes	64-bit call gate	Yes
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

Operation

```

IF Offset(SRC) > descriptor table limit
  THEN
    ZF ← 0;
  ELSE
    SegmentDescriptor ← descriptor referenced by SRC;
    IF SegmentDescriptor(Type) ≠ conforming code segment
    and (CPL > DPL) or (RPL > DPL)
    or SegmentDescriptor(Type) is not valid for instruction
      THEN
        ZF ← 0;
      ELSE
        DEST ← access rights from SegmentDescriptor as given in Description section;
        ZF ← 1;
      FI;
    FI;
  
```

Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.

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 the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The LAR instruction is not recognized in real-address mode.
-----	---

Virtual-8086 Mode Exceptions

#UD	The LAR instruction cannot be executed in virtual-8086 mode.
-----	--

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

LDDQU—Load Unaligned Integer 128 Bits

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF F0 /r LDDQU <i>xmm1, mem</i>	RM	V/V	SSE3	Load unaligned data from <i>mem</i> and return double quadword in <i>xmm1</i> .
VEX.128.F2.0F.WIG F0 /r VLDDQU <i>xmm1, m128</i>	RM	V/V	AVX	Load unaligned packed integer values from <i>mem</i> to <i>xmm1</i> .
VEX.256.F2.0F.WIG F0 /r VLDDQU <i>ymm1, m256</i>	RM	V/V	AVX	Load unaligned packed integer values from <i>mem</i> to <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The instruction is *functionally similar* to (V)MOVDQU *ymm/xmm, m256/m128* for loading from memory. That is: 32/16 bytes of data starting at an address specified by the source memory operand (second operand) are fetched from memory and placed in a destination register (first operand). The source operand need not be aligned on a 32/16-byte boundary. Up to 64/32 bytes may be loaded from memory; this is implementation dependent.

This instruction may improve performance relative to (V)MOVDQU if the source operand crosses a cache line boundary. In situations that require the data loaded by (V)LDDQU be modified and stored to the same location, use (V)MOVDQU or (V)MOVDQA instead of (V)LDDQU. To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the (V)MOVDQA instruction.

Implementation Notes

- If the source is aligned to a 32/16-byte boundary, based on the implementation, the 32/16 bytes may be loaded more than once. For that reason, the usage of (V)LDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using (V)MOVDQU.
- This instruction is a replacement for (V)MOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use (V)MOVDQA store-load pairs when data is 256/128-bit aligned or (V)MOVDQU store-load pairs when data is 256/128-bit unaligned.
- If the memory address is not aligned on 32/16-byte boundary, some implementations may load up to 64/32 bytes and return 32/16 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 32/16 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.
- If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the memory address is not aligned on an 8-byte boundary.

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation**LDDQU (128-bit Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[VLMAX-1:128] (Unmodified)

VLDDQU (VEX.128 encoded version)

DEST[127:0] \leftarrow SRC[127:0]
DEST[VLMAX-1:128] \leftarrow 0

VLDDQU (VEX.256 encoded version)

DEST[255:0] \leftarrow SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

LDDQU: `__m128i _mm_lddqu_si128 (__m128i * p);`
LDDQU: `__m256i _mm256_lddqu_si256 (__m256i * p);`

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4;

Note treatment of #AC varies.

LDMXCSR—Load MXCSR Register

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF,AE,/2 LDMXCSR <i>m32</i>	M	V/V	SSE	Load MXCSR register from <i>m32</i> .
VEX.LZ.OF.WIG AE /2 VLDMXCSR <i>m32</i>	M	V/V	AVX	Load MXCSR register from <i>m32</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32-bit memory location. See “MXCSR Control and Status Register” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of the MXCSR register and its contents.

The LDMXCSR instruction is typically used in conjunction with the (V)STMXCSR instruction, which stores the contents of the MXCSR register in memory.

The default MXCSR value at reset is 1F80H.

If a (V)LDMXCSR instruction clears a SIMD floating-point exception mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception will not be immediately generated. The exception will be generated only upon the execution of the next instruction that meets both conditions below:

- the instruction must operate on an XMM or YMM register operand,
- the instruction causes that particular SIMD floating-point exception to be reported.

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

If VLDMXCSR is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

$\text{MXCSR} \leftarrow \text{m32};$

C/C++ Compiler Intrinsic Equivalent

`_mm_setcsr(unsigned int i)`

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

- | | |
|-----|---|
| #GP | For an attempt to set reserved bits in MXCSR. |
| #UD | If VEX.vvvv != 1111B. |

LDS/LES/LFS/LGS/LSS—Load Far Pointer

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C5 /r	LDS r16,m16:16	RM	Invalid	Valid	Load DS:r16 with far pointer from memory.
C5 /r	LDS r32,m16:32	RM	Invalid	Valid	Load DS:r32 with far pointer from memory.
OF B2 /r	LSS r16,m16:16	RM	Valid	Valid	Load SS:r16 with far pointer from memory.
OF B2 /r	LSS r32,m16:32	RM	Valid	Valid	Load SS:r32 with far pointer from memory.
REX + OF B2 /r	LSS r64,m16:64	RM	Valid	N.E.	Load SS:r64 with far pointer from memory.
C4 /r	LES r16,m16:16	RM	Invalid	Valid	Load ES:r16 with far pointer from memory.
C4 /r	LES r32,m16:32	RM	Invalid	Valid	Load ES:r32 with far pointer from memory.
OF B4 /r	LFS r16,m16:16	RM	Valid	Valid	Load FS:r16 with far pointer from memory.
OF B4 /r	LFS r32,m16:32	RM	Valid	Valid	Load FS:r32 with far pointer from memory.
REX + OF B4 /r	LFS r64,m16:64	RM	Valid	N.E.	Load FS:r64 with far pointer from memory.
OF B5 /r	LGS r16,m16:16	RM	Valid	Valid	Load GS:r16 with far pointer from memory.
OF B5 /r	LGS r32,m16:32	RM	Valid	Valid	Load GS:r32 with far pointer from memory.
REX + OF B5 /r	LGS r64,m16:64	RM	Valid	N.E.	Load GS:r64 with far pointer from memory.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Loads a far pointer (segment selector and offset) from the second operand (source operand) into a segment register and the first operand (destination operand). The source operand specifies a 48-bit or a 32-bit pointer in memory depending on the current setting of the operand-size attribute (32 bits or 16 bits, respectively). The instruction opcode and the destination operand specify a segment register/general-purpose register pair. The 16-bit segment selector from the source operand is loaded into the segment register specified with the opcode (DS, SS, ES, FS, or GS). The 32-bit or 16-bit offset is loaded into the register specified with the destination operand.

If one of these instructions is executed in protected mode, additional information from the segment descriptor pointed to by the segment selector in the source operand is loaded in the hidden part of the selected segment register.

Also in protected mode, a NULL selector (values 0000 through 0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a NULL selector, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.W promotes operation to specify a source operand referencing an 80-bit pointer (16-bit selector, 64-bit offset) in memory. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```

64-BIT_MODE
  IF SS is loaded
    THEN
      IF SegmentSelector = NULL and ( (RPL = 3) or
          (RPL ≠ 3 and RPL ≠ CPL) )
        THEN #GP(0);
      ELSE IF descriptor is in non-canonical space
        THEN #GP(0); Fl;
      ELSE IF Segment selector index is not within descriptor table limits
        or segment selector RPL ≠ CPL
        or access rights indicate nonwritable data segment
        or DPL ≠ CPL
        THEN #GP(selector); Fl;
      ELSE IF Segment marked not present
        THEN #SS(selector); Fl;
      Fl;
      SS ← SegmentSelector(SRC);
      SS ← SegmentDescriptor([SRC]);
    ELSE IF attempt to load DS, or ES
      THEN #UD;
    ELSE IF FS, or GS is loaded with non-NULL segment selector
      THEN IF Segment selector index is not within descriptor table limits
          or access rights indicate segment neither data nor readable code segment
          or segment is data or nonconforming-code segment
          and ( RPL > DPL or CPL > DPL )
            THEN #GP(selector); Fl;
      ELSE IF Segment marked not present
        THEN #NP(selector); Fl;
      Fl;
      SegmentRegister ← SegmentSelector(SRC) ;
      SegmentRegister ← SegmentDescriptor([SRC]);
    Fl;
  ELSE IF FS, or GS is loaded with a NULL selector:
    THEN
      SegmentRegister ← NULLSelector;
      SegmentRegister(DescriptorValidBit) ← 0; Fl; (* Hidden flag;
          not accessible by software *)
    Fl;
  DEST ← Offset(SRC);

PREOTECTED MODE OR COMPATIBILITY MODE;
  IF SS is loaded
    THEN
      IF SegementSelector = NULL
        THEN #GP(0);
      ELSE IF Segment selector index is not within descriptor table limits
        or segment selector RPL ≠ CPL
        or access rights indicate nonwritable data segment
        or DPL ≠ CPL
        THEN #GP(selector); Fl;
      ELSE IF Segment marked not present
        THEN #SS(selector); Fl;
      Fl;

```

```

SS ← SegmentSelector(SRC);
SS ← SegmentDescriptor([SRC]);
ELSE IF DS, ES, FS, or GS is loaded with non-NUL segment selector
    THEN IF Segment selector index is not within descriptor table limits
        or access rights indicate segment neither data nor readable code segment
        or segment is data or nonconforming-code segment
        and (RPL > DPL or CPL > DPL)
            THEN #GP(selector); Fl;
    ELSE IF Segment marked not present
        THEN #NP(selector); Fl;
    Fl;
    SegmentRegister ← SegmentSelector(SRC) AND RPL;
    SegmentRegister ← SegmentDescriptor([SRC]);
Fl;
ELSE IF DS, ES, FS, or GS is loaded with a NUL selector:
    THEN
        SegmentRegister ← NULLSelector;
        SegmentRegister(DescriptorValidBit) ← 0; Fl; (* Hidden flag;
            not accessible by software *)
    Fl;
    DEST ← Offset(SRC);

```

Real-Address or Virtual-8086 Mode

```

SegmentRegister ← SegmentSelector(SRC); Fl;
DEST ← Offset(SRC);

```

Flags Affected

None.

Protected Mode Exceptions

#UD	If source operand is not a memory location. If the LOCK prefix is used.
#GP(0)	If a NUL selector is loaded into the SS register. 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 NUL segment selector.
#GP(selector)	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a non-writable data segment, or DPL is not equal to CPL. If the DS, ES, FS, or GS register is being loaded with a non-NUL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If DS, ES, FS, or GS register is being loaded with a non-NUL segment selector and the segment is marked 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.

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.
#UD	If source operand is not a memory location.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If source operand is not a memory location.
	If the LOCK prefix is used.
#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.
	If a NULL selector is attempted to be loaded into the SS register in compatibility mode.
	If a NULL selector is attempted to be loaded into the SS register in CPL3 and 64-bit mode.
	If a NULL selector is attempted to be loaded into the SS register in non-CPL3 and 64-bit mode where its RPL is not equal to CPL.
#GP(Selector)	If the FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the memory address of the descriptor is non-canonical, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the memory address of the descriptor is non-canonical, the segment selector RPL is not equal to CPL, the segment is a nonwritable data segment, or DPL is not equal to CPL.
#SS(0)	If a memory operand effective address is non-canonical
#SS(Selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked 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.
#UD	If source operand is not a memory location.
	If the LOCK prefix is used.

LEA—Load Effective Address

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8D /r	LEA r16,m	RM	Valid	Valid	Store effective address for <i>m</i> in register <i>r16</i> .
8D /r	LEA r32,m	RM	Valid	Valid	Store effective address for <i>m</i> in register <i>r32</i> .
REX.W + 8D /r	LEA r64,m	RM	Valid	N.E.	Store effective address for <i>m</i> in register <i>r64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processor's addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

Table 3-63. Non-64-bit Mode LEA Operation with Address and Operand Size Attributes

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

In 64-bit mode, the instruction's destination operand is governed by operand size attribute, the default operand size is 32 bits. Address calculation is governed by address size attribute, the default address size is 64-bits. In 64-bit mode, address size of 16 bits is not encodable. See Table 3-64.

Table 3-64. 64-bit Mode LEA Operation with Address and Operand Size Attributes

Operand Size	Address Size	Action Performed
16	32	32-bit effective address is calculated (using 67H prefix). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).
16	64	64-bit effective address is calculated (default address size). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).
32	32	32-bit effective address is calculated (using 67H prefix) and stored in the requested 32-bit register destination.
32	64	64-bit effective address is calculated (default address size) and the lower 32 bits of the address are stored in the requested 32-bit register destination.
64	32	32-bit effective address is calculated (using 67H prefix), zero-extended to 64-bits, and stored in the requested 64-bit register destination (using REX.W).
64	64	64-bit effective address is calculated (default address size) and all 64-bits of the address are stored in the requested 64-bit register destination (using REX.W).

Operation

```

IF OperandSize = 16 and AddressSize = 16
  THEN
    DEST ← EffectiveAddress(SRC); (* 16-bit address *)
  ELSE IF OperandSize = 16 and AddressSize = 32
    THEN
      temp ← EffectiveAddress(SRC); (* 32-bit address *)
      DEST ← temp[0:15]; (* 16-bit address *)
    FI;
  ELSE IF OperandSize = 32 and AddressSize = 16
    THEN
      temp ← EffectiveAddress(SRC); (* 16-bit address *)
      DEST ← ZeroExtend(temp); (* 32-bit address *)
    FI;
  ELSE IF OperandSize = 32 and AddressSize = 32
    THEN
      DEST ← EffectiveAddress(SRC); (* 32-bit address *)
    FI;
  ELSE IF OperandSize = 16 and AddressSize = 64
    THEN
      temp ← EffectiveAddress(SRC); (* 64-bit address *)
      DEST ← temp[0:15]; (* 16-bit address *)
    FI;
  ELSE IF OperandSize = 32 and AddressSize = 64
    THEN
      temp ← EffectiveAddress(SRC); (* 64-bit address *)
      DEST ← temp[0:31]; (* 16-bit address *)
    FI;
  ELSE IF OperandSize = 64 and AddressSize = 64
    THEN
      DEST ← EffectiveAddress(SRC); (* 64-bit address *)
    FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If source operand is not a memory location. If the LOCK prefix is used.
-----	--

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.

LEAVE—High Level Procedure Exit

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C9	LEAVE	NP	Valid	Valid	Set SP to BP, then pop BP.
C9	LEAVE	NP	N.E.	Valid	Set ESP to EBP, then pop EBP.
C9	LEAVE	NP	Valid	N.E.	Set RSP to RBP, then pop RBP.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Releases the stack frame set up by an earlier ENTER instruction. The LEAVE instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), which releases the stack space allocated to the stack frame. The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure's stack frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure.

See "Procedure Calls for Block-Structured Languages" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for detailed information on the use of the ENTER and LEAVE instructions.

In 64-bit mode, the instruction's default operation size is 64 bits; 32-bit operation cannot be encoded. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```

IF StackAddressSize = 32
  THEN
    ESP ← EBP;
  ELSE IF StackAddressSize = 64
    THEN RSP ← RBP; Fl;
  ELSE IF StackAddressSize = 16
    THEN SP ← BP; Fl;
  Fl;
IF OperandSize = 32
  THEN EBP ← Pop();
  ELSE IF OperandSize = 64
    THEN RBP ← Pop(); Fl;
  ELSE IF OperandSize = 16
    THEN BP ← Pop(); Fl;
  Fl;

```

Flags Affected

None.

Protected Mode Exceptions

- #SS(0) If the EBP register points to a location that is not within the limits of the current stack segment.
- #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.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #GP If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #GP(0) If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #SS(0) If the stack address is in a non-canonical form.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.

LFENCE—Load Fence

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF AE /5	LFENCE	NP	Valid	Valid	Serializes load operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. In particular, an instruction that loads from memory and that precedes an LFENCE receives data from memory prior to completion of the LFENCE. (An LFENCE that follows an instruction that stores to memory might complete **before** the data being stored have become globally visible.) Instructions following an LFENCE may be fetched from memory before the LFENCE, but they will not execute until the LFENCE completes.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue and speculative reads. 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 LFENCE instruction provides a performance-efficient way of ensuring load ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the LFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an LFENCE instruction.

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

Operation

```
Wait_On_Following_Instructions_Until(preceding_instructions_complete);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_lfence(void)
```

Exceptions (All Modes of Operation)

#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /2	LGDT <i>m</i> 16&32	M	N.E.	Valid	Load <i>m</i> into GDTR.
0F 01 /3	LIDT <i>m</i> 16&32	M	N.E.	Valid	Load <i>m</i> into IDTR.
0F 01 /2	LGDT <i>m</i> 16&64	M	Valid	N.E.	Load <i>m</i> into GDTR.
0F 01 /3	LIDT <i>m</i> 16&64	M	Valid	N.E.	Load <i>m</i> into IDTR.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

In 64-bit mode, the instruction's operand size is fixed at 8+2 bytes (an 8-byte base and a 2-byte limit). See the summary chart at the beginning of this section for encoding data and limits.

See “SGDT—Store Global Descriptor Table Register” in Chapter 4, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, for information on storing the contents of the GDTR and IDTR.

Operation

```

IF Instruction is LIDT
  THEN
    IF OperandSize = 16
      THEN
        IDTR(Limit) ← SRC[0:15];
        IDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
      ELSE IF 32-bit Operand Size
        THEN
          IDTR(Limit) ← SRC[0:15];
          IDTR(Base) ← SRC[16:47];
        FI;
      ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
        THEN
          IDTR(Limit) ← SRC[0:15];
          IDTR(Base) ← SRC[16:79];
        FI;
      FI;
    ELSE (* Instruction is LGDT *)
      IF OperandSize = 16

```

```

THEN
    GDTR(Limit) ← SRC[0:15];
    GDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
ELSE IF 32-bit Operand Size
    THEN
        GDTR(Limit) ← SRC[0:15];
        GDTR(Base) ← SRC[16:47];
    FI;
ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
    THEN
        GDTR(Limit) ← SRC[0:15];
        GDTR(Base) ← SRC[16:79];
    FI;
FI;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

- #UD If source operand is not a memory location.
 If the LOCK prefix is used.
- #GP(0) If the current privilege level is not 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.

Real-Address Mode Exceptions

- #UD If source operand is not a memory location.
 If the LOCK prefix is used.
- #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 source operand is not a memory location.
 If the LOCK prefix is used.
- #GP(0) The LGDT and LIDT instructions are not recognized in virtual-8086 mode.
- #GP If the current privilege level is not 0.

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 current privilege level is not 0.
	If the memory address is in a non-canonical form.
#UD	If source operand is not a memory location.
	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.

LLDT—Load Local Descriptor Table Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 00 /2	LLDT r/m16	M	Valid	Valid	Load segment selector <i>r/m16</i> into LDTR.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Loads the source operand into the segment selector field of the local descriptor table register (LDTR). The source operand (a general-purpose register or a memory location) contains a segment selector that points to a local descriptor table (LDT). After the segment selector is loaded in the LDTR, the processor uses the segment selector to locate the segment descriptor for the LDT in the global descriptor table (GDT). It then loads the segment limit and base address for the LDT from the segment descriptor into the LDTR. The segment registers DS, ES, SS, FS, GS, and CS are not affected by this instruction, nor is the LDTR field in the task state segment (TSS) for the current task.

If bits 2-15 of the source operand are 0, LDTR is marked invalid and the LLDT instruction completes silently. However, all subsequent references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause a general protection exception (#GP).

The operand-size attribute has no effect on this instruction.

The LLDT instruction is provided for use in operating-system software; it should not be used in application programs. This instruction can only be executed in protected mode or 64-bit mode.

In 64-bit mode, the operand size is fixed at 16 bits.

Operation

IF SRC(Offset) > descriptor table limit
 THEN #GP(segment selector); Fl;

IF segment selector is valid

 Read segment descriptor;

 IF SegmentDescriptor(Type) ≠ LDT
 THEN #GP(segment selector); Fl;

 IF segment descriptor is not present
 THEN #NP(segment selector); Fl;

 LDTR(SegmentSelector) ← SRC;
 LDTR(SegmentDescriptor) ← GDTSegmentDescriptor;
 ELSE LDTR ← INVALID
 Fl;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 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.
#GP(selector)	If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table.
	Segment selector is beyond GDT limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NP(selector)	If the LDT descriptor is not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The LLDT instruction is not recognized in real-address mode.
-----	--

Virtual-8086 Mode Exceptions

#UD	The LLDT 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 current privilege level is not 0.
	If the memory address is in a non-canonical form.
#GP(selector)	If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table.
	Segment selector is beyond GDT limit.
#NP(selector)	If the LDT descriptor is not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

LMSW—Load Machine Status Word

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 01 /6	LMSW <i>r/m16</i>	M	Valid	Valid	Loads <i>r/m16</i> in machine status word of CR0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. While in protected mode, the LMSW instruction cannot be used to clear the PE flag and force a switch back to real-address mode.

The LMSW instruction is provided for use in operating-system software; it should not be used in application programs. In protected or virtual-8086 mode, it can only be executed at CPL 0.

This instruction 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 whole CR0 register. The MOV CR0 instruction can be used to set and clear the PE flag in CR0, allowing a procedure or program to switch between protected and real-address modes.

This instruction is a serializing instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Note that the operand size is fixed at 16 bits.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

$\text{CR0}[0:3] \leftarrow \text{SRC}[0:3];$

Flags Affected

None.

Protected Mode Exceptions

- | | |
|-----------------|---|
| #GP(0) | If the current privilege level is not 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. |
| #UD | If the LOCK prefix is used. |

Real-Address Mode Exceptions

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

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.
- #UD If the LOCK prefix is used.

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 current privilege level is not 0.
- If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #UD If the LOCK prefix is used.

LOCK—Assert LOCK# Signal Prefix

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F0	LOCK	NP	Valid	Valid	Asserts LOCK# signal for duration of the accompanying instruction.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.

Note that, in later Intel 64 and IA-32 processors (including the Pentium 4, Intel Xeon, and P6 family processors), locking may occur without the LOCK# signal being asserted. See the "IA-32 Architecture Compatibility" section below.

The LOCK prefix can be prepended only to the following instructions and only to those forms of the instructions where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. If the LOCK prefix is used with one of these instructions and the source operand is a memory operand, an undefined opcode exception (#UD) may be generated. An undefined opcode exception will also be generated if the LOCK prefix is used with any instruction not in the above list. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

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

IA-32 Architecture Compatibility

Beginning with the P6 family processors, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor's cache is locked. Here, the processor's cache coherency mechanism ensures that the operation is carried out atomically with regards to memory. See "Effects of a Locked Operation on Internal Processor Caches" in Chapter 8 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, the for more information on locking of caches.

Operation

AssertLOCK#(DurationOfAccompanyingInstruction);

Flags Affected

None.

Protected Mode Exceptions

#UD

If the LOCK prefix is used with an instruction not listed: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG.

Other exceptions can be generated by the instruction when the LOCK prefix is applied.

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.

LODS/LODSB/LODSW/LODSD/LODSQ—Load String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
AC	LODS <i>m8</i>	NP	Valid	Valid	For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL.
AD	LODS <i>m16</i>	NP	Valid	Valid	For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX.
AD	LODS <i>m32</i>	NP	Valid	Valid	For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX.
REX.W + AD	LODS <i>m64</i>	NP	Valid	N.E.	Load qword at address (R)SI into RAX.
AC	LODSB	NP	Valid	Valid	For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL.
AD	LODSW	NP	Valid	Valid	For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX.
AD	LODSD	NP	Valid	Valid	For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX.
REX.W + AD	LODSQ	NP	Valid	N.E.	Load qword at address (R)SI into RAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Loads a byte, word, or doubleword from the source operand into the AL, AX, or EAX register, respectively. The source operand is a memory location, the address of which is read from the DS:ESI or the DS:SI 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-operands” form and the “no-operands” form. The explicit-operands form (specified with the LODS mnemonic) allows the source operand to be specified explicitly. Here, the source operand should be a symbol that indicates the size and location of the source value. The destination operand is then automatically selected to match the size of the source operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the DS:(E)SI registers, which must be loaded correctly before the load string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the LODS instructions. Here also DS:(E)SI is assumed to be the source operand and the AL, AX, or EAX register is assumed to be the destination operand. The size of the source and destination operands is selected with the mnemonic: LODSB (byte loaded into register AL), LODSW (word loaded into AX), or LODSD (doubleword loaded into EAX).

After the byte, word, or doubleword is transferred from the memory location into the AL, AX, or EAX register, the (E)SI 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 ESI register is decremented.) The (E)SI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. LODS/LODSQ load the quadword at address (R)SI into RAX. The (R)SI register is then incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register.

The LODS, LODSB, LODSW, and LODSD 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 further processing of the data moved into the register is usually necessary before the next transfer can be made. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a description of the REP prefix.

Operation

```

IF AL ← SRC; (* Byte load *)
    THEN AL ← SRC; (* Byte load *)
        IF DF = 0
            THEN (E)SI ← (E)SI + 1;
            ELSE (E)SI ← (E)SI - 1;
        FI;
ELSE IF AX ← SRC; (* Word load *)
    THEN IF DF = 0
        THEN (E)SI ← (E)SI + 2;
        ELSE (E)SI ← (E)SI - 2;
    FI;
ELSE IF EAX ← SRC; (* Doubleword load *)
    THEN IF DF = 0
        THEN (E)SI ← (E)SI + 4;
        ELSE (E)SI ← (E)SI - 4;
    FI;
FI;
ELSE IF RAX ← SRC; (* Quadword load *)
    THEN IF DF = 0
        THEN (R)SI ← (R)SI + 8;
        ELSE (R)SI ← (R)SI - 8;
    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 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.
#UD	If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

LOOP/LOOPcc—Loop According to ECX Counter

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E2 cb	LOOP rel8	D	Valid	Valid	Decrement count; jump short if count ≠ 0.
E1 cb	LOOPE rel8	D	Valid	Valid	Decrement count; jump short if count ≠ 0 and ZF = 1.
E0 cb	LOOPNE rel8	D	Valid	Valid	Decrement count; jump short if count ≠ 0 and ZF = 0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA

Description

Performs a loop operation using the RCX, ECX or CX register as a counter (depending on whether address size is 64 bits, 32 bits, or 16 bits). Note that the LOOP instruction ignores REX.W; but 64-bit address size can be over-ridden using a 67H prefix.

Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the IP/EIP/RIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of -128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOPcc) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (cc) is associated with each instruction to indicate the condition being tested for. Here, the LOOPcc instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

Operation

```

IF (AddressSize = 32)
    THEN Count is ECX;
ELSE IF (AddressSize = 64)
    Count is RCX;
ELSE Count is CX;
FI;

Count ← Count - 1;

IF Instruction is not LOOP
    THEN
        IF (Instruction ← LOOPE) or (Instruction ← LOOPZ)
            THEN IF (ZF = 1) and (Count ≠ 0)
                THEN BranchCond ← 1;
                ELSE BranchCond ← 0;
            FI;
            ELSE (Instruction = LOOPNE) or (Instruction = LOOPNZ)
                IF (ZF = 0) and (Count ≠ 0)
                    THEN BranchCond ← 1;
                    ELSE BranchCond ← 0;
                FI;
            ENDIF;
        ENDIF;
    ENDIF;
ENDIF;

```

```

        Fl;
        Fl;
ELSE (* Instruction = LOOP *)
    IF (Count ≠ 0)
        THEN BranchCond ← 1;
        ELSE BranchCond ← 0;
    Fl;
Fl;

IF BranchCond = 1
THEN
    IF OperandSize = 32
        THEN EIP ← EIP + SignExtend(DEST);
    ELSE IF OperandSize = 64
        THEN RIP ← RIP + SignExtend(DEST);
    Fl;
    ELSE IF OperandSize = 16
        THEN EIP ← EIP AND 0000FFFFH;
    Fl;
Fl;
IF OperandSize = (32 or 64)
    THEN IF (R/E)IP < CS.Base or (R/E)IP > CS.Limit
        #GP; Fl;
    Fl;
Fl;
ELSE
    Terminate loop and continue program execution at (R/E)IP;
Fl;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the offset being jumped to is beyond the limits of the CS segment.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the offset being jumped to is in a non-canonical form.
#UD	If the LOCK prefix is used.

LSL—Load Segment Limit

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 03 /r	LSL r16, r16/m16	RM	Valid	Valid	Load: $r16 \leftarrow$ segment limit, selector $r16/m16$.
OF 03 /r	LSL r32, r32/m16*	RM	Valid	Valid	Load: $r32 \leftarrow$ segment limit, selector $r32/m16$.
REX.W + OF 03 /r	LSL r64, r32/m16*	RM	Valid	Valid	Load: $r64 \leftarrow$ segment limit, selector $r32/m16$

NOTES:

- * For all loads (regardless of destination sizing), only bits 16-0 are used. Other bits are ignored.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Loads the unscrambled segment limit from the segment descriptor specified with the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

The segment limit is a 20-bit value contained in bytes 0 and 1 and in the first 4 bits of byte 6 of the segment descriptor. If the descriptor has a byte granular segment limit (the granularity flag is set to 0), the destination operand is loaded with a byte granular value (byte limit). If the descriptor has a page granular segment limit (the granularity flag is set to 1), the LSL instruction will translate the page granular limit (page limit) into a byte limit before loading it into the destination operand. The translation is performed by shifting the 20-bit “raw” limit left 12 bits and filling the low-order 12 bits with 1s.

When the operand size is 32 bits, the 32-bit byte limit is stored in the destination operand. When the operand size is 16 bits, a valid 32-bit limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand.

This instruction performs the following checks before it loads the segment limit into the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LSL instruction. The valid special segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, the instruction checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no value is loaded in the destination operand.

Table 3-65. Segment and Gate Descriptor Types

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Upper 8 byte of a 16-Byte descriptor	Yes
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT	Yes
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	No	Reserved	No
5	16-bit/32-bit task gate	No	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	64-bit TSS	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS	Yes
C	32-bit call gate	No	64-bit call gate	No
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

Operation

IF SRC(Offset) > descriptor table limit

THEN ZF ← 0; Fl;

Read segment descriptor;

IF SegmentDescriptor(Type) ≠ conforming code segment

and (CPL > DPL) OR (RPL > DPL)

or Segment type is not valid for instruction

THEN

ZF ← 0;

ELSE

temp ← SegmentLimit([SRC]);

IF (G ← 1)

THEN temp ← ShiftLeft(12, temp) OR 00000FFFH;

ELSE IF OperandSize = 32

THEN DEST ← temp; Fl;

ELSE IF OperandSize = 64 (* REX.W used *)

THEN DEST (* Zero-extended *) ← temp; Fl;

ELSE (* OperandSize = 16 *)

DEST ← temp AND FFFFH;

Fl;

Fl;

Flags Affected

The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is set to 0.

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 the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The LSL instruction cannot be executed in real-address mode.
-----	--

Virtual-8086 Mode Exceptions

#UD	The LSL instruction cannot be executed in virtual-8086 mode.
-----	--

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

LTR—Load Task Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 00 /3	LTR <i>r/m16</i>	M	Valid	Valid	Load <i>r/m16</i> into task register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses the segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

In 64-bit mode, the operand size is still fixed at 16 bits. The instruction references a 16-byte descriptor to load the 64-bit base.

Operation

IF SRC is a NULL selector
THEN #GP(0);

IF SRC(Offset) > descriptor table limit OR IF SRC(type) ≠ global
THEN #GP(segment selector); Fl;

Read segment descriptor;

IF segment descriptor is not for an available TSS
THEN #GP(segment selector); Fl;

IF segment descriptor is not present
THEN #NP(segment selector); Fl;

TSSsegmentDescriptor(busy) ← 1;
(* Locked read-modify-write operation on the entire descriptor when setting busy flag *)

TaskRegister(SegmentSelector) ← SRC;
TaskRegister(SegmentDescriptor) ← TSSSegmentDescriptor;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the source operand contains a NULL segment selector. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector)	If the source selector points to a segment that is not a TSS or to one for a task that is already busy. If the selector points to LDT or is beyond the GDT limit.
#NP(selector)	If the TSS is marked not present.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The LTR instruction is not recognized in real-address mode.
-----	---

Virtual-8086 Mode Exceptions

#UD	The LTR 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 current privilege level is not 0. If the memory address is in a non-canonical form. If the source operand contains a NULL segment selector.
#GP(selector)	If the source selector points to a segment that is not a TSS or to one for a task that is already busy. If the selector points to LDT or is beyond the GDT limit. If the descriptor type of the upper 8-byte of the 16-byte descriptor is non-zero.
#NP(selector)	If the TSS is marked not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

LZCNT—Count the Number of Leading Zero Bits

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
F3 OF BD /r LZCNT r16, r/m16	RM	V/V	LZCNT	Count the number of leading zero bits in r/m16, return result in r16.
F3 OF BD /r LZCNT r32, r/m32	RM	V/V	LZCNT	Count the number of leading zero bits in r/m32, return result in r32.
REX.W + F3 OF BD /r LZCNT r64, r/m64	RM	V/N.E.	LZCNT	Count the number of leading zero bits in r/m64, return result in r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Counts the number of leading most significant zero bits in a source operand (second operand) returning the result into a destination (first operand).

LZCNT differs from BSR. For example, LZCNT will produce the operand size when the input operand is zero. It should be noted that on processors that do not support LZCNT, the instruction byte encoding is executed as BSR. In 64-bit mode 64-bit operand size requires REX.W=1.

Operation

```

temp ← OperandSize - 1
DEST ← 0
WHILE (temp >= 0) AND (Bit(SRC, temp) = 0)
DO
    temp ← temp - 1
    DEST ← DEST+ 1
OD
IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI
IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI

```

Flags Affected

ZF flag is set to 1 in case of zero output (most significant bit of the source is set), and to 0 otherwise, CF flag is set to 1 if input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

LZCNT: `unsigned __int32 _lzcnt_u32(unsigned __int32 src);`
 LZCNT: `unsigned __int64 _lzcnt_u64(unsigned __int64 src);`

Protected Mode Exceptions

- #GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
- #SS(0) For an illegal address in the SS segment.
- #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.

Real-Address Mode Exceptions

- #GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
- #SS(0) For an illegal address in the SS segment.

Virtual 8086 Mode Exceptions

- #GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
- #SS(0) For an illegal address in the SS segment.
- #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.

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(0) If a memory address referencing the SS segment 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.

MASKMOVDQU—Store Selected Bytes of Double Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F F7 /r MASKMOVDQU <i>xmm1, xmm2</i>		RM	V/V	SSE2 Selectively write bytes from <i>xmm1</i> to memory location using the byte mask in <i>xmm2</i> . The default memory location is specified by DS:DI/EDI/RDI.
VEX.128.66.0F.WIG F7 /r VMASKMOVDQU <i>xmm1, xmm2</i>		RM	V/V	AVX Selectively write bytes from <i>xmm1</i> to memory location using the byte mask in <i>xmm2</i> . The default memory location is specified by DS:DI/EDI/RDI.

Instruction Operand Encoding¹

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Stores selected bytes from the source operand (first operand) into an 128-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are XMM registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVDQU instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVDQU instructions if multiple processors might use different memory types to read/write the destination memory locations.

Behavior with a mask of all 0s is as follows:

- No data will be written to memory.
- Signaling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVDQU instruction can be used to improve performance of algorithms that need to merge data on a byte-by-byte basis. MASKMOVDQU should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If VMASKMOVDQU is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

1. ModRM.MOD = 011B required

Operation

```

IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); Fl;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); Fl;
    (* Repeat operation for 3rd through 14th bytes in source operand *)
IF (MASK[127] = 1)
    THEN DEST[DI/EDI +15] ← SRC[127:120] ELSE (* Memory location unchanged *); Fl;

```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_maskmoveu_si128(__m128i d, __m128i n, char * p)
```

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L= 1
	If VEX.vvvv != 1111B.

MASKMOVQ—Store Selected Bytes of Quadword

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF F7 /r MASKMOVQ <i>mm1, mm2</i>		RM	Valid	Selectively write bytes from <i>mm1</i> to memory location using the byte mask in <i>mm2</i> . The default memory location is specified by DS:DI/EDI/RDI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Stores selected bytes from the source operand (first operand) into a 64-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are MMX technology registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVQ instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction causes a transition from x87 FPU to MMX technology state (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]).

The behavior of the MASKMOVQ instruction with a mask of all 0s is as follows:

- No data will be written to memory.
- Transition from x87 FPU to MMX technology state will occur.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- Signaling of breakpoints (code or data) is not guaranteed (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVQ instruction can be used to improve performance for algorithms that need to merge data on a byte-by-byte basis. It should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, the memory address is specified by DS:RDI.

Operation

```

IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); Fl;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); Fl;
    (* Repeat operation for 3rd through 6th bytes in source operand *)
IF (MASK[63] = 1)
    THEN DEST[DI/EDI +15] ← SRC[63:56] ELSE (* Memory location unchanged *); Fl;

```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_maskmove_si64(__m64d, __m64n, char * p)
```

Other Exceptions

See Table 22-8, "Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

MAXPD—Return Maximum Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5F /r MAXPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Return the maximum double-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 5F /r VMAXPD <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the maximum double-precision floating-point values between <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 5F /r VMAXPD <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the maximum packed double-precision floating-point values between <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation**MAX(SRC1, SRC2)**

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}
```

MAXPD (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow \text{MAX}(\text{DEST}[63:0], \text{SRC}[63:0])$
 $\text{DEST}[127:64] \leftarrow \text{MAX}(\text{DEST}[127:64], \text{SRC}[127:64])$
 $\text{DEST}[VLMAX-1:128] \text{ (Unmodified)}$

VMAXPD (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0])$
 $\text{DEST}[127:64] \leftarrow \text{MAX}(\text{SRC1}[127:64], \text{SRC2}[127:64])$
 $\text{DEST}[VLMAX-1:128] \leftarrow 0$

VMAXPD (VEX.256 encoded version)

$\text{DEST}[63:0] \leftarrow \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0])$
 $\text{DEST}[127:64] \leftarrow \text{MAX}(\text{SRC1}[127:64], \text{SRC2}[127:64])$
 $\text{DEST}[191:128] \leftarrow \text{MAX}(\text{SRC1}[191:128], \text{SRC2}[191:128])$
 $\text{DEST}[255:192] \leftarrow \text{MAX}(\text{SRC1}[255:192], \text{SRC2}[255:192])$

Intel C/C++ Compiler Intrinsic Equivalent

MAXPD: `_m128d _mm_max_pd(_m128d a, _m128d b);`
 VMAXPD: `_m256d _mm256_max_pd (_m256d a, _m256d b);`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 2.

MAXPS—Return Maximum Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5F /r MAXPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Return the maximum single-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.0F.WIG 5F /r VMAXPS <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Return the maximum single-precision floating-point values between <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG 5F /r VMAXPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Return the maximum single double-precision floating-point values between <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation**MAX(SRC1, SRC2)**

```
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
}
FI;
```

MAXPS (128-bit Legacy SSE version)

$DEST[31:0] \leftarrow MAX(DEST[31:0], SRC[31:0])$
 $DEST[63:32] \leftarrow MAX(DEST[63:32], SRC[63:32])$
 $DEST[95:64] \leftarrow MAX(DEST[95:64], SRC[95:64])$
 $DEST[127:96] \leftarrow MAX(DEST[127:96], SRC[127:96])$
 $DEST[VLMAX-1:128] \text{ (Unmodified)}$

VMAXPS (VEX.128 encoded version)

$DEST[31:0] \leftarrow MAX(SRC1[31:0], SRC2[31:0])$
 $DEST[63:32] \leftarrow MAX(SRC1[63:32], SRC2[63:32])$
 $DEST[95:64] \leftarrow MAX(SRC1[95:64], SRC2[95:64])$
 $DEST[127:96] \leftarrow MAX(SRC1[127:96], SRC2[127:96])$
 $DEST[VLMAX-1:128] \leftarrow 0$

VMAXPS (VEX.256 encoded version)

$DEST[31:0] \leftarrow MAX(SRC1[31:0], SRC2[31:0])$
 $DEST[63:32] \leftarrow MAX(SRC1[63:32], SRC2[63:32])$
 $DEST[95:64] \leftarrow MAX(SRC1[95:64], SRC2[95:64])$
 $DEST[127:96] \leftarrow MAX(SRC1[127:96], SRC2[127:96])$
 $DEST[159:128] \leftarrow MAX(SRC1[159:128], SRC2[159:128])$
 $DEST[191:160] \leftarrow MAX(SRC1[191:160], SRC2[191:160])$
 $DEST[223:192] \leftarrow MAX(SRC1[223:192], SRC2[223:192])$
 $DEST[255:224] \leftarrow MAX(SRC1[255:224], SRC2[255:224])$

Intel C/C++ Compiler Intrinsic Equivalent

MAXPS: `__m128 _mm_max_ps (__m128 a, __m128 b);`
 VMAXPS: `__m256 _mm256_max_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 2.

MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 5F /r MAXSD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Return the maximum scalar double-precision floating-point value between <i>xmm2/mem64</i> and <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 5F /r VMAXSD <i>xmm1, xmm2, xmm3/m64</i>	RVM	V/V	AVX	Return the maximum scalar double-precision floating-point value between <i>xmm3/mem64</i> and <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the first source operand and second the source operand, and returns the maximum value to the low quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed. The high quadword of the destination operand is copied from the same bits of first source operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN of either source operand be returned, the action of MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

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

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

MAX(SRC1, SRC2)

```
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; F1;
  ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; F1;
  ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  F1;
}
```

MAXSD (128-bit Legacy SSE version)

DEST[63:0] $\leftarrow \text{MAX}(\text{DEST}[63:0], \text{SRC}[63:0])$
DEST[VLMAX-1:64] (Unmodified)

VMAXSD (VEX.128 encoded version)

DEST[63:0] $\leftarrow \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0])$
DEST[127:64] $\leftarrow \text{SRC1}[127:64]$
DEST[VLMAX-1:128] $\leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

MAXSD: `__m128d _mm_max_sd(__m128d a, __m128d b)`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 3.

MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 5F /r MAXSS <i>xmm1, xmm2/m32</i>		RM V/V	SSE	Return the maximum scalar single-precision floating-point value between <i>xmm2/mem32</i> and <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 5F /r VMAXSS <i>xmm1, xmm2, xmm3/m32</i>	RVM	V/V	AVX	Return the maximum scalar single-precision floating-point value between <i>xmm3/mem32</i> and <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN from either source operand be returned, the action of MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

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

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; Fl;
    ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; Fl;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    Fl;
}
```

MAXSS (128-bit Legacy SSE version)

DEST[31:0] ← MAX(DEST[31:0], SRC[31:0])
DEST[VLMAX-1:32] (Unmodified)

VMAXSS (VEX.128 encoded version)

DEST[31:0] $\leftarrow \text{MAX}(\text{SRC1}[31:0], \text{SRC2}[31:0])$
DEST[127:32] $\leftarrow \text{SRC1}[127:32]$
DEST[VLMAX-1:128] $\leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

`__m128d _mm_max_ss(__m128d a, __m128d b)`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 3.

MFENCE—Memory Fence

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF AE /6	MFENCE	NP	Valid	Valid	Serializes load and store operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction.¹ The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any LFENCE and SFENCE instructions, and any serializing instructions (such as the CPUID instruction). MFENCE does not serialize the instruction stream.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, 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 MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the MFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an MFENCE instruction.

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

Operation

```
Wait_On_Following_Loads_And_Stores_Until(preceding_loads_and_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_mfence(void)
```

Exceptions (All Modes of Operation)

#UD If CPUID.01H:EDX.SSE2[bit 26] = 0.
 If the LOCK prefix is used.

1. A load instruction is considered to become globally visible when the value to be loaded into its destination register is determined.

MINPD—Return Minimum Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 5D /r MINPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Return the minimum double-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 5D /r VMINPD <i>xmm1,xmm2, xmm3/m128</i>	RVM	V/V	AVX	Return the minimum double-precision floating-point values between <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 5D /r VMINPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Return the minimum packed double-precision floating-point values between <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; F1;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; F1;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    F1;
}
```

MINPD (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])$
 $\text{DEST}[127:64] \leftarrow \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])$
 $\text{DEST}[\text{VLMAX}-1:128] \text{ (Unmodified)}$

VMINPD (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])$
 $\text{DEST}[127:64] \leftarrow \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])$
 $\text{DEST}[\text{VLMAX}-1:128] \leftarrow 0$

VMINPD (VEX.256 encoded version)

$\text{DEST}[63:0] \leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])$
 $\text{DEST}[127:64] \leftarrow \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])$
 $\text{DEST}[191:128] \leftarrow \text{MIN}(\text{SRC1}[191:128], \text{SRC2}[191:128])$
 $\text{DEST}[255:192] \leftarrow \text{MIN}(\text{SRC1}[255:192], \text{SRC2}[255:192])$

Intel C/C++ Compiler Intrinsic Equivalent

MINPD: `__m128d _mm_min_pd(__m128d a, __m128d b);`
 VMINPD: `__m256d _mm256_min_pd (__m256d a, __m256d b);`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 2.

MINPS—Return Minimum Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 5D /r MINPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Return the minimum single-precision floating-point values between <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.0F.WIG 5D /r VMINPS <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Return the minimum single-precision floating-point values between <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.0F.WIG 5D /r VMINPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Return the minimum single double-precision floating-point values between <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

```

MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
}

```

MINPS (128-bit Legacy SSE version)

```
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])
DEST[VLMAX-1:128] (Unmodified)
```

VMINPS (VEX.128 encoded version)

```
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])
DEST[VLMAX-1:128] ← 0
```

VMINPS (VEX.256 encoded version)

```
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])
DEST[159:128] ← MIN(SRC1[159:128], SRC2[159:128])
DEST[191:160] ← MIN(SRC1[191:160], SRC2[191:160])
DEST[223:192] ← MIN(SRC1[223:192], SRC2[223:192])
DEST[255:224] ← MIN(SRC1[255:224], SRC2[255:224])
```

Intel C/C++ Compiler Intrinsic Equivalent

MINPS: `__m128d _mm_min_ps(__m128d a, __m128d b);`
 VMINPS: `__m256 _mm256_min_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 2.

MINSD—Return Minimum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 5D /r MINSD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Return the minimum scalar double-precision floating-point value between <i>xmm2/mem64</i> and <i>xmm1</i> .
VEX.NDS.LIG.F2.0F.WIG 5D /r VMINSD <i>xmm1, xmm2, xmm3/m64</i>	RVM	V/V	AVX	Return the minimum scalar double precision floating-point value between <i>xmm3/mem64</i> and <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed. The high quadword of the destination operand is copied from the same bits in the first source operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second source) be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

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

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

```

MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; Fl;
    ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; Fl;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    Fl;
}

```

MINSD (128-bit Legacy SSE version)

```

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[VLMAX-1:64] (Unmodified)

```

MINSD (VEX.128 encoded version)

DEST[63:0] $\leftarrow \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])$
DEST[127:64] $\leftarrow \text{SRC1}[127:64]$
DEST[VLMAX-1:128] $\leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

MINSD: $__m128d __mm_min_sd(__m128d a, __m128d b)$

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 3.

MINSS—Return Minimum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 5D /r MINSS <i>xmm1, xmm2/m32</i>		RM V/V	SSE	Return the minimum scalar single-precision floating-point value between <i>xmm2/mem32</i> and <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 5D /r VMINSS <i>xmm1,xmm2, xmm3/m32</i>	RVM	V/V	AVX	Return the minimum scalar single precision floating-point value between <i>xmm3/mem32</i> and <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand and returns the minimum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN in either source operand be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

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

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

```

MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}

```

MINSS (128-bit Legacy SSE version)

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[VLMAX-1:32] (Unmodified)

VMINSS (VEX.128 encoded version)

DEST[31:0] $\leftarrow \text{MIN}(\text{SRC1}[31:0], \text{SRC2}[31:0])$
DEST[127:32] $\leftarrow \text{SRC1}[127:32]$
DEST[VLMAX-1:128] $\leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

MINSS: $__m128d\ _\text{mm_min_ss}(__m128d\ a, __m128d\ b)$

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 3.

MONITOR—Set Up Monitor Address

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 01 C8	MONITOR	NP	Valid	Valid	Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The address is DS:EAX (DS:RAX in 64-bit mode).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

The MONITOR instruction arms address monitoring hardware using an address specified in EAX (the address range that the monitoring hardware checks for store operations can be determined by using CPUID). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by MWAIT.

The content of EAX is an effective address (in 64-bit mode, RAX is used). By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used.

ECX and EDX are also used. They communicate other information to MONITOR. ECX specifies optional extensions. EDX specifies optional hints; it does not change the architectural behavior of the instruction. For the Pentium 4 processor (family 15, model 3), no extensions or hints are defined. Undefined hints in EDX are ignored by the processor; undefined extensions in ECX raises a general protection fault.

The address range must use memory of the write-back type. Only write-back memory will correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 8, “Multiple-Processor Management” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

The MONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction is subject to the permission checking and faults associated with a byte load. Like a load, MONITOR sets the A-bit but not the D-bit in page tables.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MONITOR may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MONITOR clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

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

Operation

MONITOR sets up an address range for the monitor hardware using the content of EAX (RAX in 64-bit mode) as an effective address and puts the monitor hardware in armed state. Always use memory of the write-back caching type. A store to the specified address range will trigger the monitor hardware. The content of ECX and EDX are used to communicate other information to the monitor hardware.

Intel C/C++ Compiler Intrinsic Equivalent

MONITOR: void _mm_monitor(void const *p, unsigned extensions,unsigned hints)

Numeric Exceptions

None

Protected Mode Exceptions

#GP(0)	If the value in EAX 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. If ECX ≠ 0.
#SS(0)	If the value in EAX is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0. If current privilege level is not 0.

Real Address Mode Exceptions

#GP	If the CS, DS, ES, FS, or GS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH. If ECX ≠ 0.
#SS	If the SS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

Virtual 8086 Mode Exceptions

#UD	The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).
-----	--

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the linear address of the operand in the CS, DS, ES, FS, or GS segment is in a non-canonical form. If RCX ≠ 0.
#SS(0)	If the SS register is used to access memory and the value in EAX is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If the current privilege level is not 0. If CPUID.01H:ECX.MONITOR[bit 3] = 0.

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV r/m8,r8	MR	Valid	Valid	Move r8 to r/m8.
REX + 88 /r	MOV r/m8***,r8***	MR	Valid	N.E.	Move r8 to r/m8.
89 /r	MOV r/m16,r16	MR	Valid	Valid	Move r16 to r/m16.
89 /r	MOV r/m32,r32	MR	Valid	Valid	Move r32 to r/m32.
REX.W + 89 /r	MOV r/m64,r64	MR	Valid	N.E.	Move r64 to r/m64.
8A /r	MOV r8,r/m8	RM	Valid	Valid	Move r/m8 to r8.
REX + 8A /r	MOV r8***,r/m8***	RM	Valid	N.E.	Move r/m8 to r8.
8B /r	MOV r16,r/m16	RM	Valid	Valid	Move r/m16 to r16.
8B /r	MOV r32,r/m32	RM	Valid	Valid	Move r/m32 to r32.
REX.W + 8B /r	MOV r64,r/m64	RM	Valid	N.E.	Move r/m64 to r64.
8C /r	MOV r/m16,Sreg**	MR	Valid	Valid	Move segment register to r/m16.
REX.W + 8C /r	MOV r/m64,Sreg**	MR	Valid	Valid	Move zero extended 16-bit segment register to r/m64.
8E /r	MOV Sreg,r/m16**	RM	Valid	Valid	Move r/m16 to segment register.
REX.W + 8E /r	MOV Sreg,r/m64**	RM	Valid	Valid	Move lower 16 bits of r/m64 to segment register.
A0	MOV AL,moffs8*	FD	Valid	Valid	Move byte at (seg:offset) to AL.
REX.W + A0	MOV AL,moffs8*	FD	Valid	N.E.	Move byte at (offset) to AL.
A1	MOV AX,moffs16*	FD	Valid	Valid	Move word at (seg:offset) to AX.
A1	MOV EAX,moffs32*	FD	Valid	Valid	Move doubleword at (seg:offset) to EAX.
REX.W + A1	MOV RAX,moffs64*	FD	Valid	N.E.	Move quadword at (offset) to RAX.
A2	MOV moffs8,AL	TD	Valid	Valid	Move AL to (seg:offset).
REX.W + A2	MOV moffs8***,AL	TD	Valid	N.E.	Move AL to (offset).
A3	MOV moffs16*,AX	TD	Valid	Valid	Move AX to (seg:offset).
A3	MOV moffs32*,EAX	TD	Valid	Valid	Move EAX to (seg:offset).
REX.W + A3	MOV moffs64*,RAX	TD	Valid	N.E.	Move RAX to (offset).
B0+ rb ib	MOV r8, imm8	OI	Valid	Valid	Move imm8 to r8.
REX + B0+ rb ib	MOV r8***, imm8	OI	Valid	N.E.	Move imm8 to r8.
B8+ rw iw	MOV r16, imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32, imm32	OI	Valid	Valid	Move imm32 to r32.
REX.W + B8+ rd io	MOV r64, imm64	OI	Valid	N.E.	Move imm64 to r64.
C6 /0 ib	MOV r/m8, imm8	MI	Valid	Valid	Move imm8 to r/m8.
REX + C6 /0 ib	MOV r/m8***, imm8	MI	Valid	N.E.	Move imm8 to r/m8.
C7 /0 iw	MOV r/m16, imm16	MI	Valid	Valid	Move imm16 to r/m16.
C7 /0 id	MOV r/m32, imm32	MI	Valid	Valid	Move imm32 to r/m32.
REX.W + C7 /0 io	MOV r/m64, imm32	MI	Valid	N.E.	Move imm32 sign extended to 64-bits to r/m64.

NOTES:

- * The *moffs8*, *moffs16*, *moffs32* and *moffs64* operands specify a simple offset relative to the segment base, where 8, 16, 32 and 64 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16, 32 or 64 bits.
- ** In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (see the following “Description” section for further information).
- ***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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FD	AL/AX/EAX/RAX	Moffs	NA	NA
TD	Moffs (w)	AL/AX/EAX/RAX	NA	NA
OI	opcode + rd (w)	imm8/16/32/64	NA	NA
MI	ModRM:r/m (w)	imm8/16/32/64	NA	NA

Description

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, a doubleword, or a quadword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the “Operation” algorithm below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.

A NULL segment selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing a protection exception. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP) and no memory reference occurs.

Loading the SS register with a MOV instruction inhibits all interrupts until after the execution of the next instruction. This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, **stack-pointer value**) before an interrupt occurs¹. Be aware that the LSS instruction offers a more efficient method of loading the SS and ESP registers.

When operating in 32-bit mode and moving data between a segment register and a general-purpose register, the 32-bit IA-32 processors do not require the use of the 16-bit operand-size prefix (a byte with the value 66H) with

1. If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a MOV SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that load 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 MOV ESP, EBP executes:

```
MOV SS, EDX
MOV SS, EAX
MOV ESP, EBP
```

this instruction, but most assemblers will insert it if the standard form of the instruction is used (for example, MOV DS, AX). The processor will execute this instruction correctly, but it will usually require an extra clock. With most assemblers, using the instruction form MOV DS, EAX will avoid this unneeded 66H prefix. When the processor executes the instruction with a 32-bit general-purpose register, it assumes that the 16 least-significant bits of the general-purpose register are the destination or source operand. If the register is a destination operand, the resulting value in the two high-order bytes of the register is implementation dependent. For the Pentium 4, Intel Xeon, and P6 family processors, the two high-order bytes are filled with zeros; for earlier 32-bit IA-32 processors, the two high order bytes are undefined.

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

Operation

DEST ← SRC;

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

IF SS is loaded

THEN

 IF segment selector is NULL

 THEN #GP(0); Fl;

 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); Fl;

 IF segment not marked present

 THEN #SS(selector);

 ELSE

 SS ← segment selector;

 SS ← segment descriptor; Fl;

Fl;

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)

 or ((RPL > DPL) and (CPL > DPL))

 THEN #GP(selector); Fl;

 IF segment not marked present

 THEN #NP(selector);

 ELSE

 SegmentRegister ← segment selector;

 SegmentRegister ← segment descriptor; Fl;

Fl;

IF DS, ES, FS, or GS is loaded with NULL selector

THEN

 SegmentRegister ← segment selector;

 SegmentRegister ← segment descriptor;

Fl;

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 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 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 alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If attempt is made to load the CS register. If the LOCK prefix is used.

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.
#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

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.
#UD	If attempt is made to load the CS register. If the LOCK prefix is used.

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.
 - If an attempt is made to load SS register with NULL segment selector when CPL = 3.
 - If an attempt is made to load SS register with NULL segment selector when CPL < 3 and CPL ≠ RPL.
- #GP(selector)
 - If segment selector index is outside descriptor table limits.
 - If the memory access to the descriptor table is non-canonical.
 - 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 nonwritable 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.
 - If the stack address is in a non-canonical form.
- #SS(0)
- #SS(selector)
- #PF(fault-code)
- #AC(0)
 - If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD
 - If attempt is made to load the CS register.
 - If the LOCK prefix is used.

MOV—Move to/from Control Registers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 20/r	MR	N.E.	Valid	Move control register to <i>r</i> 32.
MOV <i>r</i> 32, CR0-CR7				
OF 20/r	MR	Valid	N.E.	Move extended control register to <i>r</i> 64.
MOV <i>r</i> 64, CR0-CR7				
REX.R + OF 20 /0	MR	Valid	N.E.	Move extended CR8 to <i>r</i> 64. ¹
MOV <i>r</i> 64, CR8				
OF 22 /r	RM	N.E.	Valid	Move <i>r</i> 32 to control register.
MOV CR0-CR7, <i>r</i> 32				
OF 22 /r	RM	Valid	N.E.	Move <i>r</i> 64 to extended control register.
MOV CR0-CR7, <i>r</i> 64				
REX.R + OF 22 /0	RM	Valid	N.E.	Move <i>r</i> 64 to extended CR8. ¹
MOV CR8, <i>r</i> 64				

NOTE:

1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 8 in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the contents of a control register (CR0, CR2, CR3, CR4, or CR8) to a general-purpose register or the contents of a general purpose register to a control register. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See “Control Registers” in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for a detailed description of the flags and fields in the control registers.) This instruction can be executed only when the current privilege level is 0.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read. Attempts to reference CR1, CR5, CR6, CR7, and CR9–CR15 result in undefined opcode (#UD) exceptions.

When loading control registers, programs should not attempt to change the reserved bits; that is, always set reserved bits to the value previously read. An attempt to change CR4’s reserved bits will cause a general protection fault. Reserved bits in CR0 and CR3 remain clear after any load of those registers; attempts to set them have no impact. On Pentium 4, Intel Xeon and P6 family processors, CR0.ET remains set after any load of CR0; attempts to clear this bit have no impact.

In certain cases, these instructions have the side effect of invalidating entries in the TLBs and the paging-structure caches. See Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A* for details.

The following side effects are implementation-specific for the Pentium 4, Intel Xeon, and P6 processor family: when modifying PE or PG in register CR0, or PSE or PAE in register CR4, all TLB entries are flushed, including global entries. Software should not depend on this functionality in all Intel 64 or IA-32 processors.

In 64-bit mode, the instruction’s default operation size is 64 bits. The REX.R prefix must be used to access CR8. Use of REX.B permits access to additional registers (R8-R15). Use of the REX.W prefix or 66H prefix is ignored. Use of

the REX.R prefix to specify a register other than CR8 causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

If CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 determines whether the instruction invalidates entries in the TLBs and the paging-structure caches (see Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). The instruction does not modify bit 63 of CR3, which is reserved and always 0.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

$\text{DEST} \leftarrow \text{SRC};$

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0. If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1). If an attempt is made to write a 1 to any reserved bit in CR4. If an attempt is made to write 1 to CR4.PCIDE. If any of the reserved bits are set in the page-directory pointers table (PDPT) and the loading of a control register causes the PDPT to be loaded into the processor.
#UD	If the LOCK prefix is used. If an attempt is made to access CR1, CR5, CR6, or CR7.

Real-Address Mode Exceptions

#GP	If an attempt is made to write a 1 to any reserved bit in CR4. If an attempt is made to write 1 to CR4.PCIDE. If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0).
#UD	If the LOCK prefix is used. If an attempt is made to access CR1, CR5, CR6, or CR7.

Virtual-8086 Mode Exceptions

#GP(0)	These instructions cannot be executed in virtual-8086 mode.
--------	---

Compatibility Mode Exceptions

#GP(0)	If the current privilege level is not 0. If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1). If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H. If an attempt is made to clear CR0.PG[bit 31] while CR4.PCIDE = 1. If an attempt is made to write a 1 to any reserved bit in CR3. If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].
#UD	If the LOCK prefix is used. If an attempt is made to access CR1, CR5, CR6, or CR7.

64-Bit Mode Exceptions

- #GP(0)
 - If the current privilege level is not 0.
 - If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
 - If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
 - If an attempt is made to clear CR0.PG[bit 31].
 - If an attempt is made to write a 1 to any reserved bit in CR4.
 - If an attempt is made to write a 1 to any reserved bit in CR8.
 - If an attempt is made to write a 1 to any reserved bit in CR3.
 - If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].
- #UD
 - If the LOCK prefix is used.
 - If an attempt is made to access CR1, CR5, CR6, or CR7.
 - If the REX.R prefix is used to specify a register other than CR8.

MOV—Move to/from Debug Registers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 21/r	MR	N.E.	Valid	Move debug register to <i>r</i> 32.
MOV <i>r</i> 32, DR0-DR7				
OF 21/r	MR	Valid	N.E.	Move extended debug register to <i>r</i> 64.
MOV <i>r</i> 64, DR0-DR7				
OF 23 /r	RM	N.E.	Valid	Move <i>r</i> 32 to debug register.
MOV DR0-DR7, <i>r</i> 32				
OF 23 /r	RM	Valid	N.E.	Move <i>r</i> 64 to extended debug register.
MOV DR0-DR7, <i>r</i> 64				

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the contents of a debug register (DR0, DR1, DR2, DR3, DR4, DR5, DR6, or DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See Section 17.2, “Debug Registers”, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for a detailed description of the flags and fields in the debug registers.)

The instructions must be executed at privilege level 0 or in real-address mode.

When the debug extension (DE) flag in register CR4 is clear, these instructions operate on debug registers in a manner that is compatible with Intel386 and Intel486 processors. In this mode, references to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE flag in CR4 is set, attempts to reference DR4 and DR5 result in an undefined opcode (#UD) exception. (The CR4 register was added to the IA-32 Architecture beginning with the Pentium processor.)

At the opcode level, the *reg* field within the ModR/M byte specifies which of the debug registers is loaded or read. The two bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read.

In 64-bit mode, the instruction’s default operation size is 64 bits. Use of the REX.B prefix permits access to additional registers (R8-R15). Use of the REX.W or 66H prefix is ignored. Use of the REX.R prefix causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF ((DE = 1) and (SRC or DEST = DR4 or DR5))
  THEN
    #UD;
  ELSE
    DEST ← SRC;
```

Fl;

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- #UD If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.
If the LOCK prefix is used.
- #DB If any debug register is accessed while the DR7.GD[bit 13] = 1.

Real-Address Mode Exceptions

- #UD If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.
If the LOCK prefix is used.
- #DB If any debug register is accessed while the DR7.GD[bit 13] = 1.

Virtual-8086 Mode Exceptions

- #GP(0) The debug registers cannot be loaded or read when 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 an attempt is made to write a 1 to any of bits 63:32 in DR6.
If an attempt is made to write a 1 to any of bits 63:32 in DR7.
- #UD If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.
If the LOCK prefix is used.
If the REX.R prefix is used.
- #DB If any debug register is accessed while the DR7.GD[bit 13] = 1.

MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Move packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
66 0F 29 /r MOVAPD <i>xmm2/m128, xmm1</i>	MR	V/V	SSE2	Move packed double-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.66.0F.WIG 28 /r VMOVAPD <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move aligned packed double-precision floating-point values from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.128.66.0F.WIG 29 /r VMOVAPD <i>xmm2/m128, xmm1</i>	MR	V/V	AVX	Move aligned packed double-precision floating-point values from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.66.0F.WIG 28 /r VMOVAPD <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move aligned packed double-precision floating-point values from <i>ymm2/mem</i> to <i>ymm1</i> .
VEX.256.66.0F.WIG 29 /r VMOVAPD <i>ymm2/m256, ymm1</i>	MR	V/V	AVX	Move aligned packed double-precision floating-point values from <i>ymm1</i> to <i>ymm2/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves 2 or 4 double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM or YMM register from an 128-bit or 256-bit memory location, to store the contents of an XMM or YMM register into a 128-bit or 256-bit memory location, or to move data between two XMM or two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary or a general-protection exception (#GP) will be generated.

To move double-precision floating-point values to and from unaligned memory locations, use the (V)MOVUPD instruction.

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

128-bit versions: Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register destination are zeroed.

VEX.256 encoded version: Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVAPD (128-bit load- and register-copy- form Legacy SSE version)

DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] (Unmodified)

(V)MOVAPD (128-bit store-form version)

DEST[127:0] ← SRC[127:0]

VMOVAPD (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] ← 0

VMOVAPD (VEX.256 encoded version)

DEST[255:0] ← SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVAPD:    __m128d _mm_load_pd (double const * p);
MOVAPD:    _mm_store_pd(double * p, __m128d a);
VMOVAPD:   __m256d _mm256_load_pd (double const * p);
VMOVAPD:   _mm256_store_pd(double * p, __m256d a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 1.SSE2; additionally

#UD If VEX.vvvv != 1111B.

MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 28 /r MOVAPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Move packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
OF 29 /r MOVAPS <i>xmm2/m128, xmm1</i>	MR	V/V	SSE	Move packed single-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.0F.WIG 28 /r VMOVAPS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move aligned packed single-precision floating-point values from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.128.0F.WIG 29 /r VMOVAPS <i>xmm2/m128, xmm1</i>	MR	V/V	AVX	Move aligned packed single-precision floating-point values from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.0F.WIG 28 /r VMOVAPS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move aligned packed single-precision floating-point values from <i>ymm2/mem</i> to <i>ymm1</i> .
VEX.256.0F.WIG 29 /r VMOVAPS <i>ymm2/m256, ymm1</i>	MR	V/V	AVX	Move aligned packed single-precision floating-point values from <i>ymm1</i> to <i>ymm2/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves 4 or 8 single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM or YMM register from an 128-bit or 256-bit memory location, to store the contents of an XMM or YMM register into a 128-bit or 256-bit memory location, or to move data between two XMM or two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary or a general-protection exception (#GP) will be generated.

To move single-precision floating-point values to and from unaligned memory locations, use the (V)MOVUPS instruction.

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version:

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Operation

MOVAPS (128-bit load- and register-copy- form Legacy SSE version)

DEST[127:0] \leftarrow SRC[127:0]
 DEST[VLMAX-1:128] (Unmodified)

(V)MOVAPS (128-bit store form)

DEST[127:0] \leftarrow SRC[127:0]

VMOVAPS (VEX.128 encoded version)

DEST[127:0] \leftarrow SRC[127:0]
 DEST[VLMAX-1:128] \leftarrow 0

VMOVAPS (VEX.256 encoded version)

DEST[255:0] \leftarrow SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVAPS:      __m128 _mm_load_ps (float const * p);
MOVAPS:      _mm_store_ps(float * p, __m128 a);
VMOVAPS:     __m256 _mm256_load_ps (float const * p);
VMOVAPS:     _mm256_store_ps(float * p, __m256 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 1.SSE; additionally

#UD If VEX.vvvv != 1111B.

MOVBE—Move Data After Swapping Bytes

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 38 F0 /r	MOVBE <i>r16, m16</i>	RM	Valid	Valid	Reverse byte order in <i>m16</i> and move to <i>r16</i> .
OF 38 F0 /r	MOVBE <i>r32, m32</i>	RM	Valid	Valid	Reverse byte order in <i>m32</i> and move to <i>r32</i> .
REX.W + OF 38 F0 /r	MOVBE <i>r64, m64</i>	RM	Valid	N.E.	Reverse byte order in <i>m64</i> and move to <i>r64</i> .
OF 38 F1 /r	MOVBE <i>m16, r16</i>	MR	Valid	Valid	Reverse byte order in <i>r16</i> and move to <i>m16</i> .
OF 38 F1 /r	MOVBE <i>m32, r32</i>	MR	Valid	Valid	Reverse byte order in <i>r32</i> and move to <i>m32</i> .
REX.W + OF 38 F1 /r	MOVBE <i>m64, r64</i>	MR	Valid	N.E.	Reverse byte order in <i>r64</i> and move to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Performs a byte swap operation on the data copied from the second operand (source operand) and store the result in the first operand (destination operand). The source operand can be a general-purpose register, or memory location; the destination register can be a general-purpose register, or a memory location; however, both operands can not be registers, and only one operand can be a memory location. Both operands must be the same size, which can be a word, a doubleword or quadword.

The MOVBE instruction is provided for swapping the bytes on a read from memory or on a write to memory; thus providing support for converting little-endian values to big-endian format and vice versa.

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

Operation

TEMP ← SRC

```

IF ( OperandSize = 16)
  THEN
    DEST[7:0] ← TEMP[15:8];
    DEST[15:8] ← TEMP[7:0];
  ELES IF ( OperandSize = 32)
    DEST[7:0] ← TEMP[31:24];
    DEST[15:8] ← TEMP[23:16];
    DEST[23:16] ← TEMP[15:8];
    DEST[31:23] ← TEMP[7:0];
  ELSE IF ( OperandSize = 64)
    DEST[7:0] ← TEMP[63:56];
    DEST[15:8] ← TEMP[55:48];
    DEST[23:16] ← TEMP[47:40];
    DEST[31:24] ← TEMP[39:32];
    DEST[39:32] ← TEMP[31:24];
    DEST[47:40] ← TEMP[23:16];
    DEST[55:48] ← TEMP[15:8];
    DEST[63:56] ← TEMP[7:0];

```

FI;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	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 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.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used.

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.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used.

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 while the current privilege level is 3.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used. If REPNE (F2H) prefix is used and CPUID.01H:ECX.SSE4_2[bit 20] = 0.

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(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.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used.

MOVD/MOVQ—Move Doubleword/Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 6E /r		RM	V/V	MMX Move doubleword from <i>r/m32</i> to <i>mm</i> .
MOVD <i>mm</i> , <i>r/m32</i>				
REX.W + OF 6E /r		RM	V/N.E.	MMX Move quadword from <i>r/m64</i> to <i>mm</i> .
MOVQ <i>mm</i> , <i>r/m64</i>				
OF 7E /r		MR	V/V	MMX Move doubleword from <i>mm</i> to <i>r/m32</i> .
MOVD <i>r/m32</i> , <i>mm</i>				
REX.W + OF 7E /r		MR	V/N.E.	MMX Move quadword from <i>mm</i> to <i>r/m64</i> .
MOVQ <i>r/m64</i> , <i>mm</i>				
VEX.128.66.0F.W0 6E /		RM	V/V	AVX Move doubleword from <i>r/m32</i> to <i>xmm1</i> .
VMOVD <i>xmm1</i> , <i>r32/m32</i>				
VEX.128.66.0F.W1 6E /r		RM	V/N.E.	AVX Move quadword from <i>r/m64</i> to <i>xmm1</i> .
VMOVQ <i>xmm1</i> , <i>r64/m64</i>				
66 OF 6E /r		RM	V/V	SSE2 Move doubleword from <i>r/m32</i> to <i>xmm</i> .
MOVD <i>xmm</i> , <i>r/m32</i>				
66 REX.W OF 6E /r		RM	V/N.E.	SSE2 Move quadword from <i>r/m64</i> to <i>xmm</i> .
MOVQ <i>xmm</i> , <i>r/m64</i>				
66 OF 7E /r		MR	V/V	SSE2 Move doubleword from <i>xmm</i> register to <i>r/m32</i> .
MOVD <i>r/m32</i> , <i>xmm</i>				
66 REX.W OF 7E /r		MR	V/N.E.	SSE2 Move quadword from <i>xmm</i> register to <i>r/m64</i> .
MOVQ <i>r/m64</i> , <i>xmm</i>				
VEX.128.66.0F.W0 7E /r		MR	V/V	AVX Move doubleword from <i>xmm1</i> register to <i>r/m32</i> .
VMOVD <i>r32/m32</i> , <i>xmm1</i>				
VEX.128.66.0F.W1 7E /r		MR	V/N.E.	AVX Move quadword from <i>xmm1</i> register to <i>r/m64</i> .
VMOVQ <i>r64/m64</i> , <i>xmm1</i>				

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be general-purpose registers, MMX technology registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

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

Operation

MOVD (when destination operand is MMX technology register)

DEST[31:0] ← SRC;
DEST[63:32] ← 00000000H;

MOVD (when destination operand is XMM register)

DEST[31:0] ← SRC;
DEST[127:32] ← 000000000000000000000000000000H;
DEST[VLMAX-1:128] (Unmodified)

MOVD (when source operand is MMX technology or XMM register)

DEST ← SRC[31:0];

VMOVD (VEX-encoded version when destination is an XMM register)

DEST[31:0] ← SRC[31:0]
DEST[VLMAX-1:32] ← 0

MOVQ (when destination operand is XMM register)

DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;
DEST[VLMAX-1:128] (Unmodified)

MOVQ (when destination operand is r/m64)

DEST[63:0] ← SRC[63:0];

MOVQ (when source operand is XMM register or r/m64)

DEST ← SRC[63:0];

VMOVQ (VEX-encoded version when destination is an XMM register)

DEST[63:0] ← SRC[63:0]
DEST[VLMAX-1:64] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVD:	<code>__m64 _mm_cvtsi32_si64 (int i)</code>
MOVD:	<code>int _mm_cvtsi64_si32 (__m64m)</code>
MOVD:	<code>__m128i _mm_cvtsi32_si128 (int a)</code>
MOVD:	<code>int _mm_cvtsi128_si32 (__m128i a)</code>
MOVQ:	<code>__int64 _mm_cvtsi128_si64(__m128i);</code>
MOVQ:	<code>__m128i _mm_cvtsi64_si128(__int64);</code>

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L = 1.
 If VEX.vvvv != 1111B.

MOVDDUP—Move One Double-FP and Duplicate

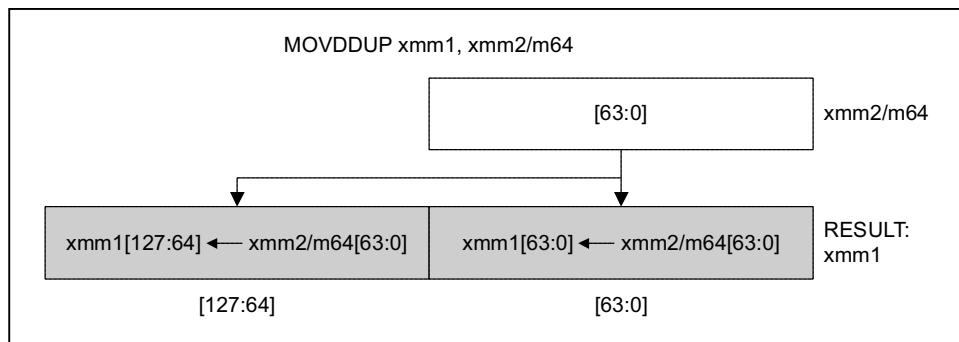
Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 12 /r MOVDDUP <i>xmm1, xmm2/m64</i>	RM	V/V	SSE3	Move one double-precision floating-point value from the lower 64-bit operand in <i>xmm2/m64</i> to <i>xmm1</i> and duplicate.
VEX.128.F2.0F.WIG 12 /r VMOVDDUP <i>xmm1, xmm2/m64</i>	RM	V/V	AVX	Move double-precision floating-point values from <i>xmm2/mem</i> and duplicate into <i>xmm1</i> .
VEX.256.F2.0F.WIG 12 /r VMOVDDUP <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move even index double-precision floating-point values from <i>ymm2/mem</i> and duplicate each element into <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 8 bytes of data at memory location m64 are loaded. When the register-register form of this operation is used, the lower half of the 128-bit source register is duplicated and copied into the 128-bit destination register. See Figure 3-24.



OM15997

Figure 3-24. MOVDDUP—Move One Double-FP and Duplicate

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

Operation

```

IF (Source = m64)
  THEN
    (* Load instruction *)
    xmm1[63:0] = m64;
    xmm1[127:64] = m64;
  ELSE
    (* Move instruction *)
    xmm1[63:0] = xmm2[63:0];
    xmm1[127:64] = xmm2[63:0];
FI;

```

MOVDDUP (128-bit Legacy SSE version)

DEST[63:0] \leftarrow SRC[63:0]
 DEST[127:64] \leftarrow SRC[63:0]
 DEST[VLMAX-1:128] (Unmodified)

VMOVDDUP (VEX.128 encoded version)

DEST[63:0] \leftarrow SRC[63:0]
 DEST[127:64] \leftarrow SRC[63:0]
 DEST[VLMAX-1:128] \leftarrow 0

VMOVDDUP (VEX.256 encoded version)

DEST[63:0] \leftarrow SRC[63:0]
 DEST[127:64] \leftarrow SRC[63:0]
 DEST[191:128] \leftarrow SRC[191:128]
 DEST[255:192] \leftarrow SRC[191:128]

Intel C/C++ Compiler Intrinsic Equivalent

MOVDDUP: `__m128d _mm_movedup_pd(__m128d a)`
 MOVDDUP: `__m128d _mm_loaddup_pd(double const * dp)`

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

MOVDQA—Move Aligned Double Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 6F /r MOVDQA <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Move aligned double quadword from <i>xmm2/m128</i> to <i>xmm1</i> .
66 0F 7F /r MOVDQA <i>xmm2/m128, xmm1</i>	MR	V/V	SSE2	Move aligned double quadword from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.66.0F.WIG 6F /r VMOVDQA <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move aligned packed integer values from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.128.66.0F.WIG 7F /r VMOVDQA <i>xmm2/m128, xmm1</i>	MR	V/V	AVX	Move aligned packed integer values from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.66.0F.WIG 6F /r VMOVDQA <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move aligned packed integer values from <i>ymm2/mem</i> to <i>ymm1</i> .
VEX.256.66.0F.WIG 7F /r VMOVDQA <i>ymm2/m256, ymm1</i>	MR	V/V	AVX	Move aligned packed integer values from <i>ymm1</i> to <i>ymm2/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

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

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVQ (128-bit load- and register-form Legacy SSE version)

DEST[127:0] \leftarrow SRC[127:0]
 DEST[VLMAX-1:128] (Unmodified)
 (* #GP if SRC or DEST unaligned memory operand *)

(V)MOVQ (128-bit store forms)

DEST[127:0] \leftarrow SRC[127:0]

VMOVQ (VEX.128 encoded version)

DEST[127:0] \leftarrow SRC[127:0]
 DEST[VLMAX-1:128] \leftarrow 0

VMOVQ (VEX.256 encoded version)

DEST[255:0] \leftarrow SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVQ:	<code>__m128i _mm_load_si128 (__m128i *p)</code>
MOVQ:	<code>void _mm_store_si128 (__m128i *p, __m128i a)</code>
VMOVQ:	<code>__m256i _mm256_load_si256 (__m256i * p);</code>
VMOVQ:	<code>_mm256_store_si256(__m256i *p, __m256i a);</code>

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 1.SSE2; additionally

#UD If VEX.vvvv != 1111B.

MOVDQU—Move Unaligned Double Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 6F /r MOVDQU <i>xmm1, xmm2/m128</i>	RM	V/V	SSE2	Move unaligned double quadword from <i>xmm2/m128</i> to <i>xmm1</i> .
F3 OF 7F /r MOVDQU <i>xmm2/m128, xmm1</i>	MR	V/V	SSE2	Move unaligned double quadword from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.F3.0F.WIG 6F /r VMOVDQU <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move unaligned packed integer values from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.128.F3.0F.WIG 7F /r VMOVDQU <i>xmm2/m128, xmm1</i>	MR	V/V	AVX	Move unaligned packed integer values from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.F3.0F.WIG 6F /r VMOVDQU <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move unaligned packed integer values from <i>ymm2/mem</i> to <i>ymm1</i> .
VEX.256.F3.0F.WIG 7F /r VMOVDQU <i>ymm2/m256, ymm1</i>	MR	V/V	AVX	Move unaligned packed integer values from <i>ymm1</i> to <i>ymm2/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description**128-bit versions:**

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.¹

To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the MOVDQA instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

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

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned to any alignment without causing a general-protection exception (#GP) to be generated

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory

1. If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the operand is not aligned on an 8-byte boundary.

location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOV DQU load and register copy (128-bit Legacy SSE version)

DEST[127:0] \leftarrow SRC[127:0]

DEST[VLMAX-1:128] (Unmodified)

(V)MOV DQU 128-bit store-form versions

DEST[127:0] \leftarrow SRC[127:0]

VMOV DQU (VEX.128 encoded version)

DEST[127:0] \leftarrow SRC[127:0]

DEST[VLMAX-1:128] \leftarrow 0

VMOV DQU (VEX.256 encoded version)

DEST[255:0] \leftarrow SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOV DQU:	void _mm_storeu_si128 (__m128i *p, __m128i a)
MOV DQU:	__m128i _mm_loadu_si128 (__m128i *p)
VMOV DQU:	__m256i _mm256_loadu_si256 (__m256i * p);
VMOV DQU:	_mm256_storeu_si256(__m256i *p, __m256i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv != 1111B.

MOVDQ2Q—Move Quadword from XMM to MMX Technology Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F D6 /r	MOVDQ2Q <i>mm, xmm</i>	RM	Valid	Valid	Move low quadword from <i>xmm</i> to <i>mmx</i> register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the low quadword from the source operand (second operand) to the destination operand (first operand). The source operand is an XMM register and the destination operand is an MMX technology register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVDQ2Q instruction is executed.

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

Operation

DEST ← SRC[63:0];

Intel C/C++ Compiler Intrinsic Equivalent

MOVDQ2Q: __m64 _mm_movepi64_pi64 (__m128i a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

- #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.
- If the LOCK prefix is used.
- #MF 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.

MOVHLPS— Move Packed Single-Precision Floating-Point Values High to Low

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 12 /r MOVHLPS <i>xmm1, xmm2</i>		RM V/V	SSE	Move two packed single-precision floating-point values from high quadword of <i>xmm2</i> to low quadword of <i>xmm1</i> .
VEX.NDS.128.0F.WIG 12 /r VMOVHLPS <i>xmm1, xmm2, xmm3</i>	RVM	V/V	AVX	Merge two packed single-precision floating-point values from high quadword of <i>xmm3</i> and low quadword of <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The high quadword of the destination operand is left unchanged. Bits (VLMAX-1:64) of the corresponding YMM destination register are unmodified.

128-bit three-argument form

Moves two packed single-precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (VLMAX-1:128) of the destination YMM register are zeroed.

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

If VMOVHLPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVHLPS (128-bit two-argument form)

DEST[63:0] \leftarrow SRC[127:64]

DEST[VLMAX-1:64] (Unmodified)

VMOVHLPS (128-bit three-argument form)

DEST[63:0] \leftarrow SRC2[127:64]

DEST[127:64] \leftarrow SRC1[127:64]

DEST[VLMAX-1:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS: $\text{__m128_mm_movehl_ps}(\text{__m128 a}, \text{__m128 b})$

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.L= 1.

MOVHPD—Move High Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD <i>xmm, m64</i>	RM	V/V	SSE2	Move double-precision floating-point value from <i>m64</i> to high quadword of <i>xmm</i> .
66 0F 17 /r MOVHPD <i>m64, xmm</i>	MR	V/V	SSE2	Move double-precision floating-point value from high quadword of <i>xmm</i> to <i>m64</i> .
VEX.NDS.128.66.0F.WIG 16 /r VMOVHPD <i>xmm2, xmm1, m64</i>	RVM	V/V	AVX	Merge double-precision floating-point value from <i>m64</i> and the low quadword of <i>xmm1</i> .
VEX128.66.0F.WIG 17/r VMOVHPD <i>m64, xmm1</i>	MR	V/V	AVX	Move double-precision floating-point values from high quadword of <i>xmm1</i> to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

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

VEX.128 encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from second XMM register (second operand) are stored in the lower 64-bits of the destination. The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

Stores a double-precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) (VEX.128.66.0F 17 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPD is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVHPD (128-bit Legacy SSE load)

DEST[63:0] (Unmodified)

DEST[127:64] ← SRC[63:0]

DEST[VLMAX-1:128] (Unmodified)

VMOVHPD (VEX.128 encoded load)

DEST[63:0] \leftarrow SRC1[63:0]
DEST[127:64] \leftarrow SRC2[63:0]
DEST[VLMAX-1:128] \leftarrow 0

VMOVHPD (store)

DEST[63:0] \leftarrow SRC[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPD: `__m128d _mm_loadh_pd (__m128d a, double *p)`
MOVHPD: `void _mm_storeh_pd (double *p, __m128d a)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L= 1.

MOVHPS—Move High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 16 /r MOVHPS <i>xmm, m64</i>	RM	V/V	SSE	Move two packed single-precision floating-point values from <i>m64</i> to high quadword of <i>xmm</i> .
OF 17 /r MOVHPS <i>m64, xmm</i>	MR	V/V	SSE	Move two packed single-precision floating-point values from high quadword of <i>xmm</i> to <i>m64</i> .
VEX.NDS.128.0F.WIG 16 /r VMOVHPS <i>xmm2, xmm1, m64</i>	RVM	V/V	AVX	Merge two packed single-precision floating-point values from <i>m64</i> and the low quadword of <i>xmm1</i> .
VEX.128.0F.WIG 17/r VMOVHPS <i>m64, xmm1</i>	MR	V/V	AVX	Move two packed single-precision floating-point values from high quadword of <i>xmm1</i> to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

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

VEX.128 encoded load:

Loads two single-precision floating-point values from the source 64-bit memory operand (third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from second XMM register (second operand) are stored in the lower 64-bits of the destination. The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

Stores two packed single-precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.NDS.128.0F 17 /r) is legal and has the same behavior as the existing OF 17 store. For VMOVHPS (store) (VEX.NDS.128.0F 17 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVHPS (128-bit Legacy SSE load)

DEST[63:0] (Unmodified)
 DEST[127:64] \leftarrow SRC[63:0]
 DEST[VLMAX-1:128] (Unmodified)

VMOVHPS (VEX.128 encoded load)

DEST[63:0] \leftarrow SRC1[63:0]
 DEST[127:64] \leftarrow SRC2[63:0]
 DEST[VLMAX-1:128] \leftarrow 0

VMOVHPS (store)

DEST[63:0] \leftarrow SRC[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS: `__m128d _mm_loadh_pi (__m128d a, __m64 *p)`
 MOVHPS: `void _mm_storeh_pi (__m64 *p, __m128d a)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L= 1.

MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 16 /r MOVLHPS <i>xmm1, xmm2</i>		RM V/V	SSE	Move two packed single-precision floating-point values from low quadword of <i>xmm2</i> to high quadword of <i>xmm1</i> .
VEX.NDS.128.0F.WIG 16 /r VMOVLHPS <i>xmm1, xmm2, xmm3</i>	RVM	V/V	AVX	Merge two packed single-precision floating-point values from low quadword of <i>xmm3</i> and low quadword of <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. The upper 128 bits of the corresponding YMM destination register are unmodified.

128-bit three-argument form

Moves two packed single-precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). The upper 128-bits of the destination YMM register are zeroed.

If VMOVLHPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

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

Operation

MOVLHPS (128-bit two-argument form)

DEST[63:0] (Unmodified)
DEST[127:64] \leftarrow SRC[63:0]
DEST[VLMAX-1:128] (Unmodified)

VMOVLHPS (128-bit three-argument form)

DEST[63:0] \leftarrow SRC1[63:0]
DEST[127:64] \leftarrow SRC2[63:0]
DEST[VLMAX-1:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS: `__m128 _mm_movehl_ps(__m128 a, __m128 b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.L= 1.

MOVLPD—Move Low Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 12 /r MOVLPD <i>xmm, m64</i>		RM	V/V	SSE2 Move double-precision floating-point value from <i>m64</i> to low quadword of <i>xmm</i> register.
66 0F 13 /r MOVLPD <i>m64, xmm</i>		MR	V/V	SSE2 Move double-precision floating-point nvalue from low quadword of <i>xmm</i> register to <i>m64</i> .
VEX.NDS.128.66.0F.WIG 12 /r VMOVLPD <i>xmm2, xmm1, m64</i>		RVM	V/V	AVX Merge double-precision floating-point value from <i>m64</i> and the high quadword of <i>xmm1</i> .
VEX.128.66.0F.WIG 13/r VMOVLPD <i>m64, xmm1</i>		MR	V/V	AVX Move double-precision floating-point values from low quadword of <i>xmm1</i> to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

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

VEX.128 encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

Stores a double-precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLPD (store) (VEX.128.66.0F 13 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPD is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVLPD (128-bit Legacy SSE load)

DEST[63:0] ← SRC[63:0]
DEST[VLMAX-1:64] (Unmodified)

VMOVLPD (VEX.128 encoded load)

DEST[63:0] \leftarrow SRC2[63:0]
DEST[127:64] \leftarrow SRC1[127:64]
DEST[VLMAX-1:128] \leftarrow 0

VMOVLPD (store)

DEST[63:0] \leftarrow SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPD: `__m128d _mm_loadl_pd (__m128d a, double *p)`
MOVLPD: `void _mm_storel_pd (double *p, __m128d a)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L= 1.
 If VEX.vvvv != 1111B.

MOVLP—Move Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 12 /r MOVLP <i>xmm, m64</i>	RM	V/V	SSE	Move two packed single-precision floating-point values from <i>m64</i> to low quadword of <i>xmm</i> .
OF 13 /r MOVLP <i>m64, xmm</i>	MR	V/V	SSE	Move two packed single-precision floating-point values from low quadword of <i>xmm</i> to <i>m64</i> .
VEX.NDS.128.0F.WIG 12 /r VMOVLP <i>xmm2, xmm1, m64</i>	RVM	V/V	AVX	Merge two packed single-precision floating-point values from <i>m64</i> and the high quadword of <i>xmm1</i> .
VEX.128.0F.WIG 13/r VMOVLP <i>m64, xmm1</i>	MR	V/V	AVX	Move two packed single-precision floating-point values from low quadword of <i>xmm1</i> to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. The upper 128-bits of the corresponding YMM destination register are preserved.

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

VEX.128 encoded load:

Loads two packed single-precision floating-point values from the source 64-bit memory operand (third operand), merges them with the upper 64-bits of the first source XMM register (second operand), and stores them in the low 128-bits of the destination XMM register (first operand). The upper 128-bits of the destination YMM register are zeroed.

128-bit store:

Loads two packed single-precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLP (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing OF 13 store. For VMOVLP (store) (VEX.128.0F 13 /r) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

If VMOVLP is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVLP(S) (128-bit Legacy SSE load)

DEST[63:0] \leftarrow SRC[63:0]
 DEST[VLMAX-1:64] (Unmodified)

VMOVLP(S) (VEX.128 encoded load)

DEST[63:0] \leftarrow SRC2[63:0]
 DEST[127:64] \leftarrow SRC1[127:64]
 DEST[VLMAX-1:128] \leftarrow 0

VMOVLP(S) (store)

DEST[63:0] \leftarrow SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLP(S): `__m128 _mm_loadl_pi (__m128 a, __m64 *p)`
 MOVLP(S): `void _mm_storel_pi (__m64 *p, __m128 a)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.L= 1.
 If VEX.vvvv != 1111B.

MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 50 /r MOVMSKPD <i>reg, xmm</i>		RM V/V	SSE2	Extract 2-bit sign mask from <i>xmm</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are filled with zeros.
VEX.128.66.0F.WIG 50 /r VMOVMSKPD <i>reg, xmm2</i>		RM V/V	AVX	Extract 2-bit sign mask from <i>xmm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.
VEX.256.66.0F.WIG 50 /r VMOVMSKPD <i>reg, ymm2</i>		RM V/V	AVX	Extract 4-bit sign mask from <i>ymm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Extracts the sign bits from the packed double-precision floating-point values in the source operand (second operand), formats them into a 2-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM register, and the destination operand is a general-purpose register. The mask is stored in the 2 low-order bits of the destination operand. Zero-extend the upper bits of the destination.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation**(V)MOVMSKPD (128-bit versions)**

```
DEST[0] ← SRC[63]
DEST[1] ← SRC[127]
IF DEST = r32
    THEN DEST[31:2] ← 0;
    ELSE DEST[63:2] ← 0;
FI
```

VMOVMSKPD (VEX.256 encoded version)

```
DEST[0] ← SRC[63]
DEST[1] ← SRC[127]
DEST[2] ← SRC[191]
DEST[3] ← SRC[255]
IF DEST = r32
    THEN DEST[31:4] ← 0;
    ELSE DEST[63:4] ← 0;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVMSKPD: int _mm_movemask_pd (__m128d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.vvvv != 1111B.

MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 50 /r MOVMSKPS <i>reg</i> , <i>xmm</i>	RM	V/V	SSE	Extract 4-bit sign mask from <i>xmm</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are filled with zeros.
VEX.128.0F.WIG 50 /r VMOVMSKPS <i>reg</i> , <i>xmm2</i>	RM	V/V	AVX	Extract 4-bit sign mask from <i>xmm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.
VEX.256.0F.WIG 50 /r VMOVMSKPS <i>reg</i> , <i>ymm2</i>	RM	V/V	AVX	Extract 8-bit sign mask from <i>ymm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.

Instruction Operand Encoding¹

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Extracts the sign bits from the packed single-precision floating-point values in the source operand (second operand), formats them into a 4- or 8-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM or YMM register, and the destination operand is a general-purpose register. The mask is stored in the 4 or 8 low-order bits of the destination operand. The upper bits of the destination operand beyond the mask are filled with zeros.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

```

DEST[0] ← SRC[31];
DEST[1] ← SRC[63];
DEST[2] ← SRC[95];
DEST[3] ← SRC[127];

IF DEST = r32
    THEN DEST[31:4] ← ZeroExtend;
    ELSE DEST[63:4] ← ZeroExtend;
FI;
```

1. ModRM.MOD = 011B required

(V)MOVMSKPS (128-bit version)

```

DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
IF DEST = r32
    THEN DEST[31:4] ← 0;
    ELSE DEST[63:4] ← 0;
FI

```

VMOVMSKPS (VEX.256 encoded version)

```

DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
DEST[4] ← SRC[159]
DEST[5] ← SRC[191]
DEST[6] ← SRC[223]
DEST[7] ← SRC[255]
IF DEST = r32
    THEN DEST[31:8] ← 0;
    ELSE DEST[63:8] ← 0;
FI

```

Intel C/C++ Compiler Intrinsic Equivalent

```

int _mm_movemask_ps(__m128 a)
int _mm256_movemask_ps(__m256 a)

```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD If VEX.vvvv != 1111B.

MOVNTDQA – Load Double Quadword Non-Temporal Aligned Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA <i>xmm1, m128</i>	RM	V/V	SSE4_1	Move double quadword from <i>m128</i> to <i>xmm</i> using non-temporal hint if WC memory type.
VEX.128.66.0F38.WIG 2A /r VMOVNTDQA <i>xmm1, m128</i>	RM	V/V	AVX	Move double quadword from <i>m128</i> to <i>xmm</i> using non-temporal hint if WC memory type.
VEX.256.66.0F38.WIG 2A /r VMOVNTDQA <i>ymm1, m256</i>	RM	V/V	AVX2	Move 256-bit data from <i>m256</i> to <i>ymm</i> using non-temporal hint if WC memory type.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

(V)MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint. A processor implementation may make use of the non-temporal hint associated with this instruction if the memory source is WC (write combining) memory type. An implementation may also make use of the non-temporal hint associated with this instruction if the memory source is WB (write back) memory type.

A processor's implementation of the non-temporal hint does not override the effective memory type semantics, but the implementation of the hint is processor dependent. For example, a processor implementation may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Another implementation of the hint for WC memory type may optimize data transfer throughput of WC reads. A third implementation may optimize cache reads generated by (V)MOVNTDQA on WB memory type to reduce cache evictions.

WC Streaming Load Hint

For WC memory type in particular, the processor never appears to read the data into the cache hierarchy. Instead, the non-temporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

- A load operation other than a (V)MOVNTDQA which references memory already resident in a temporary internal buffer.
- A non-WC reference to memory already resident in a temporary internal buffer.
- Interleaving of reads and writes to memory currently residing in a single temporary internal buffer.
- Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.
- Certain micro-architectural conditions including resource shortages, detection of a mis-speculation condition, and various fault conditions

The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in Chapter 11, "Memory Cache Control" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Because the WC protocol uses a weakly-ordered memory consistency model, an MFENCE or locked instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might reference the same WC memory locations or in order to synchronize reads of a processor with writes by other agents in the system.

Because of the speculative nature of fetching due to MOVNTDQA, Streaming loads must not be used to reference memory addresses that are mapped to I/O devices having side effects or when reads to these devices are destruc-

tive. For additional information on MOVNTDQA usages, see Section 12.10.3 in Chapter 12, “Programming with SSE3, SSSE3 and SSE4” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.

The 256-bit VMOVNTDQA addresses must be 32-byte aligned or the instruction will cause a #GP.

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

MOVNTDQA (128bit- Legacy SSE form)

DEST \leftarrow SRC

DEST[VLMAX-1:128] (Unmodified)

VMOVNTDQA (VEX.128 encoded form)

DEST \leftarrow SRC

DEST[VLMAX-1:128] \leftarrow 0

VMOVNTDQA (VEX.256 encoded form)

DEST[255:0] \leftarrow SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

(V)MOVNTDQA: `__m128i _mm_stream_load_si128 (__m128i *p);`

VMOVNTDQA: `__m256i _mm256_stream_load_si256 (const __m256i *p);`

Flags Affected

None

Other Exceptions

See Exceptions Type 1.SSE4.1; additionally

#UD If VEX.L= 1.

 If VEX.vvvv != 1111B.

MOVNTDQ—Store Double Quadword Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F E7 /r MOVNTDQ <i>m128</i> , <i>xmm</i>	MR	V/V	SSE2	Move double quadword from <i>xmm</i> to <i>m128</i> using non-temporal hint.
VEX.128.66.0F.WIG E7 /r VMOVNTDQ <i>m128</i> , <i>xmm1</i>	MR	V/V	AVX	Move packed integer values in <i>xmm1</i> to <i>m128</i> using non-temporal hint.
VEX.256.66.0F.WIG E7 /r VMOVNTDQ <i>m256</i> , <i>ymm1</i>	MR	V/V	AVX	Move packed integer values in <i>ymm1</i> to <i>m256</i> using non-temporal hint.

Instruction Operand Encoding¹

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register or YMM register, which is assumed to contain integer data (packed bytes, words, doublewords, or quadwords). The destination operand is a 128-bit or 256-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncachable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVNTDQ: void _mm_stream_si128( __m128i *p, __m128i a);
VMOVNTDQ: void _mm256_stream_si256 ( __m256i * p, __m256i a);
```

SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B is not permitted

Other Exceptions

See Exceptions Type 1.SSE2; additionally
#UD If VEX.vvvv != 1111B.

MOVNTI—Store Doubleword Using Non-Temporal Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF C3 /r	MOVNTI <i>m32, r32</i>	MR	Valid	Valid	Move doubleword from <i>r32</i> to <i>m32</i> using non-temporal hint.
REX.W + OF C3 /r	MOVNTI <i>m64, r64</i>	MR	Valid	N.E.	Move quadword from <i>r64</i> to <i>m64</i> using non-temporal hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is a general-purpose register. The destination operand is a 32-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTI instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVNTI:    void _mm_stream_si32 (int *p, int a)
MOVNTI:    void _mm_stream_si64(__int64 *p, __int64 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.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0. If the LOCK prefix is used.

Real-Address Mode Exceptions

- #GP 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.
- #UD If CPUID.01H:EDX.SSE2[bit 26] = 0.
- If the LOCK prefix is used.

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.
- #PF(fault-code) For a page fault.
- #UD If CPUID.01H:EDX.SSE2[bit 26] = 0.
- If the LOCK prefix is used.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 2B /r MOVNTPD <i>m128</i> , <i>xmm</i>	MR	V/V	SSE2	Move packed double-precision floating-point values from <i>xmm</i> to <i>m128</i> using non-temporal hint.
VEX.128.66.0F.WIG 2B /r VMOVNTPD <i>m128</i> , <i>xmm1</i>	MR	V/V	AVX	Move packed double-precision values in <i>xmm1</i> to <i>m128</i> using non-temporal hint.
VEX.256.66.0F.WIG 2B /r VMOVNTPD <i>m256</i> , <i>ymm1</i>	MR	V/V	AVX	Move packed double-precision values in <i>ymm1</i> to <i>m256</i> using non-temporal hint.

Instruction Operand Encoding¹

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed double-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register or YMM register, which is assumed to contain packed double-precision, floating-pointing data. The destination operand is a 128-bit or 256-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTPD: void _mm_stream_pd(double *p, __m128d a)
VMOVNTPD: void _mm256_stream_pd (double * p, __m256d a);

SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B is not permitted

Other Exceptions

See Exceptions Type 1.SSE2; additionally
#UD If VEX.vvvv != 1111B.

MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 2B /r MOVNTPS <i>m128</i> , <i>xmm</i>	MR	V/V	SSE	Move packed single-precision floating-point values from <i>xmm</i> to <i>m128</i> using non-temporal hint.
VEX.128.OF.WIG 2B /r VMOVNTPS <i>m128</i> , <i>xmm1</i>	MR	V/V	AVX	Move packed single-precision values <i>xmm1</i> to <i>mem</i> using non-temporal hint.
VEX.256.OF.WIG 2B /r VMOVNTPS <i>m256</i> , <i>ymm1</i>	MR	V/V	AVX	Move packed single-precision values <i>ymm1</i> to <i>mem</i> using non-temporal hint.

Instruction Operand Encoding¹

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register or YMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit or 256-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version) or 32-byte (VEX.256 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

DEST \leftarrow SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTDQ: void _mm_stream_ps(float * p, __m128 a)
 VMOVNTPS: void _mm256_stream_ps (float * p, __m256 a);

SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B is not permitted

Other Exceptions

See Exceptions Type 1.SSE; additionally
#UD If VEX.vvvv != 1111B.

MOVNTQ—Store of Quadword Using Non-Temporal Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF E7 /r	MOVNTQ <i>m64, mm</i>	MR	Valid	Valid	Move quadword from <i>mm</i> to <i>m64</i> using non-temporal hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an MMX technology register, which is assumed to contain packed integer data (packed bytes, words, or doublewords). The destination operand is a 64-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTQ: void _mm_stream_pi(__m64 * p, __m64 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

MOVQ—Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 6F /r		RM	V/V	MMX Move quadword from <i>mm/m64</i> to <i>mm</i> .
MOVQ <i>mm, mm/m64</i>		MR	V/V	MMX Move quadword from <i>mm</i> to <i>mm/m64</i> .
OF 7F /r		RM	V/V	MMX Move quadword from <i>mm/m64, mm</i>
MOVQ <i>mm/m64, mm</i>		MR	V/V	MMX Move quadword from <i>xmm2/mem64</i> to <i>xmm1</i> .
F3 OF 7E /r		RM	V/V	SSE2 Move quadword from <i>xmm2/mem64</i> to <i>xmm1</i> .
MOVQ <i>xmm1, xmm2/m64</i>		RM	V/V	AVX Move quadword from <i>xmm2</i> to <i>xmm1</i> .
VEX.128.F3.0F.WIG 7E /r		RM	V/V	AVX VMOVQ <i>xmm1, xmm2</i>
VEX.128.F3.0F.WIG 7E /r		RM	V/V	AVX VMOVQ <i>xmm1, m64</i>
VMOVQ <i>xmm1, m64</i>		MR	V/V	SSE2 Load quadword from <i>m64</i> to <i>xmm1</i> .
66 OF D6 /r		MR	V/V	SSE2 Move quadword from <i>xmm1</i> to <i>xmm2/mem64</i> .
MOVQ <i>xmm2/m64, xmm1</i>		MR	V/V	AVX Move quadword from <i>xmm2</i> register to <i>xmm1/m64</i> .
VEX.128.66.0F.WIG D6 /r		MR	V/V	AVX VMOVQ <i>xmm1/m64, xmm2</i>
VMOVQ <i>xmm1/m64, xmm2</i>				

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

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

Note: In VEX.128.66.0F D6 instruction version, VEX.vvvv and VEX.L=1 are reserved and the former must be 1111b otherwise instructions will #UD.

Note: In VEX.128.F3.0F 7E version, VEX.vvvv and VEX.L=1 are reserved and the former must be 1111b, otherwise instructions will #UD.

Operation

MOVQ instruction when operating on MMX technology registers and memory locations:

DEST \leftarrow SRC;

MOVQ instruction when source and destination operands are XMM registers:

DEST[63:0] \leftarrow SRC[63:0];

DEST[127:64] \leftarrow 0000000000000000H;

MOVQ instruction when source operand is XMM register and destination operand is memory location:

```
DEST ← SRC[63:0];
```

MOVQ instruction when source operand is memory location and destination operand is XMM register:

```
DEST[63:0] ← SRC;
DEST[127:64] ← 0000000000000000H;
```

VMOVQ (VEX.NDS.128.F3.0F 7E) with XMM register source and destination:

```
DEST[63:0] ← SRC[63:0]
DEST[VLMAX-1:64] ← 0
```

VMOVQ (VEX.128.66.0F D6) with XMM register source and destination:

```
DEST[63:0] ← SRC[63:0]
DEST[VLMAX-1:64] ← 0
```

VMOVQ (7E) with memory source:

```
DEST[63:0] ← SRC[63:0]
DEST[VLMAX-1:64] ← 0
```

VMOVQ (D6) with memory dest:

```
DEST[63:0] ← SRC2[63:0]
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVQ: m128i _mm_mov_epi64(__m128i a)
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 22-8, "Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

MOVQ2DQ—Move Quadword from MMX Technology to XMM Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 OF D6 /r	MOVQ2DQ <i>xmm, mm</i>	RM	Valid	Valid	Move quadword from <i>mmx</i> to low quadword of <i>xmm</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the quadword from the source operand (second operand) to the low quadword of the destination operand (first operand). The source operand is an MMX technology register and the destination operand is an XMM register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVQ2DQ instruction is executed.

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

Operation

```
DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVQ2DQ: __128i_mm_movpi64_pi64 ( __m64 a)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#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.
	If the LOCK prefix is used.
#MF	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.

MOVS/MOVSB/MOVSW/MOVSD/MOVSQ—Move Data from String to String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A4	MOVS <i>m8, m8</i>	NP	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R E)SI to (R E)DI.
A5	MOVS <i>m16, m16</i>	NP	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R E)SI to (R E)DI.
A5	MOVS <i>m32, m32</i>	NP	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R E)SI to (R E)DI.
REX.W + A5	MOVS <i>m64, m64</i>	NP	Valid	N.E.	Move qword from address (R E)SI to (R E)DI.
A4	MOVSB	NP	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R E)SI to (R E)DI.
A5	MOVSW	NP	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R E)SI to (R E)DI.
A5	MOVSD	NP	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R E)SI to (R E)DI.
REX.W + A5	MOVSQ	NP	Valid	N.E.	Move qword from address (R E)SI to (R E)DI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Moves the byte, word, or doubleword specified with the second operand (source operand) to the location specified with the first operand (destination operand). Both the source and destination operands are located in memory. The address of the source operand is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The address of the destination operand is read from the ES:EDI or the ES:DI registers (again depending on the address-size attribute of the instruction). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

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 MOVS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source and destination operands should be symbols that indicate the size and location of the source value and the destination, respectively. 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 and destination operand symbols must specify the correct **type** (size) of the operands (bytes, words, or doublewords), but they do not have to specify the correct **location**. The locations of the source and destination operands are always specified by the DS:(E)SI and ES:(E)DI registers, which must be loaded correctly before the move string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the MOVS instructions. Here also DS:(E)SI and ES:(E)DI are assumed to be the source and destination operands, respectively. The size of the source and destination operands is selected with the mnemonic: MOVSB (byte move), MOVSW (word move), or MOVSD (doubleword move).

After the move operation, the (E)SI and (E)DI registers are 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 and (E)DI register are incre-

mented; if the DF flag is 1, the (E)SI and (E)DI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

NOTE

To improve performance, more recent processors support modifications to the processor's operation during the string store operations initiated with MOVS and MOVS_B. See Section 7.3.9.3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for additional information on fast-string operation.

The MOVS, MOVS_B, MOVSW, and MOVS_D instructions can be preceded by the REP prefix (see "REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix" in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a description of the REP prefix) for block moves of ECX bytes, words, or doublewords.

In 64-bit mode, the instruction's default address size is 64 bits, 32-bit address size is supported using the prefix 67H. The 64-bit addresses are specified by RSI and RDI; 32-bit address are specified by ESI and EDI. Use of the REX.W prefix promotes doubleword operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← SRC;

Non-64-bit Mode:

```

IF (Byte move)
  THEN IF DF = 0
    THEN
      (E)SI ← (E)SI + 1;
      (E)DI ← (E)DI + 1;
    ELSE
      (E)SI ← (E)SI - 1;
      (E)DI ← (E)DI - 1;
    FI;
  ELSE IF (Word move)
    THEN IF DF = 0
      (E)SI ← (E)SI + 2;
      (E)DI ← (E)DI + 2;
    FI;
  ELSE
    (E)SI ← (E)SI - 2;
    (E)DI ← (E)DI - 2;
  FI;
ELSE IF (Doubleword move)
  THEN IF DF = 0
    (E)SI ← (E)SI + 4;
    (E)DI ← (E)DI + 4;
  FI;
  ELSE
    (E)SI ← (E)SI - 4;
    (E)DI ← (E)DI - 4;
  FI;
FI;

```

64-bit Mode:

```

IF (Byte move)
  THEN IF DF = 0

```

```

THEN
    (R|E)SI ← (R|E)SI + 1;
    (R|E)DI ← (R|E)DI + 1;
ELSE
    (R|E)SI ← (R|E)SI - 1;
    (R|E)DI ← (R|E)DI - 1;
FI;
ELSE IF (Word move)
THEN IF DF = 0
    (R|E)SI ← (R|E)SI + 2;
    (R|E)DI ← (R|E)DI + 2;
    FI;
ELSE
    (R|E)SI ← (R|E)SI - 2;
    (R|E)DI ← (R|E)DI - 2;
    FI;
ELSE IF (Doubleword move)
THEN IF DF = 0
    (R|E)SI ← (R|E)SI + 4;
    (R|E)DI ← (R|E)DI + 4;
    FI;
ELSE
    (R|E)SI ← (R|E)SI - 4;
    (R|E)DI ← (R|E)DI - 4;
    FI;
ELSE IF (Quadword move)
THEN IF DF = 0
    (R|E)SI ← (R|E)SI + 8;
    (R|E)DI ← (R|E)DI + 8;
    FI;
ELSE
    (R|E)SI ← (R|E)SI - 8;
    (R|E)DI ← (R|E)DI - 8;
    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 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.
#UD	If the LOCK prefix is used.

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.

#UD If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

MOVSD—Move Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 10 /r MOVSD <i>xmm1, xmm2/m64</i>	RM	V/V	SSE2	Move scalar double-precision floating-point value from <i>xmm2/m64</i> to <i>xmm1</i> register.
VEX.NDS.LIG.F2.0F.WIG 10 /r VMOVSD <i>xmm1, xmm2, xmm3</i>	RVM	V/V	AVX	Merge scalar double-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> to <i>xmm1</i> register.
VEX.LIG.F2.0F.WIG 10 /r VMOVSD <i>xmm1, m64</i>	XM	V/V	AVX	Load scalar double-precision floating-point value from <i>m64</i> to <i>xmm1</i> register.
F2 OF 11 /r MOVSD <i>xmm2/m64, xmm1</i>	MR	V/V	SSE2	Move scalar double-precision floating-point value from <i>xmm1</i> register to <i>xmm2/m64</i> .
VEX.NDS.LIG.F2.0F.WIG 11 /r VMOVSD <i>xmm1, xmm2, xmm3</i>	MVR	V/V	AVX	Merge scalar double-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> registers to <i>xmm1</i> .
VEX.LIG.F2.0F.WIG 11 /r VMOVSD <i>m64, xmm1</i>	MR	V/V	AVX	Move scalar double-precision floating-point value from <i>xmm1</i> register to <i>m64</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
XM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MVR	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	NA

Description

MOVSD moves a scalar double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double-precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double-precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

For non-VEX encoded instruction syntax and when the source and destination operands are XMM registers, the high quadword of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the high quadword of the destination operand is cleared to all 0s.

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

Note: For the “VMOVSD *m64, xmm1*” (memory store form) instruction version, VEX.vvvv is reserved and must be 1111b, otherwise instruction will #UD.

Note: For the “VMOVSD *xmm1, m64*” (memory load form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

VEX encoded instruction syntax supports two source operands and a destination operand if ModR/M.mod field is 11B. VEX.vvvv is used to encode the first source operand (the second operand). The low 128 bits of the destination operand stores the result of merging the low quadword of the second source operand with the quad word in bits 127:64 of the first source operand. The upper bits of the destination operand are cleared.

Operation

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, XMM2)

DEST[63:0] ← SRC[63:0]

DEST[VLMAX-1:64] (Unmodified)

MOVSD/VMOVSD (128-bit versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)

DEST[63:0] ← SRC[63:0]

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, m64)

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← 0

DEST[VLMAX-1:128] (Unmodified)

VMOVSD (VEX.NDS.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)

DEST[63:0] ← SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)

DEST[63:0] ← SRC2[63:0]

DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, m64)

DEST[63:0] ← SRC[63:0]

DEST[VLMAX-1:64] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVSD: `__m128d _mm_load_sd(double *p)`

MOVSD: `void _mm_store_sd(double *p, __m128d a)`

MOVSD: `__m128d _mm_store_sd(__m128d a, __m128d b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

MOVSHDUP—Move Packed Single-FP High and Duplicate

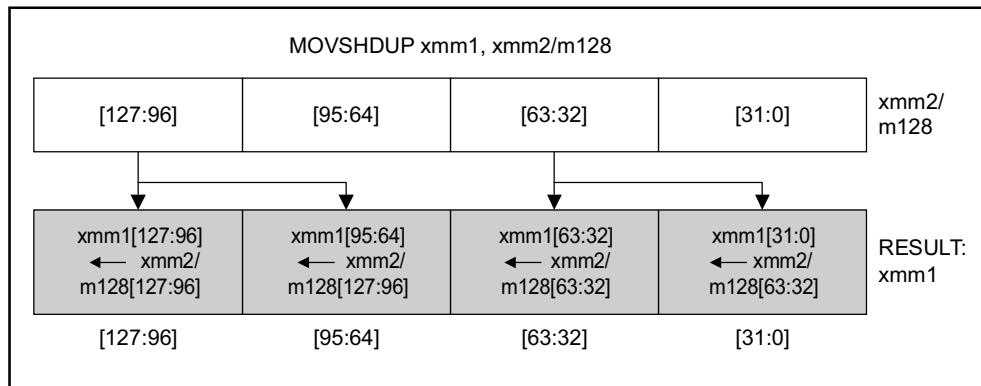
Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 16 /r MOVSHDUP <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Move two single-precision floating-point values from the higher 32-bit operand of each qword in <i>xmm2/m128</i> to <i>xmm1</i> and duplicate each 32-bit operand to the lower 32-bits of each qword.
VEX.128.F3.0F.WIG 16 /r VMOVSHDUP <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move odd index single-precision floating-point values from <i>xmm2/mem</i> and duplicate each element into <i>xmm1</i> .
VEX.256.F3.0F.WIG 16 /r VMOVSHDUP <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move odd index single-precision floating-point values from <i>ymm2/mem</i> and duplicate each element into <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location m128 are loaded and the single-precision elements in positions 1 and 3 are duplicated. When the register-register form of this operation is used, the same operation is performed but with data coming from the 128-bit source register. See Figure 3-25.



OM15998

Figure 3-25. MOVSHDUP—Move Packed Single-FP High and Duplicate

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

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVSHDUP (128-bit Legacy SSE version)

DEST[31:0] ← SRC[63:32]
 DEST[63:32] ← SRC[63:32]
 DEST[95:64] ← SRC[127:96]
 DEST[127:96] ← SRC[127:96]
 DEST[VLMAX-1:128] (Unmodified)

VMOVSHDUP (VEX.128 encoded version)

DEST[31:0] ← SRC[63:32]
 DEST[63:32] ← SRC[63:32]
 DEST[95:64] ← SRC[127:96]
 DEST[127:96] ← SRC[127:96]
 DEST[VLMAX-1:128] ← 0

VMOVSHDUP (VEX.256 encoded version)

DEST[31:0] ← SRC[63:32]
 DEST[63:32] ← SRC[63:32]
 DEST[95:64] ← SRC[127:96]
 DEST[127:96] ← SRC[127:96]
 DEST[159:128] ← SRC[191:160]
 DEST[191:160] ← SRC[191:160]
 DEST[223:192] ← SRC[255:224]
 DEST[255:224] ← SRC[255:224]

Intel C/C++ Compiler Intrinsic Equivalent

(V)MOVSHDUP: `__m128 _mm_movehdup_ps(__m128 a)`
 VMOVSHDUP: `__m256 _mm256_movehdup_ps (__m256 a);`

Exceptions

General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions

None

Other Exceptions

See Exceptions Type 2.

MOVSLDUP—Move Packed Single-FP Low and Duplicate

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 12 /r MOVSLDUP <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Move two single-precision floating-point values from the lower 32-bit operand of each qword in <i>xmm2/m128</i> to <i>xmm1</i> and duplicate each 32-bit operand to the higher 32-bits of each qword.
VEX.128.F3.0F.WIG 12 /r VMOVSLDUP <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move even index single-precision floating-point values from <i>xmm2/mem</i> and duplicate each element into <i>xmm1</i> .
VEX.256.F3.0F.WIG 12 /r VMOVSLDUP <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move even index single-precision floating-point values from <i>ymm2/mem</i> and duplicate each element into <i>ymm1</i> .

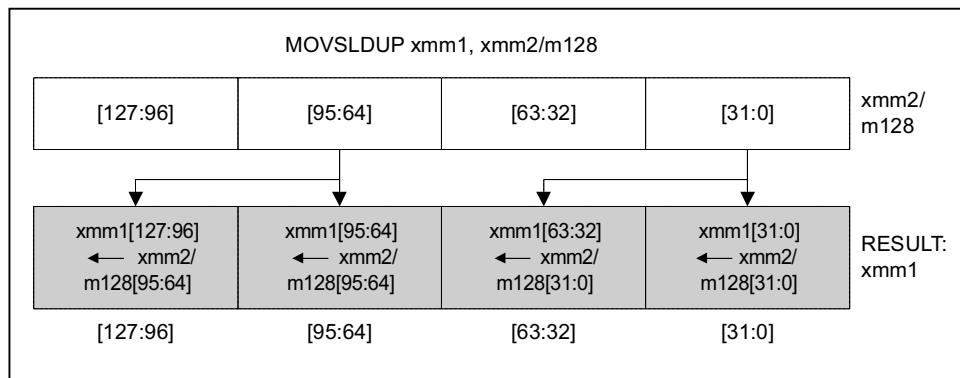
Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location m128 are loaded and the single-precision elements in positions 0 and 2 are duplicated. When the register-register form of this operation is used, the same operation is performed but with data coming from the 128-bit source register.

See Figure 3-26.



OM15999

Figure 3-26. MOVSLDUP—Move Packed Single-FP Low and Duplicate

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

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVSLDUP (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[VLMAX-1:128] (Unmodified)
```

VMOVSLDUP (VEX.128 encoded version)

```
DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[VLMAX-1:128] ← 0
```

VMOVSLDUP (VEX.256 encoded version)

```
DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[159:128] ← SRC[159:128]
DEST[191:160] ← SRC[159:128]
DEST[223:192] ← SRC[223:192]
DEST[255:224] ← SRC[223:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

```
(V)MOVSLDUP:   __m128 _mm_moveldup_ps(__m128 a)
VMOVSLDUP:   __m256 _mm256_moveldup_ps (__m256 a);
```

Exceptions

General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv != 1111B.

MOVSS—Move Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 10 /r MOVSS <i>xmm1, xmm2/m32</i>	RM	V/V	SSE	Move scalar single-precision floating-point value from <i>xmm2/m32</i> to <i>xmm1</i> register.
VEX.NDS.LIG.F3.0F.WIG 10 /r VMOVSS <i>xmm1, xmm2, xmm3</i>	RVM	V/V	AVX	Merge scalar single-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> to <i>xmm1</i> register.
VEX.LIG.F3.0F.WIG 10 /r VMOVSS <i>xmm1, m32</i>	XM	V/V	AVX	Load scalar single-precision floating-point value from <i>m32</i> to <i>xmm1</i> register.
F3 OF 11 /r MOVSS <i>xmm2/m32, xmm</i>	MR	V/V	SSE	Move scalar single-precision floating-point value from <i>xmm1</i> register to <i>xmm2/m32</i> .
VEX.NDS.LIG.F3.0F.WIG 11 /r VMOVSS <i>xmm1, xmm2, xmm3</i>	MVR	V/V	AVX	Move scalar single-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> to <i>xmm1</i> register.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS <i>m32, xmm1</i>	MR	V/V	AVX	Move scalar single-precision floating-point value from <i>xmm1</i> register to <i>m32</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
XM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MVR	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	NA

Description

Moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

For non-VEX encoded syntax and when the source and destination operands are XMM registers, the high doublewords of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the high doublewords of the destination operand is cleared to all 0s.

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

VEX encoded instruction syntax supports two source operands and a destination operand if ModR/M.mod field is 11B. VEX.vvvv is used to encode the first source operand (the second operand). The low 128 bits of the destination operand stores the result of merging the low dword of the second source operand with three dwords in bits 127:32 of the first source operand. The upper bits of the destination operand are cleared.

Note: For the "VMOVSS *m32, xmm1*" (memory store form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Note: For the "VMOVSS *xmm1, m32*" (memory load form) instruction version, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Operation

MOVSS (Legacy SSE version when the source and destination operands are both XMM registers)

DEST[31:0] ← SRC[31:0]

DEST[VLMAX-1:32] (Unmodified)

MOVSS/VMOVSS (when the source operand is an XMM register and the destination is memory)

DEST[31:0] ← SRC[31:0]

DEST[127:32] ← 0

DEST[VLMAX-1:128] (Unmodified)

VMOVSS (VEX.NDS.128.F3.0F 11 /r where the destination is an XMM register)

DEST[31:0] ← SRC2[31:0]

DEST[127:32] ← SRC1[127:32]

DEST[VLMAX-1:128] ← 0

VMOVSS (VEX.NDS.128.F3.0F 10 /r where the source and destination are XMM registers)

DEST[31:0] ← SRC2[31:0]

DEST[127:32] ← SRC1[127:32]

DEST[VLMAX-1:128] ← 0

VMOVSS (VEX.NDS.128.F3.0F 10 /r when the source operand is memory and the destination is an XMM register)

DEST[31:0] ← SRC[31:0]

DEST[VLMAX-1:32] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVSS: `__m128 _mm_load_ss(float * p)`

MOVSS: `void _mm_store_ss(float * p, __m128 a)`

MOVSS: `__m128 _mm_move_ss(__m128 a, __m128 b)`

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

MOVsx/MOVsxD—Move with Sign-Extension

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF BE /r	MOVsx r16, r/m8	RM	Valid	Valid	Move byte to word with sign-extension.
OF BE /r	MOVsx r32, r/m8	RM	Valid	Valid	Move byte to doubleword with sign-extension.
REX + OF BE /r	MOVsx r64, r/m8*	RM	Valid	N.E.	Move byte to quadword with sign-extension.
OF BF /r	MOVsx r32, r/m16	RM	Valid	Valid	Move word to doubleword, with sign-extension.
REX.W + OF BF /r	MOVsx r64, r/m16	RM	Valid	N.E.	Move word to quadword with sign-extension.
REX.W** + 63 /r	MOVsxD r64, r/m32	RM	Valid	N.E.	Move doubleword to quadword with sign-extension.

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.

** The use of MOVsxD without REX.W in 64-bit mode is discouraged, Regular MOV should be used instead of using MOVsxD without REX.W.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Op/En RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 7-6 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). The size of the converted value depends on the operand-size attribute.

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

Operation

DEST ← SignExtend(SRC);

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.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

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.
- #UD If the LOCK prefix is used.

MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 10 /r MOVUPD <i>xmm1</i> , <i>xmm2/m128</i>		RM	V/V	SSE2 Move packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.128.66.0F.WIG 10 /r VMOVUPD <i>xmm1</i> , <i>xmm2/m128</i>		RM	V/V	AVX Move unaligned packed double-precision floating-point from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.256.66.0F.WIG 10 /r VMOVUPD <i>ymm1</i> , <i>ymm2/m256</i>		RM	V/V	AVX Move unaligned packed double-precision floating-point from <i>ymm2/mem</i> to <i>ymm1</i> .
66 0F 11 /r MOVUPD <i>xmm2/m128</i> , <i>xmm</i>		MR	V/V	SSE2 Move packed double-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.66.0F.WIG 11 /r VMOVUPD <i>xmm2/m128</i> , <i>xmm1</i>		MR	V/V	AVX Move unaligned packed double-precision floating-point from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.66.0F.WIG 11 /r VMOVUPD <i>ymm2/m256</i> , <i>ymm1</i>		MR	V/V	AVX Move unaligned packed double-precision floating-point from <i>ymm1</i> to <i>ymm2/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

128-bit versions:

Moves a double quadword containing two packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, store the contents of an XMM register into a 128-bit memory location, or move data between two XMM registers.

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

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.¹

To move double-precision floating-point values to and from memory locations that are known to be aligned on 16-byte boundaries, use the MOVAPD instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

1. If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the operand is not aligned on an 8-byte boundary.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

MOVUPD (128-bit load and register-copy form Legacy SSE version)

DEST[127:0] \leftarrow SRC[127:0]
 DEST[VLMAX-1:128] (Unmodified)

(V)MOVUPD (128-bit store form)

DEST[127:0] \leftarrow SRC[127:0]

VMOVUPD (VEX.128 encoded version)

DEST[127:0] \leftarrow SRC[127:0]
 DEST[VLMAX-1:128] \leftarrow 0

VMOVUPD (VEX.256 encoded version)

DEST[255:0] \leftarrow SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVUPD:     __m128 _mm_loadu_pd(double * p)
MOVUPD:     void _mm_storeu_pd(double *p, __m128 a)
VMOVUPD:    __m256d _mm256_loadu_pd (__m256d * p);
VMOVUPD:    _mm256_storeu_pd(__m256d *p, __m256d a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4

Note treatment of #AC varies; additionally

#UD If VEX.vvvv != 1111B.

MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 10 /r MOVUPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Move packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.128.0F.WIG 10 /r VMOVUPS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Move unaligned packed single-precision floating-point from <i>xmm2/mem</i> to <i>xmm1</i> .
VEX.256.0F.WIG 10 /r VMOVUPS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Move unaligned packed single-precision floating-point from <i>ymm2/mem</i> to <i>ymm1</i> .
OF 11 /r MOVUPS <i>xmm2/m128, xmm1</i>	MR	V/V	SSE	Move packed single-precision floating-point values from <i>xmm1</i> to <i>xmm2/m128</i> .
VEX.128.0F.WIG 11 /r VMOVUPS <i>xmm2/m128, xmm1</i>	MR	V/V	AVX	Move unaligned packed single-precision floating-point from <i>xmm1</i> to <i>xmm2/mem</i> .
VEX.256.0F.WIG 11 /r VMOVUPS <i>ymm2/m256, ymm1</i>	MR	V/V	AVX	Move unaligned packed single-precision floating-point from <i>ymm1</i> to <i>ymm2/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

128-bit versions: Moves a double quadword containing four packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, store the contents of an XMM register into a 128-bit memory location, or move data between two XMM registers.

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

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.¹

To move packed single-precision floating-point values to and from memory locations that are known to be aligned on 16-byte boundaries, use the MOVAPS instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

1. If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the operand is not aligned on an 8-byte boundary.

Operation

MOVUPS (128-bit load and register-copy form Legacy SSE version)

DEST[127:0] \leftarrow SRC[127:0]
 DEST[VLMAX-1:128] (Unmodified)

(V)MOVUPS (128-bit store form)

DEST[127:0] \leftarrow SRC[127:0]

VMOVUPS (VEX.128 encoded load-form)

DEST[127:0] \leftarrow SRC[127:0]
 DEST[VLMAX-1:128] \leftarrow 0

VMOVUPS (VEX.256 encoded version)

DEST[255:0] \leftarrow SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVUPS:    __m128 _mm_loadu_ps(double * p)
MOVUPS:    void _mm_storeu_ps(double *p, __m128 a)
VMOVUPS:   __m256 _mm256_loadu_ps (__m256 * p);
VMOVUPS:   _mm256_storeu_ps(__m256 *p, __m256 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4

Note treatment of #AC varies; additionally

#UD If VEX.vvvv != 1111B.

MOVZX—Move with Zero-Extend

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F B6 /r	MOVZX r16, r/m8	RM	Valid	Valid	Move byte to word with zero-extension.
0F B6 /r	MOVZX r32, r/m8	RM	Valid	Valid	Move byte to doubleword, zero-extension.
REX.W + 0F B6 /r	MOVZX r64, r/m8*	RM	Valid	N.E.	Move byte to quadword, zero-extension.
0F B7 /r	MOVZX r32, r/m16	RM	Valid	Valid	Move word to doubleword, zero-extension.
REX.W + 0F B7 /r	MOVZX r64, r/m16	RM	Valid	N.E.	Move word to quadword, zero-extension.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value. The size of the converted value depends on the operand-size attribute.

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

Operation

DEST ← ZeroExtend(SRC);

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.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

MPSADBW – Compute Multiple Packed Sums of Absolute Difference

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 42 /r ib MPSADBW <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in <i>xmm1</i> and <i>xmm2/m128</i> and writes the results in <i>xmm1</i> . Starting offsets within <i>xmm1</i> and <i>xmm2/m128</i> are determined by <i>imm8</i> .
VEX.NDS.128.66.0F3A.WIG 42 /r ib VMPSADBW <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and writes the results in <i>xmm1</i> . Starting offsets within <i>xmm2</i> and <i>xmm3/m128</i> are determined by <i>imm8</i> .
VEX.NDS.256.66.0F3A.WIG 42 /r ib VMPSADBW <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX2	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in <i>xmm2</i> and <i>ymm3/m128</i> and writes the results in <i>ymm1</i> . Starting offsets within <i>ymm2</i> and <i>xmm3/m128</i> are determined by <i>imm8</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(V)MPSADBW sums the absolute difference of 4 unsigned bytes (block_2) in the second source operand with sequential groups of 4 unsigned bytes (block_1) in the first source operand. The immediate byte provides bit fields that specify the initial offset of block_1 within the first source operand, and the offset of block_2 within the second source operand. The offset granularity in both source operands are 32 bits. The sum-absolute-difference (SAD) operation is repeated 8 times for (V)MPSADW between the same block_2 (fixed offset within the second source operand) and a variable block_1 (offset is shifted by 8 bits for each SAD operation) in the first source operand. Each 16-bit result of eight SAD operations is written to the respective word in the destination operand.

128-bit Legacy SSE version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source operand and destination operand are the same. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. Bits 7:3 of the immediate byte are ignored.

VEX.128 encoded version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (127:128) of the corresponding YMM register are zeroed. Bits 7:3 of the immediate byte are ignored.

VEX.256 encoded version: The sum-absolute-difference (SAD) operation is repeated 8 times for MPSADW between the same block_2 (fixed offset within the second source operand) and a variable block_1 (offset is shifted by 8 bits for each SAD operation) in the first source operand. Each 16-bit result of eight SAD operations between block_2 and block_1 is written to the respective word in the lower 128 bits of the destination operand.

Additionally, VMPSADBW performs another eight SAD operations on block_4 of the second source operand and block_3 of the first source operand. (Imm8[4:3]*32 + 128) specifies the bit offset of block_4 within the second source operand. (Imm[5]*32+128) specifies the initial bit offset of the block_3 within the first source operand. Each 16-bit result of eight SAD operations between block_4 and block_3 is written to the respective word in the upper 128 bits of the destination operand.

The first source operand is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits 7:6 of the immediate byte are ignored.

Note: If VMPSADBW is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

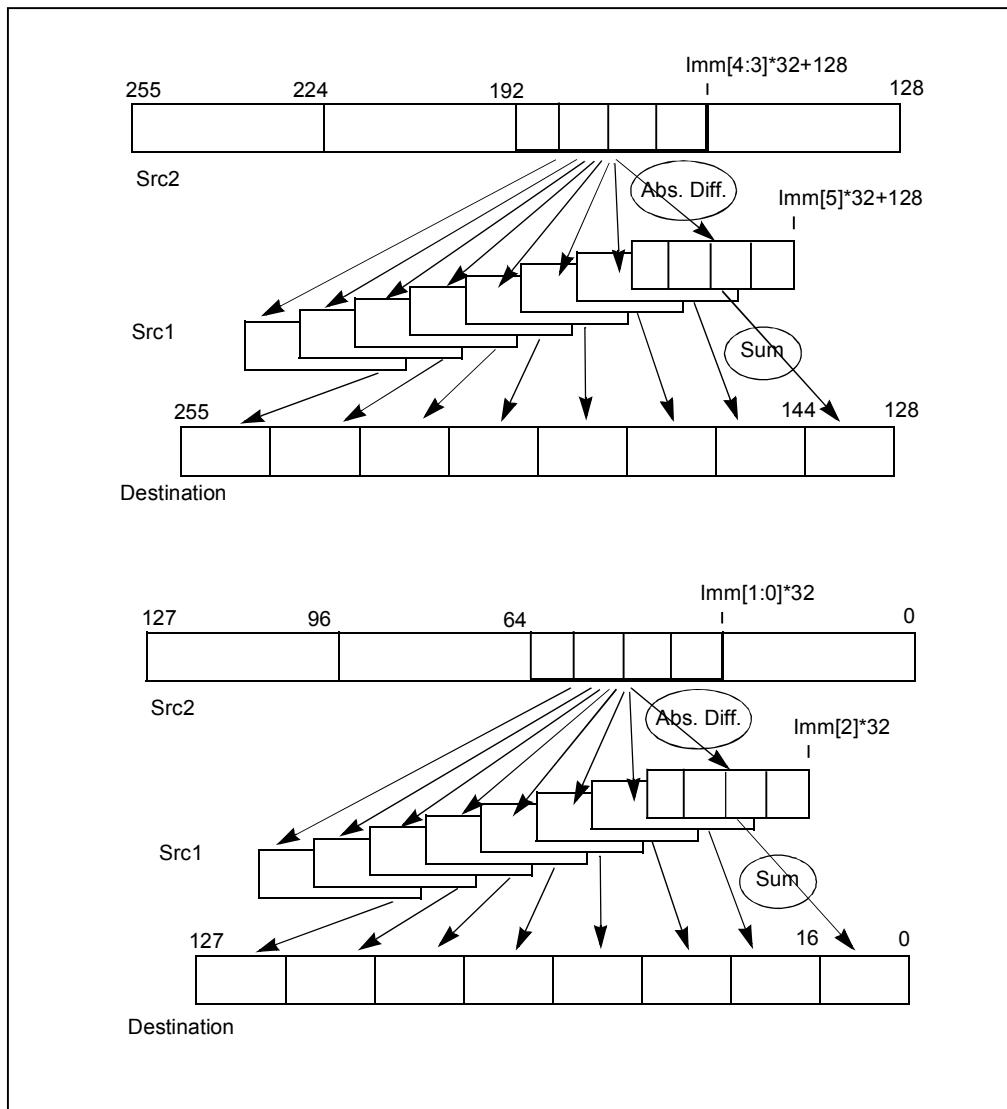


Figure 3-27. VMPSADBW Operation

Operation

VMPSADBW (VEX.256 encoded version)

```

SRC2_OFFSET <- imm8[1:0]*32
SRC1_OFFSET <- imm8[2]*32
SRC1_BYTEn <- SRC1[SRC1_OFFSET+7:SRC1_OFFSET]
SRC1_BYTEn <- SRC1[SRC1_OFFSET+15:SRC1_OFFSET+8]
SRC1_BYTEn <- SRC1[SRC1_OFFSET+23:SRC1_OFFSET+16]
SRC1_BYTEn <- SRC1[SRC1_OFFSET+31:SRC1_OFFSET+24]
SRC1_BYTEn <- SRC1[SRC1_OFFSET+39:SRC1_OFFSET+32]
SRC1_BYTEn <- SRC1[SRC1_OFFSET+47:SRC1_OFFSET+40]
  
```

```

SRC1_BYT6 ← SRC1[SRC1_OFFSET+55:SRC1_OFFSET+48]
SRC1_BYT7 ← SRC1[SRC1_OFFSET+63:SRC1_OFFSET+56]
SRC1_BYT8 ← SRC1[SRC1_OFFSET+71:SRC1_OFFSET+64]
SRC1_BYT9 ← SRC1[SRC1_OFFSET+79:SRC1_OFFSET+72]
SRC1_BYT10 ← SRC1[SRC1_OFFSET+87:SRC1_OFFSET+80]
SRC2_BYT0 ← SRC2[SRC2_OFFSET+7:SRC2_OFFSET]
SRC2_BYT1 ← SRC2[SRC2_OFFSET+15:SRC2_OFFSET+8]
SRC2_BYT2 ← SRC2[SRC2_OFFSET+23:SRC2_OFFSET+16]
SRC2_BYT3 ← SRC2[SRC2_OFFSET+31:SRC2_OFFSET+24]

```

```

TEMPO ← ABS(SRC1_BYT0 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT1 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT2 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT3 - SRC2_BYT3)
DEST[15:0] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT1 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT2 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT3 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT4 - SRC2_BYT3)
DEST[31:16] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT2 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT3 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT4 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT5 - SRC2_BYT3)
DEST[47:32] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT3 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT4 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT5 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT6 - SRC2_BYT3)
DEST[63:48] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT4 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT5 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT6 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT7 - SRC2_BYT3)
DEST[79:64] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT5 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT6 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT7 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT8 - SRC2_BYT3)
DEST[95:80] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT6 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT7 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT8 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT9 - SRC2_BYT3)
DEST[111:96] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT7 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT8 - SRC2_BYT1)

```

INSTRUCTION SET REFERENCE, A-M

```

TEMP2 ← ABS(SRC1_BYTEn - SRC2_BYTEn)
TEMP3 ← ABS(SRC1_BYTEn+1 - SRC2_BYTEn+1)
DEST[127:112] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

SRC2_OFFSET ← imm8[4:3]*32 + 128
SRC1_OFFSET ← imm8[5]*32 + 128
SRC1_BYTEn ← SRC1[SRC1_OFFSET+7:SRC1_OFFSET]
SRC1_BYTEn+1 ← SRC1[SRC1_OFFSET+15:SRC1_OFFSET+8]
SRC1_BYTEn+2 ← SRC1[SRC1_OFFSET+23:SRC1_OFFSET+16]
SRC1_BYTEn+3 ← SRC1[SRC1_OFFSET+31:SRC1_OFFSET+24]
SRC1_BYTEn+4 ← SRC1[SRC1_OFFSET+39:SRC1_OFFSET+32]
SRC1_BYTEn+5 ← SRC1[SRC1_OFFSET+47:SRC1_OFFSET+40]
SRC1_BYTEn+6 ← SRC1[SRC1_OFFSET+55:SRC1_OFFSET+48]
SRC1_BYTEn+7 ← SRC1[SRC1_OFFSET+63:SRC1_OFFSET+56]
SRC1_BYTEn+8 ← SRC1[SRC1_OFFSET+71:SRC1_OFFSET+64]
SRC1_BYTEn+9 ← SRC1[SRC1_OFFSET+79:SRC1_OFFSET+72]
SRC1_BYTEn+10 ← SRC1[SRC1_OFFSET+87:SRC1_OFFSET+80]

```

```

SRC2_BYTEn ← SRC2[SRC2_OFFSET+7:SRC2_OFFSET]
SRC2_BYTEn+1 ← SRC2[SRC2_OFFSET+15:SRC2_OFFSET+8]
SRC2_BYTEn+2 ← SRC2[SRC2_OFFSET+23:SRC2_OFFSET+16]
SRC2_BYTEn+3 ← SRC2[SRC2_OFFSET+31:SRC2_OFFSET+24]

```

```

TEMPO ← ABS(SRC1_BYTEn - SRC2_BYTEn)
TEMP1 ← ABS(SRC1_BYTEn+1 - SRC2_BYTEn+1)
TEMP2 ← ABS(SRC1_BYTEn+2 - SRC2_BYTEn+2)
TEMP3 ← ABS(SRC1_BYTEn+3 - SRC2_BYTEn+3)
DEST[143:128] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYTEn+4 - SRC2_BYTEn+4)
TEMP1 ← ABS(SRC1_BYTEn+5 - SRC2_BYTEn+5)
TEMP2 ← ABS(SRC1_BYTEn+6 - SRC2_BYTEn+6)
TEMP3 ← ABS(SRC1_BYTEn+7 - SRC2_BYTEn+7)
DEST[159:144] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYTEn+8 - SRC2_BYTEn+8)
TEMP1 ← ABS(SRC1_BYTEn+9 - SRC2_BYTEn+9)
TEMP2 ← ABS(SRC1_BYTEn+10 - SRC2_BYTEn+10)
TEMP3 ← ABS(SRC1_BYTEn+11 - SRC2_BYTEn+11)
DEST[175:160] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYTEn+12 - SRC2_BYTEn+12)
TEMP1 ← ABS(SRC1_BYTEn+13 - SRC2_BYTEn+13)
TEMP2 ← ABS(SRC1_BYTEn+14 - SRC2_BYTEn+14)
TEMP3 ← ABS(SRC1_BYTEn+15 - SRC2_BYTEn+15)
DEST[191:176] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYTEn+16 - SRC2_BYTEn+16)
TEMP1 ← ABS(SRC1_BYTEn+17 - SRC2_BYTEn+17)
TEMP2 ← ABS(SRC1_BYTEn+18 - SRC2_BYTEn+18)
TEMP3 ← ABS(SRC1_BYTEn+19 - SRC2_BYTEn+19)
DEST[207:192] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```
TEMPO ← ABS(SRC1_BYTEn+20 - SRC2_BYTEn+20)
```

```

TEMP1 ← ABS(SRC1_BYT6 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT7 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT8 - SRC2_BYT3)
DEST[223:208] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT6 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT7 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT8 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT9 - SRC2_BYT3)
DEST[239:224] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT7 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT8 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT9 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT10 - SRC2_BYT3)
DEST[255:240] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

VMPSADBW (VEX.128 encoded version)

```

SRC2_OFFSET ← imm8[1:0]*32
SRC1_OFFSET ← imm8[2]*32
SRC1_BYT0 ← SRC1[SRC1_OFFSET+7:SRC1_OFFSET]
SRC1_BYT1 ← SRC1[SRC1_OFFSET+15:SRC1_OFFSET+8]
SRC1_BYT2 ← SRC1[SRC1_OFFSET+23:SRC1_OFFSET+16]
SRC1_BYT3 ← SRC1[SRC1_OFFSET+31:SRC1_OFFSET+24]
SRC1_BYT4 ← SRC1[SRC1_OFFSET+39:SRC1_OFFSET+32]
SRC1_BYT5 ← SRC1[SRC1_OFFSET+47:SRC1_OFFSET+40]
SRC1_BYT6 ← SRC1[SRC1_OFFSET+55:SRC1_OFFSET+48]
SRC1_BYT7 ← SRC1[SRC1_OFFSET+63:SRC1_OFFSET+56]
SRC1_BYT8 ← SRC1[SRC1_OFFSET+71:SRC1_OFFSET+64]
SRC1_BYT9 ← SRC1[SRC1_OFFSET+79:SRC1_OFFSET+72]
SRC1_BYT10 ← SRC1[SRC1_OFFSET+87:SRC1_OFFSET+80]

```

```

SRC2_BYT0 ← SRC2[SRC2_OFFSET+7:SRC2_OFFSET]
SRC2_BYT1 ← SRC2[SRC2_OFFSET+15:SRC2_OFFSET+8]
SRC2_BYT2 ← SRC2[SRC2_OFFSET+23:SRC2_OFFSET+16]
SRC2_BYT3 ← SRC2[SRC2_OFFSET+31:SRC2_OFFSET+24]

```

```

TEMPO ← ABS(SRC1_BYT0 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT1 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT2 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT3 - SRC2_BYT3)
DEST[15:0] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT1 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT2 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT3 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT4 - SRC2_BYT3)
DEST[31:16] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT2 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT3 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT4 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT5 - SRC2_BYT3)
DEST[47:32] ← TEMPO + TEMP1 + TEMP2 + TEMP3

```

```

TEMPO ← ABS(SRC1_BYT0 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT1 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT2 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT3 - SRC2_BYT3)
DEST[63:48] ← TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO ← ABS(SRC1_BYT4 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT5 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT6 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT7 - SRC2_BYT3)
DEST[79:64] ← TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO ← ABS(SRC1_BYT5 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT6 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT7 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT8 - SRC2_BYT3)
DEST[95:80] ← TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO ← ABS(SRC1_BYT6 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT7 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT8 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT9 - SRC2_BYT3)
DEST[111:96] ← TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO ← ABS(SRC1_BYT7 - SRC2_BYT0)
TEMP1 ← ABS(SRC1_BYT8 - SRC2_BYT1)
TEMP2 ← ABS(SRC1_BYT9 - SRC2_BYT2)
TEMP3 ← ABS(SRC1_BYT10 - SRC2_BYT3)
DEST[127:112] ← TEMPO + TEMP1 + TEMP2 + TEMP3
DEST[VLMAX-1:128] ← 0

MPSADBW (128-bit Legacy SSE version)
SRC_OFFSET ← imm8[1:0]*32
DEST_OFFSET ← imm8[2]*32
DEST_BYT0 ← DEST[DEST_OFFSET+7:DEST_OFFSET]
DEST_BYT1 ← DEST[DEST_OFFSET+15:DEST_OFFSET+8]
DEST_BYT2 ← DEST[DEST_OFFSET+23:DEST_OFFSET+16]
DEST_BYT3 ← DEST[DEST_OFFSET+31:DEST_OFFSET+24]
DEST_BYT4 ← DEST[DEST_OFFSET+39:DEST_OFFSET+32]
DEST_BYT5 ← DEST[DEST_OFFSET+47:DEST_OFFSET+40]
DEST_BYT6 ← DEST[DEST_OFFSET+55:DEST_OFFSET+48]
DEST_BYT7 ← DEST[DEST_OFFSET+63:DEST_OFFSET+56]
DEST_BYT8 ← DEST[DEST_OFFSET+71:DEST_OFFSET+64]
DEST_BYT9 ← DEST[DEST_OFFSET+79:DEST_OFFSET+72]
DEST_BYT10 ← DEST[DEST_OFFSET+87:DEST_OFFSET+80]

SRC_BYT0 ← SRC[SRC_OFFSET+7:SRC_OFFSET]
SRC_BYT1 ← SRC[SRC_OFFSET+15:SRC_OFFSET+8]
SRC_BYT2 ← SRC[SRC_OFFSET+23:SRC_OFFSET+16]
SRC_BYT3 ← SRC[SRC_OFFSET+31:SRC_OFFSET+24]

TEMPO ← ABS(DEST_BYT0 - SRC_BYT0)
TEMP1 ← ABS(DEST_BYT1 - SRC_BYT1)

```

```
TEMP2 ← ABS( DEST_BYTE2 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE3 - SRC_BYTE3)
DEST[15:0] ← TEMPO + TEMP1 + TEMP2 + TEMP3
```

```
TEMPO ← ABS( DEST_BYTE1 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE2 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE3 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE4 - SRC_BYTE3)
DEST[31:16] ← TEMPO + TEMP1 + TEMP2 + TEMP3
```

```
TEMPO ← ABS( DEST_BYTE2 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE3 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE4 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE5 - SRC_BYTE3)
DEST[47:32] ← TEMPO + TEMP1 + TEMP2 + TEMP3
```

```
TEMPO ← ABS( DEST_BYTE3 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE4 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE5 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE6 - SRC_BYTE3)
DEST[63:48] ← TEMPO + TEMP1 + TEMP2 + TEMP3
```

```
TEMPO ← ABS( DEST_BYTE4 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE5 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE6 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE7 - SRC_BYTE3)
DEST[79:64] ← TEMPO + TEMP1 + TEMP2 + TEMP3
```

```
TEMPO ← ABS( DEST_BYTE5 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE6 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE7 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE8 - SRC_BYTE3)
DEST[95:80] ← TEMPO + TEMP1 + TEMP2 + TEMP3
```

```
TEMPO ← ABS( DEST_BYTE6 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE7 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE8 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE9 - SRC_BYTE3)
DEST[111:96] ← TEMPO + TEMP1 + TEMP2 + TEMP3
```

```
TEMPO ← ABS( DEST_BYTE7 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE8 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE9 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE10 - SRC_BYTE3)
DEST[127:112] ← TEMPO + TEMP1 + TEMP2 + TEMP3
DEST[VLMAX-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)MPSADBW: `__m128i _mm_mpsadbw_epu8 (__m128i s1, __m128i s2, const int mask);`
 VMPSADBW: `__m256i _mm256_mpsadbw_epu8 (__m256i s1, __m256i s2, const int mask);`

Flags Affected

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

MUL—Unsigned Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /4	MUL r/m8	M	Valid	Valid	Unsigned multiply ($AX \leftarrow AL * r/m8$).
REX + F6 /4	MUL r/m8*	M	Valid	N.E.	Unsigned multiply ($AX \leftarrow AL * r/m8$).
F7 /4	MUL r/m16	M	Valid	Valid	Unsigned multiply ($DX:AX \leftarrow AX * r/m16$).
F7 /4	MUL r/m32	M	Valid	Valid	Unsigned multiply ($EDX:EAX \leftarrow EAX * r/m32$).
REX.W + F7 /4	MUL r/m64	M	Valid	N.E.	Unsigned multiply ($RDX:RAX \leftarrow RAX * r/m64$).

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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in Table 3-66.

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Table 3-66. MUL Results

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX
Quadword	RAX	r/m64	RDX:RAX

Operation

```

IF (Byte operation)
  THEN
    AX ← AL * SRC;
  ELSE (* Word or doubleword operation *)
    IF OperandSize = 16
      THEN
        DX:AX ← AX * SRC;
      ELSE IF OperandSize = 32
        THEN EDX:EAX ← EAX * SRC; Fl;
      ELSE (* OperandSize = 64 *)
        RDX:RAX ← RAX * SRC;
    Fl;
Fl;

```

Flags Affected

The OF and CF flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are 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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.
#UD	If the LOCK prefix is used.

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.

MULPD—Multiply Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 59 /r MULPD <i>xmm1, xmm2/m128</i>		RM V/V	SSE2	Multiply packed double-precision floating-point values in <i>xmm2/m128</i> by <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 59 /r VMULPD <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Multiply packed double-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG 59 /r VMULPD <i>yymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Multiply packed double-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply of the two or four packed double-precision floating-point values from the source operand (second operand) and 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, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the destination YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

MULPD (128-bit Legacy SSE version)

```
DEST[63:0] ← DEST[63:0] * SRC[63:0]
DEST[127:64] ← DEST[127:64] * SRC[127:64]
DEST[VLMAX-1:128] (Unmodified)
```

VMULPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64] * SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```

VMULPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64] * SRC2[127:64]
DEST[191:128] ← SRC1[191:128] * SRC2[191:128]
DEST[255:192] ← SRC1[255:192] * SRC2[255:192]
```

Intel C/C++ Compiler Intrinsic Equivalent

MULPD: `__m128d _mm_mul_pd (m128d a, m128d b)`

VMULPD: `__m256d _mm256_mul_pd (__m256d a, __m256d b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2

MULPS—Multiply Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
OF 59 /r MULPS <i>xmm1, xmm2/m128</i>		RM V/V	SSE	Multiply packed single-precision floating-point values in <i>xmm2/mem</i> by <i>xmm1</i> .
VEX.NDS.128.0F.WIG 59 /r VMULPS <i>xmm1,xmm2, xmm3/m128</i>		RVM V/V	AVX	Multiply packed single-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.0F.WIG 59 /r VMULPS <i>ymm1, ymm2, ymm3/m256</i>		RVM V/V	AVX	Multiply packed single-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. 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, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (VLMAX-1:128) of the destination YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

MULPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[VLMAX-1:128] (Unmodified)
```

VMULPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[VLMAX-1:128] ← 0
```

VMULPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0]$
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] * \text{SRC2}[63:32]$
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64]$
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] * \text{SRC2}[127:96]$
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] * \text{SRC2}[159:128]$
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] * \text{SRC2}[191:160]$
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] * \text{SRC2}[223:192]$
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] * \text{SRC2}[255:224].$

Intel C/C++ Compiler Intrinsic Equivalent

MULPS: `__m128 _mm_mul_ps(__m128 a, __m128 b)`
 VMULPS: `__m256 _mm256_mul_ps (__m256 a, __m256 b);`

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2

MULSD—Multiply Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 OF 59 /r MULSD <i>xmm1, xmm2/m64</i>		RM V/V	SSE2	Multiply the low double-precision floating-point value in <i>xmm2/mem64</i> by low double-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F2.OF.WIG 59/r VMULSD <i>xmm1,xmm2, xmm3/m64</i>		RVM V/V	AVX	Multiply the low double-precision floating-point value in <i>xmm3/mem64</i> by low double precision floating-point value in <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the low double-precision floating-point value in the source operand (second operand) by 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, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

MULSD (128-bit Legacy SSE version)

DEST[63:0] \leftarrow DEST[63:0] * SRC[63:0]
DEST[VLMAX-1:64] (Unmodified)

VMULSD (VEX.128 encoded version)

DEST[63:0] \leftarrow SRC1[63:0] * SRC2[63:0]
DEST[127:64] \leftarrow SRC1[127:64]
DEST[VLMAX-1:128] \leftarrow 0

Intel C/C++ Compiler Intrinsic Equivalent

MULSD: $\text{__m128d}_\text{mm_mul_sd}(\text{m128d } \text{a}, \text{m128d } \text{b})$

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3

MULSS—Multiply Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 OF 59 /r MULSS <i>xmm1, xmm2/m32</i>		RM V/V	SSE	Multiply the low single-precision floating-point value in <i>xmm2/mem</i> by the low single-precision floating-point value in <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 59 /r VMULSS <i>xmm1,xmm2, xmm3/m32</i>	RVM	V/V	AVX	Multiply the low single-precision floating-point value in <i>xmm3/mem</i> by the low single-precision floating-point value in <i>xmm2</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the low single-precision floating-point value from the source operand (second operand) by 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, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation**MULSS (128-bit Legacy SSE version)**

$DEST[31:0] \leftarrow DEST[31:0] * SRC[31:0]$
 $DEST[VLMAX-1:32] \text{ (Unmodified)}$

VMULSS (VEX.128 encoded version)

$DEST[31:0] \leftarrow SRC1[31:0] * SRC2[31:0]$
 $DEST[127:32] \leftarrow SRC1[127:32]$
 $DEST[VLMAX-1:128] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

MULSS: $__m128_mm_mul_ss(__m128\ a, __m128\ b)$

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3

MULX – Unsigned Multiply Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.F2.0F38.W0 F6 /r MULX r32a, r32b, r/m32	RVM	V/V	BMI2	Unsigned multiply of $r/m32$ with EDX without affecting arithmetic flags.
VEX.NDD.LZ.F2.0F38.W1 F6 /r MULX r64a, r64b, r/m64	RVM	V/N.E.	BMI2	Unsigned multiply of $r/m64$ with RDX without affecting arithmetic flags.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (w)	ModRM:r/m (r)	RDX/EDX is implied 64/32 bits source

Description

Performs an unsigned multiplication of the implicit source operand (EDX/RDX) and the specified source operand (the third operand) and stores the low half of the result in the second destination (second operand), the high half of the result in the first destination operand (first operand), without reading or writing the arithmetic flags. This enables efficient programming where the software can interleave add with carry operations and multiplications.

If the first and second operand are identical, it will contain the high half of the multiplication result.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
// DEST1: ModRM:reg
// DEST2: VEX.vvvv
IF (OperandSize = 32)
    SRC1 ← EDX;
    DEST2 ← (SRC1*SRC2)[31:0];
    DEST1 ← (SRC1*SRC2)[63:32];
ELSE IF (OperandSize = 64)
    SRC1 ← RDX;
    DEST2 ← (SRC1*SRC2)[63:0];
    DEST1 ← (SRC1*SRC2)[127:64];
FI
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language when possible.

```
unsigned int mulx_u32(unsigned int a, unsigned int b, unsigned int * hi);
unsigned __int64 mulx_u64(unsigned __int64 a, unsigned __int64 b, unsigned __int64 * hi);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally
#UD If VEX.W = 1.

MWAIT—Monitor Wait

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 01 C9	MWAIT	NP	Valid	Valid	A hint that allow the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

MWAIT instruction provides hints to allow the processor to enter an implementation-dependent optimized state. There are two principal targeted usages: address-range monitor and advanced power management. Both usages of MWAIT require the use of the MONITOR instruction.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MWAIT may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MWAIT clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

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

ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. The first processors to implement MWAIT supported only the zero value for EAX and ECX. Later processors allowed setting ECX[0] to enable masked interrupts as break events for MWAIT (see below). Software can use the CPUID instruction to determine the extensions and hints supported by the processor.

MWAIT for Address Range Monitoring

For address-range monitoring, the MWAIT instruction operates with the MONITOR instruction. The two instructions allow the definition of an address at which to wait (MONITOR) and a implementation-dependent-optimized operation to commence at the wait address (MWAIT). The execution of MWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by MONITOR.

The following cause the processor to exit the implementation-dependent-optimized state: a store to the address range armed by the MONITOR instruction, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal. Other implementation-dependent events may also cause the processor to exit the implementation-dependent-optimized state.

In addition, an external interrupt causes the processor to exit the implementation-dependent-optimized state either (1) if the interrupt would be delivered to software (e.g., as it would be if HLT had been executed instead of MWAIT); or (2) if ECX[0] = 1. Software can execute MWAIT with ECX[0] = 1 only if CPUID.05H:ECX[bit 1] = 1. (Implementation-specific conditions may result in an interrupt causing the processor to exit the implementation-dependent-optimized state even if interrupts are masked and ECX[0] = 0.)

Following exit from the implementation-dependent-optimized state, control passes to the instruction following the MWAIT instruction. A pending interrupt that is not masked (including an NMI or an SMI) may be delivered before execution of that instruction. Unlike the HLT instruction, the MWAIT instruction does not support a restart at the MWAIT instruction following the handling of an SMI.

If the preceding MONITOR instruction did not successfully arm an address range or if the MONITOR instruction has not been executed prior to executing MWAIT, then the processor will not enter the implementation-dependent-optimized state. Execution will resume at the instruction following the MWAIT.

This document was created with Win2PDF available at <http://www.win2pdf.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.
This page will not be added after purchasing Win2PDF.